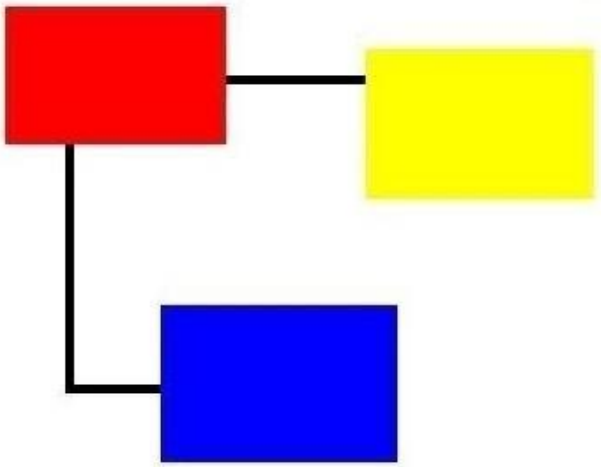
# FLOW-BASED PROGRAMMING

J. Paul Morrison



van Nostrand Reinhold, 1994

(material added, 2009)

Diagrams being upgraded, 2009

[Page intentionally left blank]

# Table of Contents

# Prologue

Some of my colleagues have suggested that I fill in some background to what you are going to read about in this book. So let me introduce myself.... I was born in London, England, just before the start of World War II, received what is sometimes referred to as a classical education, learned to speak several languages, including the usual dead ones, and studied Anthropology at King's College, Cambridge. I have since discovered that Alan Turing had attended my college, but while I was there I was learning to recognize Neanderthal skulls, and hearing Edmund Leach lecture about the Kachin, so I regret that I cannot claim to have programmed EDSAC, the machine being developed at Cambridge, although I later took an aptitude test for another marvelous machine, the Lyons' LEO (Lyons Electronic Office), whose design was based on EDSAC's. But maybe computing was in the air at King's!

In 1959 I joined IBM (UK) as an Electronic Data Processing Machines Representative. I had come into computers by a circuitous route: around the age of 12, I got bitten by the symbolic logic bug. This so intrigued me that all during my school and university years I read up on it, played with the concepts for myself, and looked forward to the time when all the world's problems would be solved by the judicious rearrangement of little mathematical symbols. Having also been fascinated by the diversity of human languages since childhood, the idea of really getting to the root of what things meant was very exciting. It wasn't until later in life that I realized that many great minds had tried this route without much success, and that, while it is certainly a beguiling concept and there have been many such attempts in earlier centuries, the universe of human experience is too complex and dynamic, with too many interacting factors, to be encoded in such a simple way. This does not mean that attempts to convey knowledge to a computer will not work - it is just that there seem to be certain built-in limitations. The human functions which we tend to think of as being simple, almost trivial, such as vision, speech comprehension or the ability to make one's way along a busy street, are often the hardest to explain to a computer. What we call common sense turns out to be quite uncommon....

While symbolic logic has not delivered on its original promise of making the world's important decisions simpler, it is perfectly adapted to the design of computers, and I became fascinated by the idea of machines which could perform logical operations. This fascination has stayed with me during my 33 years with the IBM Corporation in three different countries (by the way, this is why most of the systems I will be mentioning will be IBM systems - I apologize for this, but that's my background!), but I've always been struck by the apparent mismatch between the power of these machines and the difficulty of getting them to do what we wanted. I gradually came to concentrate on one basic problem: why should the process of developing applications on computers be so difficult, when they can obviously do anything we can figure out the rules for?

There is definitely an advantage to having cut my proverbial teeth in this field at a time when very few people had even heard of computers: over the intervening years I have had time to digest new concepts and see which of them succeeded and which failed. Over the course of three and a bit decades [this was written in 1994], many concepts, techniques and fads have sprung up with great attendant fanfare, and have either faded out or just become part of the regular curriculum. Ideas which took a decade or two to evolve are now familiar to kids fresh out of university. I got advance notice of many of these concepts, and often had time to understand them before they became widespread! A list of these wonders would be too long to include here, and perhaps only of interest to historians. Some of them fell by the wayside, but many of them are still around - some good and some not so good! We who were working in the field also certainly contributed our fair share of techniques and fads, also some good and some not so good!

I think my first enthusiasm was compiler compilers. I first worked with a fascinating system called BABEL - appropriate name - which was going to make it far easier to write compilers. I still use some of its ideas today, 30 years later. We shall see later in this book that there are interesting parallels between compiler theory and the subject matter of this book, and there seems to be an important role for what are sometimes called "mini-languages" (I will be talking some more about them in Chapter 17). Certainly compiler compilers comprise a piece of the answer, but they did not result in the productivity improvement that we were looking for.

I have also always been taken with interpreters - I believe my first exposure to these was BLIS (the Bell Laboratories Interpretive System), which made the 650 look like a sequential machine. Probably the characteristic of interpreters which really appeals to people is the ability to debug without having to change languages. Of course, some of the recent debugging tools are starting to bring this capability to the world of Higher Level Languages (HLLs), but the ability to just slot in a TYPE or "say" command and rerun a test is so impressive that all the languages which became really popular have always been interpreters, no matter how awkward the syntax! In a survey of machine cycle usage done a few years ago at IBM's Research Center at Yorktown Heights, they found that the vast majority of cycles were being used by CMS EXEC statements - strings of CMS commands glued together to do specific jobs of work.

Another important concept for productivity improvement is that of a reusable subroutine library. I also believe strongly that reuse is another key piece of the solution, but not exactly in the form in which we visualized it in those days. In company after company, I have seen people start up shared subroutine libraries with a fine flurry of enthusiasm, only to find the action slowing to a standstill after some 30 or 40 subroutines have been developed and made available. Some companies are claiming much higher numbers, but I suspect these are shops which measure progress, and reward their people, based on how many subroutines are created and added to the library, rather than on whether they are actually used. Although organizational and economic changes are also required to really capitalize on any form of reuse, I believe there is a more fundamental reason why these libraries never really take off, and that is the philosophy of the von Neumann machine. I will be going into this in more detail in Chapter 1, but I found I was able to predict which subroutines would land up in these libraries, and it was always "one moment in time" functions, e.g. binary search, date routines, various kinds of conversions. I tried to build an easy-to-use, general purpose update (yes, I really tried), and I just couldn't do it (except for supporting a tiny subset of all the possible variations)! This experience is what got me thinking about a radically different approach to producing reusable code. I hope that, as you read this book, you will agree that there is another approach, and that it is completely complementary to the old one.

Rapid prototyping and the related idea of iterative development were (and are still) another enthusiasm of mine. Rapid prototyping is a process of reducing the uncertainties in the development process by trying things out. I believe that *anything* you are uncertain about should be prototyped: complex algorithms, unfamiliar hardware, data base structures, human interfaces (especially!), and so on. I believe this technique will become even more important in the next few decades as we move into ever more complex environments. Here again, we will have to modify or even abandon the old methodologies. Dave Olson's 1993 book, *"Exploiting Chaos: Cashing in on the Realities of Software Development"*, describes a number of approaches to combining iterative development with milestones to get the best of both worlds, plus some fascinating digressions into the new concepts of chaos and "strange attractors". There are some very strange attractors in our business! I have also believed for some time that most prototypes should not just be thrown away once they have served their purpose. A prototype should be able to be "grown", step by step, into a full-fledged system. Since the importance of prototypes is that they reduce uncertainty, rewriting applications in a different language is liable to bring a lot of it back!

The pattern of all these innovations is always the same - from the subroutine to Object-Oriented Programming: someone finds a piece of the answer and we get a small increment in productivity, but not the big break-through we have been looking for, and eventually this technique makes its way into the general bag of tricks that every experienced programmer carries in his or her back pocket.

By the way, I should state at the outset that my focus is not on mathematical applications, but on

business applications - the former is a different ball-game, and one happily played by academics all over the world. Business applications are different, and much of my work has been to try to determine exactly why they should be so different, and what we can do to solve the problem of building and maintaining them. These kinds of applications often have a direct effect on the competitiveness of the companies that use them, and being able to build and maintain this type of application more effectively will be a win-win situation for those of us in the industry and for those who use our services.

Before I start to talk about a set of concepts which, based on my experience over the last 30 years, I think really does provide a quantum jump in improving application development productivity, I would like to mention something which arises directly out of my own personal background. Coming from an artistic background, I find I tend to think about things in visual terms. One of the influences in the work described in this book was a feeling that one should be able to express applications in a graphical notation which would take advantage of people's visualization abilities. This feeling may have been helped along by exposure to a system called GPSS (General Purpose Simulation System). This system can be highly graphical, and it (along with other simulation systems) has another very interesting characteristic, namely that its constructs tend to match objects in the real world. It is not surprising that Simula (another language originally designed for simulation) is viewed as one of the forerunners of many of today's advanced programming languages.

Another effect of my personal orientation is a desire, almost a preoccupation, with beauty in programming. While I will stress many times that programming should not be the production of unique pieces of cabinetry, this does not mean that programs cannot exhibit beauty. There are places and times in the world's history where people have invested great creativity in useful objects such as spoons or drinking cups. Conversely, the needs of primitive mass-production, supported by a naïve view of value, resulted in factories turning out vast numbers of identical, artistically crude objects (although obviously there were some exceptions), which in turn are thought to have led to a deadening of the sensibilities of a whole culture. I believe that modern technology therefore can do more than just make our lives more comfortable - I believe it can actually help to bring the aesthetic back into its proper place in our life experience.

One more comment about my personal biasses (of which I have many, so I'm told): it has always seemed to me that application design and building is predominantly a creative activity, and creativity is a uniquely human ability - one that (I believe) computers and robots will never exhibit. On the other hand, any activity which bores people should be done by computers, and will probably be done better by them. So the trick is to split work appropriately between humans and machines - it is the partnership between the two that can lead to the most satisfying and productive era the world has ever known (I also read a lot of science fiction!). One of the points often missed by the purveyors of methodologies is that each stage of refinement of a design is not simply an expansion of information already in existence, but a creative act. We should absolutely

avoid reentering the same information over and over again - that's boring! - but, on the other hand, we should never imagine that any stage of refinement of a design can somehow be magically done without human input. Robots are great at remembering and following rules - only humans create.

*Corollary I:* Do not use humans for jobs computers can do better - this is a waste of human energy and creativity, the only real resource on this planet, and demeans the human spirit.

*Corollary II*: Do not expect computers to provide that creative spark that only humans can provide. If computers ever do become creative, they won't be computers any more - they will be people! And I do not consider creativity the same as random number generation....

The other personal slant I brought to this quest was the result of a unique educational system which inculcated in its victims (sorry, students) the idea that there is really no area of human endeavour which one should be afraid to tackle, and that indeed we all could realistically expect to contribute to any field of knowledge we addressed. This perhaps outdated view may have led me to rush in where angels fear to tread.... However, this pursuit has at the least kept me entertained and given my professional life a certain direction for several decades.

In past centuries, the dilettante or amateur has contributed a great deal to the world's store of knowledge and beauty. Remember, most of the really big paradigm shifts were instigated by outsiders! The word "amateur" comes from the idea of loving. One should be proud to be called an computing amateur! "Dilettante" is another fine word with a similar flavour - it comes from an Italian word meaning "to delight in". I therefore propose another theorem: if an activity isn't fun, humans probably shouldn't be doing it. I feel people should use the feeling of fun as a touchstone to see if they are on the right track. Here is a quote from my colleague, P.R. Ewing, which also agrees with my own experience: "The guys who turn out the most code are the ones who are having fun!" Too many experts are deadly serious. Play is not something we have to put away when we reach the state of adulthood - it is a very important way for humans to expand their understanding of the universe and all the interesting and delightful beings that occupy it. This feeling that the subject matter of this book is fun is one of the most common reactions we have encountered, and is one of the main things which makes my collaborators and myself believe that we have stumbled on something important. In what follows I hope to convey some of this feeling. Please forgive me if some whimsy sneaks in now and then!

# Chap. I: Introduction

Imagine that you have a large and complex application running in your shop, and you discover that you need what looks like fairly complex changes made to it in a hurry. You consult your programmers and they tell you that the changes will probably take several months, but they will take a look. A meeting is called of all the people involved - not just programmers and analysts, but users and operations personnel as well. The essential logic of the program is put up on the wall, and the program designers walk through the program structure with the group. During the ensuing discussion, they realize that two new modules have to be written and some other ones have to change places. Total time to make the changes - a week!

Quite a few parts of this scenario sound unlikely, don't they? Users, operations people and programmers all talking the same language - unthinkable! But it actually did happen just the way I described. The factor that made this experience so different from most programmers' everyday experience is the truly revolutionary technology I will be describing in this book.

While this technology has been in use for productive work for the last 20 years, it has also been waiting in the wings, so to speak, for its right time to come on stage. Perhaps because there is a "paradigm shift" involved, to use Kuhn's phrase (Kuhn 1970), it has not been widely known up to now, but I believe now is the time to open it up to a wider public.

This technology provides a consistent application view from high level design all the way down to implementation. It requires applications to be built using reusable "black boxes" and encourages developers to construct such black boxes, which can then improve the productivity of other developers. It forces developers to focus on data and its transformations, rather than starting with procedural code. It encourages rapid prototyping and results in more reliable, more maintainable systems. It is compatible with distributed systems, and appears to be on a convergent path with Object-Oriented Programming. In this book, I will describe the concepts underlying this technology and give examples of experience gained using it. Does it sound too good to be true? You be the judge! In the following pages, we will be describing what I believe is

a genuine revolution in the process of creating application programs to support the data processing requirements of companies around the world.

Today, in the early 90's, the bulk of all business programming is done using techniques which have not changed much in 30 years. Most of it is done using what are often called Higher-Level Languages (HLLs), by far the most popular of which is COBOL, the COmmon Business-Oriented Language. A distant second is probably PL/I, not as widespread in terms of number of customers, but in use at some of the biggest organizations in North America. C appears to be gaining steadily in popularity, especially as it is often the first programming language students encounter at university. It appears to be especially convenient for writing system software, due to its powerful and concise pointer manipulation facilities, but by the same token, it may be less well adapted for writing business applications. Some languages are used by particular sectors of the programming community or for certain specialized purposes. There are also the "4th generation languages", which are higher level than the HLLs but usually more specialized.

There are plenty of design methodologies and front-end tools to do them with, but most of these do not really affect the mechanics of creating programs. After the design has been done, the programmer still has the job of converting his or her elegant design into strings of commands in the chosen programming language. Although generators have had some success, by and large most of today's programmers painstakingly create their programs by hand, like skilled artisans hand-crafting individual pieces of cabinetry. One of the "grand old men" of the computing fraternity, Nat Rochester, said a number of years ago that programming probably absorbs more creativity than any other professional pursuit, and most of it is invisible to the outside world. Things really haven't changed all that much since those days. There are also what might be called procedural or organizational approaches to improving the application development process, e.g. structured walk-throughs, the buddy system, chief programmer teams, third-party testing. My experience is that the approaches of this type which have been successful will still be valid whatever tool we eventually use for producing applications. However, if all you do is take the existing hand-crafting technology and add a massive bureaucracy to cross-check every chisel stroke and hammer blow, I believe you will only get minor improvements in your process, at a considerable cost in productivity and morale. What is needed instead is a fundamental change in the way we do things, after which we will be able to see which procedures and organizations fit naturally into the new world.

It is a truism that most businesses in the Western world would stop functioning if it were not for the efforts of tens of thousands, if not hundreds of thousands, of application programmers. These people are practising a craft which most of the population does not understand, and would not be willing to do if it did. The archetypal programmer is viewed as a brilliant but impractical individual who has a better rapport with computers than with people, slaving long hours at a terminal which is at the very least damaging to his or her eyesight. In fact, of course, the programmer is the key interface between his clients, who speak the language of business, and the

computer and its systems, which speak the language of electrons. The more effectively and reliably the programmer can bridge between these worlds, the better will be the applications which he or she builds, but this requires an unusual combination of talents. If you have any of these paragons in your organization, guard them like the treasures they are! In what follows, one of the recurring themes will be that the problems with today's programming technology arise almost entirely from the continuing mismatch between the problem area the programmer works in and the tools he or she has to work with. Only if we can narrow the gap between the world of users and that of application developers, can we produce applications which fit the needs of users and do it in a timely and cost-effective manner.

The significant fact I have come to realize over the last twenty years is that application programming in its present form really *is* hard and in fact has not progressed all that much since the days of the first computers. This lack of progress is certainly not due to any shortage of advocates of this or that shiny new tool, but very few of these wonder products have delivered what was promised. When I started in the business in 1959, we already had higher-level languages, interpreters and subroutine calls - these are still the basic tools of today's programming professionals. The kind of programming most of us do has its roots in the procedural programming that arose during the 40's and 50's: this new invention called a computer 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, 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.

In addition, as programmers build new systems, these add to the amount of resources being expended on maintaining them, to the point where the ability of many companies to develop new applications is being seriously impacted by the burden of maintaining old systems. This in turn adversely affects their ability to compete in the new competitive global market-place. Many writers have talked about the programming backlog - the backlog of programming work that DP departments are planning to do but can't get to because of lack of resources. I have also heard people use the phrase "hidden backlog" - this is programming work that users would like to get done but know there's no point in even talking to their DP department about, so it tends not to show up in the statistics! I think this is at least partly why non-DP departments have been buying up PCs in recent years - they feel that having their own machines will make them independent of the DP department, but of course this only means they face the same old programming problems on their own machines!

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 what follows really do have the potential to solve this problem, and I hope that, as you read this book, you will come to agree with us. However, they represent a true paradigm change which fundamentally changes the way we look at the programming process. Like many important discoveries, this new paradigm is basically quite simple, but far-reaching in its implications.

Mention of a new paradigm makes one immediately think of another new paradigm which is growing steadily in popularity, namely Object-Oriented Programming (usually abbreviated to OOP). What I am about to describe is not OOP, but bears certain similarities to it, and especially to the more advanced OOP concepts, specifically the concept of "active objects". In the long run, these two paradigms appear to be on a converging path, and, as I will be describing in a later chapter, I believe that it may well be possible to fuse the two sets of concepts to achieve the best of both worlds. In most of this book, however, I will be presenting our concepts and experience as they evolved historically, using our own terminology.

After a 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. From an arithmetic point of view, one seldom encounters a multiplication or division in business programming, let alone anything as arcane as a square root. The vast majority of business applications do such things as 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. Given what seems like a fairly simple problem space, I wondered why application development should be so arduous and why, once built, a program should be so hard to maintain. Over the last few years, 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.

This model is the traditional one that has been so productive over the last few decades, designed around a single instruction counter which walks sequentially through strings of codes deciding what to do at each step. These codes can be treated both as data (e.g. by compilers) and as commands. This design is usually, but not necessarily, combined with a uniform array of memory cells from which the instructions take data, and into which they return it. As described in a recent 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, they don't have enough mathematical education or they don't work hard

enough - I'm sure you've run into all of these explanations. I don't believe any of these are valid - I believe there is a far more fundamental problem - namely that, at a basic level, the medium is simply inappropriate for the task at hand. In fact, when you look at them objectively, the symptoms our business is experiencing now are quite characteristic of what one would expect when people try to do a complicated job using the wrong tools. Take a second and really try to imagine building a functioning automobile out of clay! It's highly malleable when wet, so you should be able to make anything, but after it has been fired it is very hard but very brittle! In fact that's quite a good analogy for the "feel" of most of our applications today!

The time is now ripe for a new paradigm to replace the von Neumann model as the bridging model between hardware and software. The one we will be describing is similar to the one Valiant proposes (I'll talk about his in more detail in Chapter 27) and in fact seems to be one of a family of related concepts which have appeared over the last few years in the literature. The common concept underlying much of this work is basically that, to solve these problems, we have to relax the tight sequential programming style characteristic of the von Neumann machine, and structure programs as collections of communicating, asynchronous processes. 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. We have tended to believe that the tight control of execution sequence imposed by this approach is the only way to get predictable code, and that therefore it was necessary for reliable systems. It turns out that machines (and people) work more efficiently if you only retain the constraints that matter and relax the ones that don't, and you can do this without any loss of reliability. The intent of this book is to try to describe a body of experience which has been built up using a particular set of implementations of this concept over the years, so I will not go into more detail at this point. In this chapter, we will be talking more about the history of this concept than about specific implementations or experience gained using them.

Another factor which makes me think it is timely for this technology to be made public is that we are facing a growing crisis in application development. At the same time as new requirements are appearing, the underlying technology is changing faster and faster. The set of concepts I will be describing seems to fit well with current directions for both software and hardware. Not only can it support in a natural manner the requirements of distributed, heterogeneous applications, but it also seems an appropriate programming technology for the new multiprocessor machines being worked on by universities and leading-edge computer manufacturers all over the world. As the late Wayne Stevens, the noted writer on the subject of application design methodologies, has pointed out in several of his articles (e.g. Stevens 1985), the paradigm we will be describing provides a consistent, natural way to view applications from the workings of whole companies all the way down to the smallest component. Since you can describe manual applications with data-flow diagrams, the connection between manual and system procedures can be shown seamlessly.

In what follows, I will be using the term "Flow-Based Programming" (or FBP for short) to describe this new set of concepts and the software needed to support it. We have in the past used the term "Data Flow" as it conveys a number of the more important aspects of this technology, but there is a sizable body of published work on what is called "dataflow architectures" in computer design and their associated software (for instance the very exciting work coming out of MIT), so the term dataflow may cause confusion in some academic circles. It was also pointed out to me a few years ago that, when control flow is needed explicitly, FBP can provide it by the use of such mechanisms as triggers, so the term Flow-Based Programming avoids the connotation that we cannot do control flow. This is not to say that the two types of data flow do not have many concepts in common - dataflow computer architectures arise also from the perception that the von Neumann machine design that has been so successful in the past must be generalized if we are to move forward, whether we are trying to perform truly huge amounts of computation such as weather calculations or simply produce applications which are easier to build and maintain.

One significant difference between the two schools, at least at this time, is that most of the other data flow work has been mathematically oriented, so it tends to work with numbers and arrays of numbers. Although my early data flow work during the late 60s also involved simple numeric values travelling through a network of function blocks, my experience with simulation systems led me to the realization that it would be more productive in business applications to have the things which flow be structured objects, which I called "entities". This name reflected the idea that these structured objects tended to represent entities in the outside world. (In our later work, we realized that the name "entity" might cause confusion with the idea of entities in data modelling, although there are points of resemblance, so we decided to use a different word). Such a system is also, not coincidentally, a natural design for *simulating* applications, so the distinction between applications and their simulations becomes much less significant than in conventional programming. You can think of an entity as being like a record in storage, but active (in that it triggers events), rather than passive (just being read or written). Entities flow through a network of processes, like cars in a city, or boats in a river system. They differ from the mathematical tokens of dataflow computers or my early work chiefly in that they have *structure*: each entity represents an object with attributes, for example an employee will have attributes such as salary, date of hire, manager, etc. As you read this book, it should become clear why there has to be at least one layer of the application where the entities move as individual units, although it may very well be possible to integrate the various dataflow approaches at lower levels.

At this point I am going to have to describe FBP briefly, to give the reader something to visualize, but first a caveat: the brief description that follows will probably not be enough to let you picture what FBP is and how it does it. If we don't do this at this point, however, experience shows that readers find it hard to relate what I am describing to their own knowledge. The reverse risk is that they may jump to conclusions which may prevent them from seeing what is

truly new about the concepts I will be describing later. I call this the "It's just..." syndrome.

In conventional programming, when you sit down to write a program, you write code down the page - a linear string of statements describing the series of actions you want the computer to execute. Since we are of course all writing structured code now, we start with a main line containing mostly subroutine calls, which can then be given "meaning" later by coding up the named subroutines. A number of people have speculated about the possibility of instead building a program by just plugging prewritten pieces of logic together. This has sometimes been called 'Legoland' programming. Even though that is essentially what we do when we use utilities, there has always been some doubt whether this approach has the power to construct large scale applications, and, if it has, whether such applications would perform. I now have the pleasure to announce that the answer is 'Yes' to both these questions!

The "glue" that FBP uses to connect the pieces together is an example of what Yale's Gelernter and Carriero (1992) have called a "coordination language". I feel the distinction between coordination languages and procedural languages is a useful one, and helps to clarify what is different about FBP. Conventional programming languages instruct the machine what logic to execute; coordination languages tell the machine how to coordinate multiple modules written in one or several programming languages. There is quite a bit of published material on various approaches to coordination, but much of that work involves the use of special-purpose languages, which reduces the applicability of these concepts to traditional languages and environments. Along with Gelernter and Carriero, I feel a better approach is to have a language-independent coordination notation, which can coordinate modules written in a variety of different procedural languages. The individual modules have to have a common Application Programming Interface to let them talk to the coordination software, but this can be relatively simple.

Coordination and modularity are two sides of the same coin, and several years ago Nate Edwards of IBM 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". Although I am not sure when Nate first brought the two words "configurable" and "modularity" together, the report on a planning session in Palo Alto in 1976 uses the term, and Nate's 1977 paper (Edwards 1977) contains both the terms "configurable architecture" and "controlled modularity". While Nate Edwards' work is fairly non-technical and pragmatic, his background is mainly in hardware, rather than software, which may be why his work has not received the attention it deserves. One of the important characteristics of a system exhibiting configurable modularity, such as most modern hardware or Flow-Based Programming, is that you can build systems out of "black box" reusable modules, much like the chips which are used to build logic in hardware. You also, of course, have to have something to connect them together with, but they do not have to be modified in any way to make this happen. Of course, this is characteristic of almost all the things we attach to each other in real life - in fact, almost everywhere except in

conventional programming. In FBP, these black boxes are the basic building blocks that a developer uses to build an application. New black boxes can be written as needed, but a developer tries to use what is available first, before creating new components. In FBP, the emphasis shifts from building everything new to connecting preexisting pieces and only building new when building a new component is cost-justified. Nate Edwards played a key role in getting the hardware people to follow this same principle - and now of course, like all great discoveries, it seems that we have always known this! We have to help software developers to move through the same paradigm shift. If you look at the literature of programming from this standpoint, you will be amazed at how few writers write from the basis of reuse - in fact the very term seems to suggest an element of surprise, as if reuse were a fortuitous occurrence that happens seldom and usually by accident! In real life, we *use* a knife or a fork - we don't *re*use it!

We will be describing similarities between FBP and other similar pieces of software in later chapters, but perhaps it would be useful at this point to use DOS pipes to draw a simple analogy. If you have used DOS you will know that you can take separate programs and combine them using a vertical bar (|), e.g.

```
A | B
```

This is a very simple form of what I have been calling coordination of separate programs. It tells the system that you want to feed the output of A into the input of B, but neither A nor B have to be modified to make this happen. A and B have to have connection points ("plugs" and "sockets") which the system can use, and of course there has to be some software which understands the vertical bar notation and knows what to do with it. FBP broadens this concept in a number of directions which vastly increase its power. It turns out that this generalization results in a very different approach to building applications, which results in systems which are both more reliable and more maintainable. In the following pages I hope to be able to prove this to your satisfaction!

The FBP systems which have been built over the last 20 years have therefore basically all had the following components:

- a number of precoded, pretested functions, provided in object code form, not source code form ("black boxes") - this set is open-ended and (hopefully) constantly growing
- a "Driver" - a piece of software which coordinates the different independent modules, and implements the API (Application Programming Interface) which is used by the components to communicate with each other and with the Driver
- a notation for specifying how the components are connected together into one or more networks (an FBP application designer starts with pictures, and then converts them into specifications to be executed by the Driver)
- this notation can be put into a file for execution by the Driver software. In the most successful implementation of FBP so far (DFDM - described in the next section of this

chapter), the network could either be compiled and link edited to produce an executable program, or it could be interpreted directly (with of course greater initialization overhead). In the interpreted mode, the components are loaded in dynamically, so you can make changes and see the results many times in a few minutes. As we said before, people find this mode extremely productive. Later, when debugging is finished, you can convert the interpretable form to the compilable form to provide better performance for your production version.

- procedures to enable you to convert, compile and package individual modules and partial networks
- documentation (reference and tutorial) for all of the above

In the above list I have not included education - but of course this is probably the most important item of all. To get the user started, there is a need for formal education - this may only take a few days or weeks, and I hope that this book will get the reader started on understanding many of the basic concepts. However, education also includes the practical experience that comes from working with many different applications, over a number of months or years. In this area especially, we have found that FBP feels very different from conventional programming. Unlike most other professions, in programming we tend to underestimate the value of experience, which may in fact be due to the nature of the present-day programming medium. In other professions we do not recommend giving a new practitioner a pile of books, and then telling him or her to go out and do brain surgery, build a bridge, mine gold or sail across the Atlantic. Instead it is expected that there will be a series of progressive steps from student or apprentice to master. Application development using FBP feels much more like an engineering-style discipline: we are mostly assembling structures out of preexisting components with well-defined specifications, rather than building things from scratch using basic raw materials. In such a medium, experience is key: it takes time to learn what components are available, how they fit together and what trade-offs can be made. However, unlike bridge-builders, application developers using FBP can also get simple applications working very fast, so they can have the satisfaction of seeing quite simple programs do non-trivial things very early. Education in FBP is a hands-on affair, and it is really a pleasure seeing people's reactions when they get something working without having to write a line of code!

Now that graphics hardware and software have become available at reasonable cost and performance, it seems very desirable 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 already been done along these lines and seems to bear this idea out. Many potential users of FBP systems will already have one or more graphical design tools, and, as we shall see, there is an especially good match between Structured Analysis and FBP, so that it seems feasible, and desirable, to base FBP graphical tools on existing graphical tools for doing Structured Analysis, with the appropriate information added for creating running FBP

programs.

Now I feel it would be useful to give you a bit of historical background on FBP: the first implementation of this concept was built by myself in 1969 and 1970 in Montreal, Quebec. This proved very productive - so much so that it was taken into a major Canadian company, where it was used for all the batch programming of a major on-line system. This system was called the Advanced Modular Processing System (AMPS). This system and the experience gained from it are described in a fair amount of detail in an article I wrote a few years later for the IBM Systems Journal (Morrison 1978). I am told this was the first article ever published in the Systems Journal by an author from what was then called the Americas/Far East area of IBM (comprising Canada, South America and the Far East).

Although the concepts are not well known, they have actually been in the public domain for many years. The way this happened is as follows: in late 1970 or early '71 I approached IBM Canada's Intellectual Property department to see if we could take out a patent on the basic idea. Their recommendation, which I feel was prescient, was that this concept seemed to them more like a law of nature, which is not patentable. They did recommend, however, that I write up a Technical Disclosure Bulletin (TDB), which was duly published and distributed to patent offices world-wide (Morrison 1971). A TDB is a sort of inverse patent - while a patent protects the owner but requires him or her to try to predict all possible variations on a concept, a TDB puts a concept into the public domain, and thereby protects the registering body from being restricted or impeded in the future in any use they may wish to make of the concept. In the case of a TDB, it places the onus on someone else who might be trying to patent something based on your concept to prove that their variation was not obvious to someone "skilled in the art".

Towards the end of the 80's, Wayne Stevens and I jointly developed a new version of this software, called the Data Flow Development Manager (DFDM). It is described in Appendix A of Wayne Stevens' latest book (Stevens 1991) (which, by the way, contains a lot of good material on application design techniques in general). What I usually refer to in what follows as "processes" were called "coroutines" in DFDM, after Conway (1963), who described an early form of this concept in a paper back in the 60's, and foresaw even then some of its potential. "Coroutine" is formed from the word "routine" together with the Latin prefix meaning "with", as compared with "subroutine", which is formed with the prefix meaning "under". (Think of "*co*operative" vs. "*sub*ordinate").

DFDM was used for a number of projects (between 40 and 50) of various sizes within IBM Canada. A few years later, Kenji Terao got a project started within IBM Japan to support us in developing an improved version for the Japanese market. This version is, at the time of writing, the only dialect of FBP which has been made available in the market-place, and I believe enormous credit is due to Kenji and all the dedicated and forward-looking people in IBM Japan who helped to make this happen. While this version of DFDM was in many ways more robust or "industrial strength" than the one which we had been using within IBM Canada, much of the

experience which I will be describing in the following pages is based on what we learned using the IBM Canada internal version of DFDM, or on the still earlier AMPS system. Perhaps someone will write a sequel to this book describing the Japanese experience with DFDM...

Last, but I hope not least, there is a PC-based system written in C, which attempts to embody many of the best ideas of its ancestors. [Reference to HOMEDATA in book removed from this web page, as they are (to the best of my knowledge) no longer involved in this effort.] It [THREADS] has been available since the summer of 1993, running on Intel-based machines. It has been tested on 268, 386 and 486-based machines. Since it is written in C, we are hoping that it will also be possible to port it later to other versions of C, although there is a small amount of environment-dependent code which will have to be modified by hand. This software is called THREADS - THREads-based Application Development System (I love self-referential names!). Like DFDM, it also has interpreted and compiled versions, so applications can be developed iteratively, and then compiled to produce a single EXE file, which eliminates the network decoding phase.

The terminology used in this book is not exactly the same as that used by AMPS and DFDM, as a number of these terms turned out to cause confusion. For instance, the data chunks that travel between the asynchronous processes were called "entities" in AMPS and DFDM, but, as I said above, this caused confusion for people experienced in data modelling. They do seem to correspond with the "entities" of data modelling, but "entities" have other connotations which could be misleading. "Objects" would present other problems, and we were not comfortable with the idea of creating totally new words (although some writers have used them effectively). The "tuples" of Carriero and Gelernter's Linda (1989) are very close, but this name also presents a slightly different image from the FBP concept. We therefore decided to use the rather neutral term "information packet" (or "IP" for short) for this concept. This term was coined as part of work that we did following the development of DFDM, in which we also tied FBP concepts in with other work appearing in the literature or being developed in other parts of IBM. Some of the extensions to the basic AMPS and DFDM substructure that I will be talking about later were also articulated during this period. When I need to refer to ideas drawn from this work I will use the name FPE (for Flow-Based Programming Environment), although that is not the acronym used by that project. THREADS follows this revised terminology, and includes a number of ideas from FPE.

As I stated in the prologue, for most of my 33 years in the computer business I have been almost exclusively involved with business applications. Although business applications are often more complex than scientific applications, the academic community generally has not shown much interest in this area up until now. This is a "catch 22" situation, as business would benefit from the work done in academia, yet academia (with some noteworthy exceptions) tends not to regard business programming as an interesting area to work in. My hope is that FBP can act as a bridge between these two worlds, and in later chapters I will be attempting to tie FBP to other related

theoretical work which working programmers probably wouldn't normally encounter. 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!). AMPS has been in use for 20 years, supporting one of the biggest companies in North America, and as recently as this year (1992), one of their senior people told me, "AMPS has served us well, and we expect it will continue to do so for a long time to come." Business systems have to evolve over time as the market requirements change, so clearly their system has been able to grow and adapt over the years as the need arose - this is a living system, not some outdated curiosity which has become obsolete with the advance of technology.

And now I would like to conclude this chapter with an unsolicited testimonial from a DFDM user, which we received a few years ago:

> "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."

Notice that 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, mentioned all the points that were significant to him as a developer - he 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! In DFDM, "coroutines" are the basic building blocks, which programmers can hook together to build applications. They are either already available ("on the shelf"), or the programmer can write new ones, in which case he or she will naturally try to reuse them as often as possible - to get the most bang for the proverbial buck. Although it is not very hard to write new PL/I coroutines, the majority of application developers don't want to write new code - they just want to get their applications working for the client, preferably using as little programming effort as will suffice to get a quality job done. Of course there are always programmers who love the process of programming and, as we shall see in the following pages, there is an important role

for them also in this new world which is evolving.

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. In later discussions with him, he has stressed the need to keep application structures simple. In FBP, you can use hierarchic decomposition to create multiple layers, each containing a simple structure, rather than being required to create a single, flat, highly complex structure. In fact, structures which are so complex that he would have trouble with them are difficult for everyone. He also points out that tools which he would find useful, such as something which can turn network diagrams into lists of connections, would also significantly assist normally sighted people as they work with these structures.

Rej's point about the advantages of keeping the structures simple is also borne out by the fact that another application of DFDM resulted in a structure of about 200 processes, but the programmer involved (another very bright individual) never drew a single picture! He built it up gradually using hierarchical decomposition, and it has since had one of the lowest error rates of any application in the shop. I hope that, as you read on, you will be able to figure out some of the reasons for this high level of reliability for yourself.

In what follows, I will be describing the main features of FBP and what we have learned from developing and using its various implementations. Information on some of these has appeared in a number of places and I feel it is time to try to pull together some of the results of 20 years of experience with these concepts into a single place, so that the ideas can be seen in context. A vast number of papers have appeared over the years, written by different writers in different fields of computing, which I believe are all various facets of a single diamond, but I hope that, by pulling a lot of connected ideas together in one place, the reader will start to get a glimpse of the total picture. Perhaps there is someone out there who is waiting for these ideas, and will be inspired to carry them further, either in research or in the marketplace!

# Chap. II: Higher-Level Languages, 4GLs and CASE

In the Prologue I alluded to the concept of compiler compilers. At the time I was most interested in them (the mid-60s), the accepted wisdom was that more sophisticated compilers were the answer to the productivity problem. It was clear to everyone that an expression like

```
W = (X + Y) / (Z + 3)
```

was infinitely superior to the machine language equivalent, which might look something like the following:

```
LOAD Z
ADD 3
STORE TEMP
LOAD X
ADD Y
DIV TEMP
STORE W
```

This is a made-up machine with a single accumulator, but you get the general idea. One of the reasons the syntax shown on the first line could effectively act as a bridge between humans and computers was that it was syntactically clean, and based on a solid, well-understood, mathematical foundation - namely arithmetic... with the exception of a rather strange use of the equals sign!

During this period it was not unreasonable to expect that this expressive power could be extended into other functions that machines needed to perform. This seemed to be supported by the fact that, although I/O was getting more complex to program at the machine language level, operating systems were coming on stream which still allowed the programmer to essentially write one

statement to execute a simple I/O operation. On the IBM 1401 a Read Card command consisted of one instruction and also one character! MVS's GET, on the other hand, might cause several hundreds or thousands of machine language instructions to be executed, but the programmer still basically wrote one statement.

On this foundation, we started getting one programming language after another: COBOL was going to be the language that enabled the person in the street, or at least managers of programmers, to do programming! Algol became a program-level documentation standard for algorithms. IBM developed PL/I (I worked on one rather short-lived version of that); people developed compilers in their basements; graduate students wrote compilers for their theses at universities (they still do). There was always the feeling that one of these languages was going to be the key to unlocking the productivity that we all felt was innate in programmers. While it is true that the science of compiler writing advanced by leaps and bounds, by and large programmer productivity (at least in business application development) did not go up, or if it did, it soon plateaued at a new level.

COBOL and PL/I were general-purpose compilers. There were also many languages specialized for certain jobs: simulation languages, pattern-matching languages, report generation languages. And let us not forget APL - APL is an extremely powerful language, and it also opened up arcane areas of mathematics like matrix handling for those of us who had never quite understood it in school. Being able to do a matrix multiply in 5 key-strokes **(A+.×B)** is still a level of expressiveness that I think few programming languages will ever be able to match! Its combination of sheer power in the mathematical area and the fact that there was no compile-time typing allowed one to get interesting programs up and running extremely fast. I once read a comment in a mathematical paper that the author didn't think the work would have been possible without APL - and I believe him. Although it was used a certain amount in business for calculation-oriented programming, especially in banking and insurance, and also as a base for some of the early query-type languages, APL did little for most commercial programming, plodding along using COBOL, and more recently PL/I, PASCAL, BASIC...

APL also illustrates in a different way the importance of minimizing the "gap" between idea and its expression - along with all of the most popular programming languages, it is an interpreter, which means that you can enter the program, and then immediately run it, without having to go through a compile and/or link step. Granted, this is more perception than actual fact (as one can build compile steps which are so fast that the user doesn't perceive them as a barrier), but the fact remains that some very awkward languages have become immensely popular because they did not require a compile step. A lot of the CPU cycles used in the industry on IBM machines are being used to run CMS EXECs or TSO CLISTs. Both of these are simple languages which let you stitch commands together into runnable "programs". Both are yielding nowadays to Mike Cowlishaw's REXX, which occupies the same niche, but also provides a vastly more powerful set of language constructs, to the point where one can build pretty complex programs with it. REXX

is also interpretive, so it also allows one to change a program and see the results of that change very quickly.

Why didn't languages (even the interpretive ones) improve productivity more than they did? I will be exploring this more in the next chapter, but one thing that I noticed fairly early on was that they didn't do much for logic (IF, THEN, ELSE, DO WHILE, etc.). For many kinds of business programming, what pushes up the development time is the logic - there actually may not be much in the way of straight calculations. A logical choice can be thought of as a certain amount of work, whether you write it like this:

```
IF x > 2
THEN
 result = 1
ELSE
 result = 2
ENDIF
```

or like this:

```
result = (x>2) ? 1 : 2;
```

or even draw it as a Nassi-Shneiderman or Chapin chart. One can argue that, because both of the above phrasings involve one binary decision, they involve approximately the same amount of mental work. The more complex the logic, the more difficult the coding. In fact, there is a complexity measure used quite widely in the industry called McCabe's Cyclomatic complexity measure, which is based very directly on the number of binary decisions in a program. However, in our work we have discovered that the amount of logic in conventional programming is reducible, because much of the logic in conventional programming has to do with the synchronization of data, rather than with business logic. Since FBP eliminates the need for a lot of this synchronization logic, this means that FBP actually does reduce the amount of logic in programs.

A number of writers have made the point that productivity is only improved if you can reduce the number of statements needed to represent an idea. Put another way, you have to reduce the "gap" between the language of business and the language of computers. What is the lower limit on the number of bits or keystrokes to represent an idea, though? If a condition and its branches comprise one "idea", then there is a lower limit to how compactly it can be represented. If it is part of a greater "idea", then there is a hope of representing it more compactly. From Information

Theory we learn that the number of bits needed to represent something is the log of the number of alternatives you have to choose between. If something is always true, there is no choice, so it doesn't need any bits to represent it. If you have 8 choices, then it takes 3 (i.e. log of 8 to the base 2) bits to represent it. In programming terms: if you only allow an individual two marital states, you only need 1 bit (log of 2 to the base 2). If you want to support a wider "universe", where people may be single, married, divorced, widowed, or common law spouses, that is five alternatives, so you need 3 bits (2 bits are not enough as they only allow 4 choices). And they're not mutually exclusive, so you could need even more bits!

This in turn leads to our next idea: one way to reduce the information requirements of a program is to pick options from a more limited universe. However, the user has to be willing to live within this more circumscribed universe. I remember an accounting package that was developed in Western Canada, which defined very tightly the universe in which its customers were supposed to operate. Within that universe, it provided quite a lot of function. I believe that its main file records were always 139 bytes long (or a similar figure), and you couldn't change them. If you asked them about it, the developers' reaction would be: why would anyone want to? Somewhat predictably, it didn't catch on because many customers felt it was too limited. The example that sticks in my memory was that of one customer who wanted to change a report title and had to be told it couldn't be done. Again, why would anyone feel that was so important? Well, it seems that customers, especially big ones, tend to feel that report headers should look the way they want, rather than the way the package says they are going to look. Our experience was that smaller customers might be willing to adapt their business to the program, especially if you could convince them that you understood it better than they did, but bigger customers expected the program to adapt to their business. And it was really quite a powerful package for the price! I learned a lot from that, and the main thing was that a vendor can provide standard components, but the customer has to be able to write custom components as well. Even if it costs more to do the latter, it is the customer's choice, not the vendor's. And that of course means that the customer must be able to visualize what the tool is doing. This is also related to the principle of Open Architecture: no matter how impressive a tool is, if it can't talk to other tools, it isn't going to survive over the long haul (paraphrased from Wayne Stevens).

The above information-theoretic concept is at the root of what are now called 4GLs (4th Generation Languages). These provide more productivity by taking advantage of frequently appearing application patterns, e.g. interactive applications. If you are writing applications to run in an interactive system, you know that you are going to keep running into patterns like:

- read a key entered by the user onto a screen
- get the correct record, or generate a message if the record does not exist
- display selected fields from that record on the screen.

Another one (very often the next one) might be:

- display the fields of a record
- determine which ones were changed by the user
- check for incorrect formats, values, etc.
- if everything is OK,
  write the updated record back
- else
  display the appropriate error message(s)

Another very common pattern (especially in what is called "decision support" or "decision assist" type applications) occurs when a list is presented on the screen and the user can select one of the items or scroll up or down through the list (the list may not all fit on one screen). Some systems allow more than one item to be selected, which are then processed in sequence.

These recurrent patterns occur so frequently that it makes a lot of sense to provide skeletons for these different scenarios, and declarative (non-procedural) ways of having the programmer fill in the information required to flesh them out.

The attractiveness of 4GLs has also been enhanced by the unattractiveness of IBM's standard screen definition facilities! The screen definition languages for both IMS and CICS are coded up using S/370 Assembler macros (high-level statements which generate the constants which define all the parts of a screen). This technique allows them to provide a lot of useful capabilities, but screen definitions written this way are hard to write and even harder to maintain! Say you want to make a field longer and move it down a few lines, you find yourself changing a large number of different values which all have to be kept consistent (the values are often not even the same, but have to be kept consistent according to some formula). I once wrote a prototyping tool which allowed screens to be specified in WYSIWYG (What You See Is What You Get) format, and could then be used to generate both the screen definition macros and also all the HLL declares that had to correspond to it. It was quite widely used internally within IBM, and in fact one project, which needed to change some MFS, started out by converting the old MFS into the prototyper specifications, so that they could make their changes, and then generate everything automatically. This way, they could be sure that everything stayed consistent. When such a screen definition tool is integrated with a 4GL, you get a very attractive combination. It's even better when the prototyping tool is built using FBP, as it can then be "grown" into a full interactive application by incrementally expanding the network. This ability to grow an application from a prototype seems very desirable, and is one of the things that make FBP attractive for developing interactive applications.

The problem, of course, with the conventional 4GL comes in when the customer, like our customer above who wanted to change a report title, wants something that the 4GL does not provide. Usually this kind of thing is handled by means of exits. A system which started out simple eventually becomes studded with exits, which require complex parametrization, and whose function cannot be understood without understanding the internals of the product - the

flow of the product. Since part of the effectiveness of a 4GL comes from its being relatively opaque and "black boxy", exits undermine its very reason for being.

An example of this in the small is IBM's MVS Sort utility (or other Sorts which are compatible with it) - as long as one can work with the standard parameters for the Sort as a whole, it's pretty clear what it is doing, and how to talk to it. Now you decide you want to do some processing on each input record before it goes into the Sort. You now have to start working with the E15 exit. This requires that you form a concept of how Sort works on the inside - a very different matter. E15 and E35 (the output exit routine) have to be independent, non-reentrant, load modules, so this puts significant constraints on the ways in which applications can use load module libraries... and so on. Luckily Sort also has a LINKable interface, so DFDM [and AMPS before it] used this, turned E15 and E35 inside-out, and converted the whole thing into a well-behaved reusable component. Much easier to use and you get improved performance as well due to the reduction in I/O! In a similar sort of way, FBP can also capitalize on the same regularities as 4GLs do by providing reusable components (composite or elementary) as well as standard network shapes. Instead of programmers having to understand the internal logic of a 4GL, they can be provided with a network and specifications of the data requirements of the various components. Instead of having to change mental models to understand and use exits, the programmer has a single model based on data and its transformations, and is free to rearrange the network, replace components by custom ones, or make other desired changes.

I should point out also that the regularities referred to above have also provided a fertile breeding-ground for various source code reuse schemes. My feeling about source code reuse is that it suffers from a fundamental flaw: even if building a new program can be made relatively fast, once you have built it, it must be added to the ever-growing list of programs you have to maintain. It is even worse if, as is often required, the reusable source code components have to be modified before your program can work, because then you have lost the trail connecting the original pieces of code to your final program if one of them has to be enhanced, e.g. to fix a bug. Even if no modification takes place, the new program has to be added to the list of program assets your installation owns. Already in some shops, maintenance is taking 80% of the programming resource, so each additional application adds to this burden. In FBP, ideally all that is added to the asset base is a network - the components are all black boxes, and so a new application costs a lot less to maintain.

A related type of tool are program generators - this is also source-level reuse with a slightly different emphasis. As above, an important question is whether you can modify the generated code. If you can't, you are limited to the choices built into the generator; if you can, your original source material becomes useless from a maintenance point of view, and can only be regarded as a high-level (and perhaps even misleading) specification. Like out of date documentation, it might almost be safer to throw it away...

I don't want to leave this general area without talking about CASE (Computer-Aided Software

Engineering) tools. The popularity of these tools arises from several very valid concepts. First, people should not have to enter the same information multiple times - especially when the different forms of this data are clearly related, but you have to be a computer to figure out how! We saw this in the case of the prototyping tool mentioned above. If you view the development process as a process of expressing creativity in progressive stages within the context of application knowledge, then you want to capture this in a machine, and not just as text, but in a form which captures meaning, so that it can be converted to the various formats required by other software, on demand. This information can then be converted to other forms, added to, presented in a variety of display formats, etc.

There are a number of such tools out in the marketplace today, addressing different parts of the development process, and I see these as the forerunners of the more sophisticated tools which will become available in the next few years. Graphical tools are now becoming economical, and I believe that graphical techniques are the right direction, as they take advantage of human visualization skills. I happen to believe HIPOs (remember them? Hierarchical Input, Process, Output) had part of the right answer, but adequate graphical tools were not available in those days, and maintaining them with pencil, eraser and a template was a pain! However, when someone went through all that trouble, and produced a finished HIPO diagram, the results were beautiful and easy to understand. Unfortunately, systems don't stand still and they were very hard to maintain, given the technology of those days!

Our experience is that Structured Analysis is a very natural first stage for FBP development, so CASE tools which support Structured Analysis diagrams and which have open architectures are natural partners with FBP. In fact, FBP is the only approach which lets you carry a Structured Analysis design all the way to execution - with conventional programming, you cannot convert the design into an executable program structure. There is a chasm, which nobody has been able to bridge in practice, although there are some theoretical approaches, such as the Jackson Inversion, which have been partially successful. In FBP, you can just keep adding information to a design which uses the Structured Analysis approach until you have a working program. In what follows, you will see that FBP diagrams do not really require much information at the network level to create a running program which is not already captured by the Structured Analysis design. Probably the most important point is that one has to distinguish between code components and processes (occurrences of components), and some Structured Analysis tools do not make a clear distinction between these two concepts. As we shall see in the following chapter, an FBP network consists of multiple communicating processes, but a tool which is viewed as primarily being for diagramming may be forgiven for assuming that all the blocks in a diagram are unique, different programs. The need to *execute* a picture imposes a discipline on the design process and the designer, which means that these confusions have to be resolved. We actually developed some PC code to convert a diagram on one of the popular CASE tools into a DFDM network specification, which was used successfully for several projects.

FBP's orientation towards reuse also forces one to distinguish between a particular use of a component and its general definition. This may seem obvious in hindsight, but, even when documenting conventional programs, you would be amazed how often programmers give a fine generalized description of, say, a date routine, but forget to tell the reader which of its functions is being used in a particular situation. Even in a block diagram I find that programmers often write in the general description of the routine and omit its specific use (you need both). This is probably due to the fact that, in conventional programming, the developer of a routine is usually its only user as well, so s/he forgets to "change hats". When the developer and user are different people, it is easier for the user to stay in character.

To summarize, HLLs, 4GLs and CASE are all steps along a route towards the place we want to be, and all have lessons to teach us, and capabilities which are definitely part of the answer. What is so exciting about FBP is that it allows one to take the best qualities of all the approaches which went before, and combine them into a larger whole which is greater than the sum of its parts.

# Chap. III: Basic Concepts

In the middle years of this century, it was expected that eventually computers would be able to do anything that humans could explain to them. But even then, some people wondered how easy this would turn out to be. A book first published in the 1950's, *"Faster than Thought"* (Bowden 1963), contains the following remark:

> "It always outrages pure mathematicians when they encounter commercial work for the first time to find how difficult it is." (p. 258)

and further on:

> "...it is so hard in practice to get any programme right that several mathematicians may be needed to look after a big machine in an office. ... It seems, in fact, that experienced programmers will always be in demand" (p. 259)

However, in those days, the main barrier was thought to be the difficulty of ascertaining the exact rules for running a business. This did turn out to be as hard as people expected, but for rather different reasons. Today we see that the problem is actually built right into the fundamental design principles of our basic computing engine - the so-called von Neumann machine we alluded to in Chapter 1.

The von Neumann machine is perfectly adapted to the kind of mathematical or algorithmic needs for which it was developed: tide tables, ballistics calculations, etc., but business applications are rather different in nature. As one example of these differences, the basic building block of programming is the subroutine, and has been since it was first described by Ada, Countess Lovelace, Lord Byron's daughter, in the last century [of course that is the 19th!] (quite an achievement, since computers did not exist yet!). This concept was solidly founded on the mathematical idea of a function, and any programmer today knows about a number of standard

subroutines, and can usually come up with new ones as needed. Examples might include "square root", "binary search", "sine", "display a currency value in human-readable format", etc. What are the basic building blocks of business applications? It is easy to list functions such as "merge", "sort", "paginate a report", and so forth, but none of these seem to lend themselves to being encapsulated as subroutines. They are of a different nature - rather than being functions that operate at a single moment in time, they all have the characteristic of operating over a period of time, across a number (sometimes a very large number) of input and output items.

We now begin to get some inkling of what this difference might consist of. Business programming works with data and concentrates on how this data is transformed, combined and separated, to produce the desired outputs and modify stored data according to business requirements. Broadly speaking, whereas the conventional approaches to programming (referred to as "control flow") start with process and view data as secondary, business applications are usually designed starting with data and viewing process as secondary - processes are just the way data is created, manipulated and destroyed. We often call this approach "data flow", and it is a key concept of many of our design methodologies. It is when we try to convert this view into procedural code that we start to run into problems.

I am now going to approach this difference from a different angle. Let's compare a pure algorithmic problem with a superficially similar business-related one. We will start with a simple numeric algorithm: calculating a number raised to the 'n'th power, where 'n' is a positive, integer exponent. This should be very familiar to our readers, and in fact is not the best way to do this calculation, but it will suffice to make my point. In pseudocode, we can express this as follows:

```
/* Calculate a to the power b  */
        x = b
        y = 1
        do while x > 0
            y = y * a
            x = x - 1
        enddo
        return y
```

Figure 3.1

This is a very simple algorithm - easy to understand, easy to verify, easy to modify, based on well-understood mathematical concepts. Let us now look at what is superficially a similar piece of logic, but this time involving files of records, rather than integers. It has the same kind of structure as the preceding algorithm, but works with streams of records. The problem is to create a file OUT which is a subset of another one IN, where the records to be output are those which satisfy a given criterion "c". Records which do not satisfy "c" are to be omitted from the output file. This is a pretty common requirement and is usually coded using some form of the following logic:

```
read into a from IN
do while read has not reached end of file
    if c is true
        write from a to OUT
    endif
    read into a from IN
enddo
```

Figure 3.2

What action is applied to those records which do not satisfy our criterion? Well, they disappear rather mysteriously due to the fact that they are not written to OUT before being destroyed by the next "read". Most programmers reading this probably won't see anything strange in this code, but, if you think about it, doesn't it seem rather odd that it should be possible to drop important things like records from an output file by means of what is really a quirk of timing?

Part of the reason for this is that most of today's computers have a uniform array of pigeon-holes for storage, and this storage behaves very differently from the way storage systems behave in real life. In real life, paper put into a drawer remains there until deliberately removed. It also takes up space, so that the drawer will eventually fill up, preventing more paper from being added. Compare this with the computer concept of storage - you can reach into a storage slot any number of times and get the same data each time (without being told that you have done it already), or you can put a piece of data in on top of a previous one, and the earlier one just disappears.... Although destructive storage is not integral to the the von Neumann machine, it is assumed in many functions of the machine, and this is the kind of storage which is provided on most modern computers. Since the storage of these machines is so sensitive to timing and because the sequencing of every instruction has to be predefined (and humans make mistakes!), it is incredibly difficult to get a program above a certain complexity to work properly. And of course this storage paradigm has been enshrined in most of our higher level languages in the concept of a "variable". In a celebrated article John Backus (1978) actually apologized for inventing FORTRAN! That's what I meant earlier about the strange use of the equals sign in Higher Level Languages. To a logician the statement J = J + 1 is a contradiction (unless J is infinity?) - yet programmers no longer notice anything strange about it!

Suppose we decide instead that a record should be treated as a real thing, like a memo or a letter, which, once created, exists for a definite period of time, and must be explicitly destroyed before it can leave the system. We could expand our pseudo-language very easily to include this concept by adding a "discard" statement (of course the record has to be identified somehow). Our program might now read as follows:

```
read record a from IN
do while read has not reached end of file
    if c is true
        write a to OUT
    else
        discard a
    endif
    read record a from IN
enddo
```

Figure 3.3

Now we can reinterpret "a": rather than thinking of it as an area of storage, let us think of "a" as a "handle" which designates a particular "thing" - it is a way of locating a thing, rather than the storage area containing the thing. In fact, these data things should not be thought of as being in storage: they are "somewhere else" (after all, it does not matter where "read" puts them, so long as the information they contain becomes available to the program). These things really have more attributes than just their images in storage. The storage image can be thought of as rather like the projection of a solid object onto a plane - manipulating the image does not affect the real thing behind the image. Now a consequence of this is that, if we reuse a handle, we will lose access to the thing it is the handle of. This therefore means that we have a responsibility to properly dispose of things before we can reuse their handles.

Notice the difficulty we have finding a good word for "thing": the problem is that this is really a concept which is "atomic", in the sense that it cannot be decomposed into yet more fundamental objects. It has had a number of names in the various dialects of FBP, and has some affinities with the concept of "object" in Object-Oriented Programming, but I feel it is better to give it its own unique name. In what follows, we will be using the term "information packet" (or "IP"). IPs may vary in length from 0 bytes to 2 billion - the advantage of working with "handles" is that IPs are managed the same way, and cost the same to send and receive, independently of their size.

So far, we have really only added one concept - that of IPs - to the conceptual model we are building. The pseudo-code in Figure 3.3 was a main-line program, running alone. Since this main-line can call subroutines, which in turn can call other subroutines, we have essentially the same structure as a conventional program, with one main line and subroutines hung off it, and so on. Now, instead of just making a single program more complex, as is done in conventional programming, let us head off in a rather different direction: visualize an application built up of many such main-line programs running concurrently, passing IPs around between them. This is very like a factory with many machines all running at the same time, connected by conveyor belts. Things being worked on (cars, ingots, radios, bottles) travel over the conveyor belts from one machine to another. In fact, there are many analogies we might use: cafeterias, offices with

memos flowing between them, people at a cocktail party, and so forth. After I had been working with these concepts for several years, I took my children to look at a soft-drink bottling plant. We saw machines for filling the bottles, machines for putting caps on them and machines for sticking on labels, but it is the connectivity and the flow between these various machines that ensures that what you buy in the store is filled with the right stuff and hasn't all leaked out before you purchase it!

An FBP application may thus be thought of as a "data factory". The purpose of any application is to take data and process it, much as an ingot is transformed into a finished metal part. In the old days, we thought that it would be possible to design software factories, but now we see that this was the wrong image: we don't want to mass-produce code - we want less code, rather than more. In hindsight it is obvious - it is the data which has to be converted into useful information in a factory, not the programs.

Now think of the differences between the characteristics of such a factory and those of a conventional, single main-line program. In any factory, many processes are going on at the same time, and synchronization is only necessary at the level of an individual work item. In conventional programming, we have to know exactly when events take place, otherwise things are not going to work right. This is largely because of the way the storage of today's computers works - if data is not processed in exactly the right sequence, we will get wrong results, and we may not even be aware that it has happened! There is no flexibility or adaptability. In our factory image, on the other hand, we don't really care if one machine runs before or after another, as long as processes are applied to a given work item in the right order. For instance, a bottle must be filled before it is capped, but this does not mean that all the bottles must be filled before any of them can be capped. It turns out that conventional programs are full of this kind of unnecessary synchronization, which reduces productivity, puts unnecessary strains on the programmers and generally makes application maintenance somewhere between difficult and impossible. In a real factory, unnecessary constraints of this sort would mean that some machines would not be utilized efficiently. In programming, it means that code steps have to be forced into a single sequence which is extremely difficult for humans to visualize correctly, because of a mistaken belief that the machine requires it. It doesn't!

An application can alternatively be expressed as a network of simple programs, with data travelling between them, and in fact we find that this takes advantage of developers' visual or "spatial" imagination. It is also a good match with the design methodologies generally referred to under the umbrella of Structured Analysis. The so-called "Jackson inversion model" (M. Jackson 1975) designs applications in this form, but then proposes that all the processes except one (the main-line) be "inverted" into subroutines. This is no longer necessary! Interestingly, Jackson starts off his discussion of program inversion with a description of a simple multiprogramming scheme, using a connection with a capacity of one (in our terminology). He then goes on to say, "Multi-programming is expensive. Unless we have an unusually favourable environment we will

not wish to use the sledgehammer of multi-programming to crack the nut of a small structure clash." In FBP we have that "unusually favourable environment", and I and a number of other people believe he wrote off multi-programming much too fast!

How do we get multiple "machines" to share a real machine? That's something we have known how to do for years - we just give them slices of machine time whenever they have something to do - in other words, let them "time-share". In conventional programming, time-sharing was mostly used to utilize the hardware efficiently, but it turns out that it can also be applied to convert our "multiple main-line" model into a working application. There are a great many possible time-sharing techniques, but in FBP we have found one of the simplest works fine: we simply let one of our "machines" (called "processes") run until an FBP service request cannot be satisfied. When this happens, the process is suspended and some other ready process gets control. Eventually, the suspended process can proceed (because the blocking condition has been relieved) and will get control when time becomes available. Dijkstra called such processes "sequential processes", and explains that the trick is that they do not know that they have been suspended. Their logic is unchanged - they are just "stretched" in time.

Now you may have guessed that the FBP service requests we referred to above have to do with communication between processes. Processes are connected by means of FIFO (first-in, first-out) queues or connections which can hold up to some maximum number of IPs (called a queue's "capacity"). For a given queue, this capacity may run from one to quite large numbers. Connections use slightly different verbs from files, so we will convert the pseudocode in the previous example, replacing:

- "read" with "receive"
- "write" with "send"
- "discard" with "drop"
- "end of file" with "end of data"

A "receive" service may get blocked because there is no data currently in the connection, and a "send" may be blocked because the connection is full and cannot accept any more data for a while. Think of a conveyor belt with room for just so many bottles, televisions, or whatever: it can be empty, full or some state in between. All these situations are normally just temporary, of course, and will change over time. We have to give connections a maximum capacity, not only so that our applications will fit into available storage, but also so that all data will eventually be processed (otherwise data could just accumulate in connections and never get processed).

Now processes cannot name connections directly - they refer to them by naming ports, the points where processes and connections meet. More about ports later.

The previous pseudocode now looks like this:

```
receive from IN using a
do while receive has not reached end of data
    if c is true
        send a to OUT
    else
        drop a
    endif
    receive from IN using a
enddo
```

Figure 3.4

I deliberately used the word "using" on the "receive" to stress the nature of "a" as a handle, but "send a" seems more natural than "send using a". Note the differences between our file processing program and our FBP component:

- differences in the verbs (already mentioned)
- IPs "out there", rather than records in storage
- IPs must be positively disposed of
- port names instead of file names.

We said earlier that IPs are things which have to be explicitly destroyed - in our multi-process implementation, we require that all IPs be accounted for: any process which receives an IP has its "number of owned IPs" incremented, and must reduce this number back to zero before it exits. It can do this in essentially the same ways we dispose of a memo: destroy it, pass it on or file it. Of course, you can't dispose of an IP (or even access it) if you don't have its handle (or if its handle has been reused to designate another IP). Just like a memo, an IP cannot be reaccessed by a process once it has been disposed of (in most FBP implementations we zero out the handle after disposing of the IP to prevent exactly that from happening).

To get philosophical for a moment, proponents of "garbage collection" tend to feel that IPs should disappear when nobody is looking at them (no handles reference them), but the majority of the developers of FBP implementations felt that that was exactly what was wrong with conventional programming: things could disappear much too easily. So we insisted that a process get rid of all of its IPs, explicitly, one way or another, before it gives up control. If you inject an IP in at one end of the network, you know it is going to get processed unless or until some process explicitly destroys it! Conversely, if you build a table IP, and forget to drop it when you are finished, many people might argue that it would be nice to have the system dispose of it for you. On the other hand... I could argue that such an error (if it is an error) may be a sign of something more fundamentally wrong with the design or the code. And anyway, all recent FBP implementations detect this error, and list the IPs not disposed of, so it is easy to figure out what you did wrong. So..., if we can do that, why not let garbage collection be automatic? Well, our

team took a vote, and strict accounting won by a solid majority! It might have lost if the group had had a different composition! Eventually, we could make this both an environmental decision and an attribute of each component, so we could detect if a "loose" component was being run in a "tight" shop.

Now what are "IN" and "OUT" that our pseudocode receives IPs from and sends them to? They are not the names of connections, as this would tie a component's code to one place in the network, but are things called "ports". "Portae" means "gates" in Latin, and ports (like seaports) can be thought of as specific places in a wall or boundary where things or people go in or out. Weinberg (1975) describes a port as

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

Now doors have an "inside" aspect and an "outside" aspect - the name of the inside aspect might be used by people inside the house to refer to doors, e.g. "let the cat out the side door", while the outside aspect is related to what the door opens onto, and will be of interest to city planners or visitors. Ports in FBP have the same sort of dual function: they allow an FBP component to refer to them without needing to be aware of what they open onto. Port names establish a relationship between the receives and sends inside the program and program structure information defined outside the component. This somewhat resembles subroutine parameters, where the inside (parameters) and the outside (arguments) have to correspond, even though they are compiled separately. In the case of parameters, this correspondence is established by means of position (sequence number). In fact, in AMPS and DFDM ports were identified by numbers rather than by names. While this convention gives improved performance by allowing ports to be located faster, our experience is that users generally find names easier to relate to than numbers - just as we say "back door" and "front door", rather than "door 1", "door 3", etc. For this reason, THREADS uses port names, not numbers.

Some ports can be defined to be arrays, so that they are referenced by an index as well as a name. Thus, instead of sending to a single OUT port, some components can have a variable number of OUT ports, and can therefore say "send this IP to the first (or n'th) OUT port". This can be very useful for components which, say, make multiple copies of a given set of information. The individual slots of an array-type port are called "port elements". Naturally you can also have array-type ports as input ports, which might for instance be used in components which do various kinds of merge operation. Finally, a method is provided for the component to find out how many elements of an array-type port are connected, and which ones. In DFDM we did not need array-type ports as ports were numbered, so you could define, say, ports 3 to 20 as acting as a single array.

Now let's draw a picture of the component we built above. It's called a "filter" and looks like this:

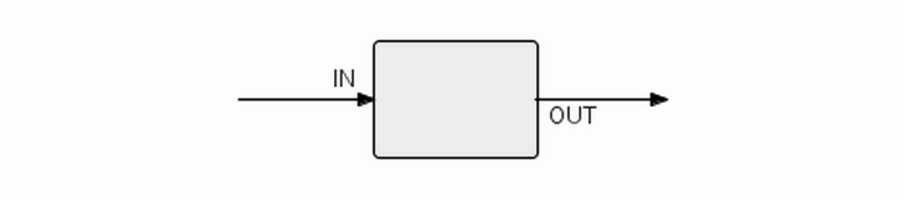



Figure 3.5

This component type has a characteristic "shape", which we will encounter frequently in what follows. FBP is a graphical style of programming, and its usability is much enhanced by (although it does not require) a good picture-drawing tool. Pictures are an international language and we have found that FBP provides an excellent medium of communication between application designers and developers and the other people in an organization who need to be involved in the development process.

Now we will draw a different shape of component, called a "selector". Its function is to apply some criterion "c" to all incoming IPs (they are received at port IN), and send out the ones that match the specified criterion to one output port (ACC), while sending the rejected ones to the other output port (REJ). Here is the corresponding picture:

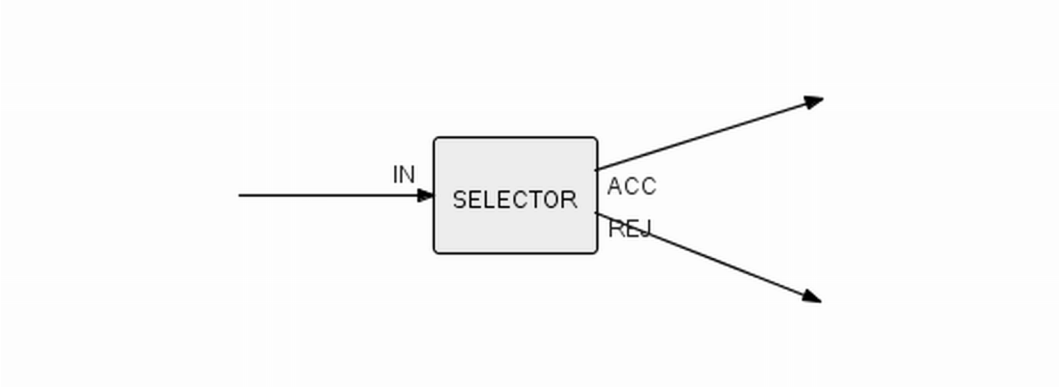



Figure 3.6

You will probably have figured out the logic for this component yourself. In any case, here it is:

```
receive from IN using a
 do while receive has not reached end of data
     if c is true
         send a to ACC
     else
         send a to REJ
     endif
     receive from IN using a
 enddo
```

Figure 3.7

Writing components is very much like writing simple main-line programs. Things really get interesting when we decide to put them together. Suppose we want to apply the filter to some data and then apply the selector to the output of the filter: all we need to record is that, for this application, OUT of FILTER is connected to IN of SELECTOR. You will notice that IN is used as a port name by both FILTER and SELECTOR, but this is not a problem, as port names only have to be unique within a given component, not across an entire application.

Let us draw this schematically:

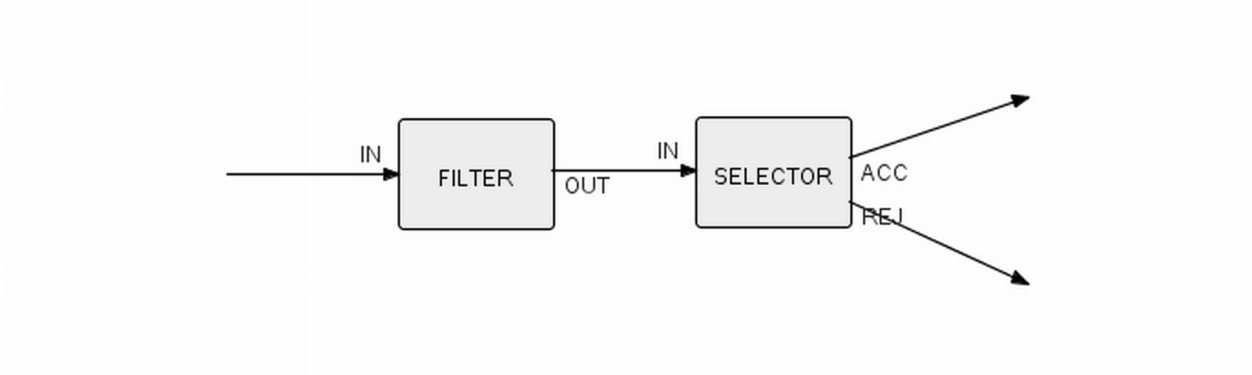



Figure 3.8

We have just drawn our first (partial) FBP structure! FBP software can execute this kind of diagram directly (without our having to convert it to procedural code), and in fact you can reconfigure it in any way you like - add components, remove them, rearrange them, etc., endlessly. This is the famous "Legoland programming" which we have all been waiting for!

Now what is the line marked "C" in the diagram? It is the connection across which IPs will travel when passing from FILTER to SELECTOR. It may be thought of as a pipe which can hold up to some maximum number of IPs (its "capacity"). So to define this structure we have to record the fact that OUT of FILTER is connected to IN of SELECTOR by means of a connection with

capacity of "n".

Now how do we prove to our satisfaction that this connection is processing our data correctly? Well, there are two constraints that apply to IPs passing between any two processes. If we use the names in the above example, then:

- every IP must arrive at SELECTOR after it leaves FILTER
- any pair of IPs leaving FILTER in a given sequence must arrive at SELECTOR in the same sequence

The first constraint is called the "flow" constraint, while the second is called the "order-preserving" constraint. If you think about it using a factory analogy, this is all you need to ensure correct processing. Suppose two processes A and B respectively generate and consume two IPs, X and Y. A will send X, then send Y; B will receive X, then receive Y (order-preserving constraint). Also, B must receive X after A sends it, and similarly for Y (flow constraint). This does not mean that B cannot issue its "receives" earlier - it will just be suspended until the data arrives. It also does not matter whether A sends Y out before or after B receives X. The system is perfectly free to do whatever results in the best performance. We can show this schematically - clearly the second diagonal line can slide forward or back in time without affecting the final result.

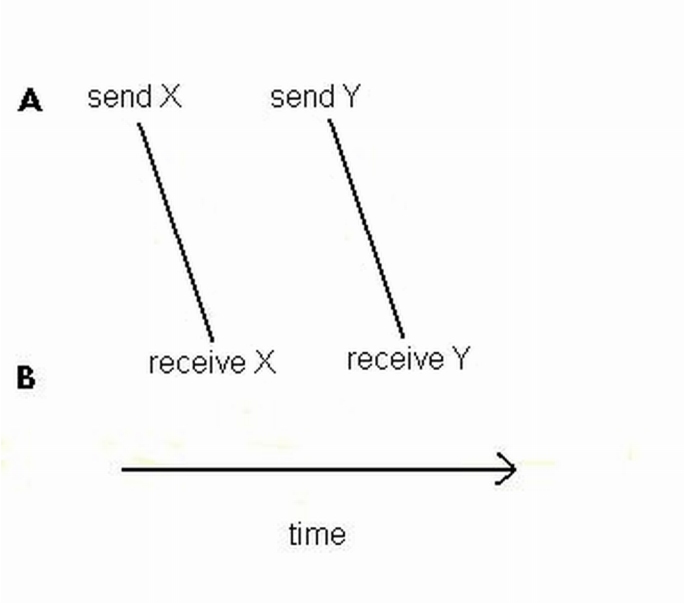



Figure 3.9

Connections may have more than one input end, but they may only have one output. IPs from

multiple sources will merge on a connection, arriving at the other end in "first come, first served" sequence. It can be argued that by allowing this, we lose the predictability of the relationship between output and input, but it is easy enough to put a code in the IPs if you ever want to separate them again.

Up to now, we have been ignoring where IPs come from originally. We have talked about receiving them, sending them and dropping them, but presumably they must have originally come into existence at some point in time. This very essential function is called "create" and is the responsibility of whichever component first decides that an IP is needed. The "lifetime" of an IP is the interval between the time it is created and the time it is destroyed. From one point of view it is obvious that something needs to be created before it can be used, but how does this apply to a business application? Well, a lot of the IPs in an application are created from file records: generally file records are turned into IPs at file reading time, and the IPs are turned back into file records (and then destroyed) at file writing time. There are two standard components to do these functions (Read Sequential and Write Sequential), which we will talk more about in succeeding chapters. However, it often happens that you decide to create a brand new IP for a particular purpose which may never appear on a file - one example of these are the "control IPs" which we will describe in a later chapter. Another example might be a counting component which counts the IPs it receives, using a "counter" IP. This IP is used to maintain the count, and is finally sent to the output port when the counter terminates. Such a component will receive the IPs being counted, but, before it starts counting, it has to create a counter IP in which the count is to be maintained.

This is the typical logic of a Counter component (by the way, this kind of component normally tries to send incoming IPs to an output port, and drops them if this port is not connected):

```
create counter IP using c
zero out counter field in counter IP
receive from IN using a
do while receive has not reached end of data
    increment count in counter IP
    send a to OUT
    if send wasn't successful,
        drop a
    endif
    receive from IN using a
enddo
send c to COUNT port
```

Figure 3.10

To discover whether OUT is connected, we simply try to send to this port. If the send works, the IP is disposed of; if not, we still have it, so we dispose of it by dropping it. What if COUNT is

not connected? Since the whole point of this component is to calculate a count, if COUNT is not connected there's not much point in even running this component, so it would be even better to test this right up front.

As we said above, all IPs must eventually be disposed of, and this will be done by some function which knows that the IP in question is no longer needed. This will often be Writer components, but not necessarily. The Selector example above should really be enhanced as follows:

```
receive from IN using a
do while receive has not reached end of data
    if c is true
        send a to ACC
    else
        send a to REJ
    endif
    if the send wasn't successful,
        drop a
    endif
    receive from IN using a
enddo
```

Figure 3.11

At the beginning of this chapter, we talked about data as being primary. In FBP it is not file records which are primary, but the IPs passing through the application's processes and connections. Our experience is that this is almost the first thing an FBP designer must think about. Once you start by designing your IPs first, you realize that file records are only one of the ways a particular process may decide to store IPs. The other thing file records are used for is to act as interfaces with other systems, but even here they still have to be converted to IPs before another process can handle them.

Let's think about the IPs passing across any given connection - what is their layout? Clearly it must be something that the upstream process can output and the downstream process can handle. The FBP methodology requires the designer to lay out the IPs first, and then define the transforms which apply to them. If two connected components have different requirements for their data, it is simple to insert a "transform" component between them. The general rule is that two neighbours must either agree on the format of data they share, or agree on data descriptions which encode the data format in some way. Suppose, for instance, that process B can handle two formats of IP. If the application designer knows that process A is always going to generate the first format, s/he may parametrize B so that it knows what to expect. If A is going to generate an unpredictable mix of the two formats, it will have to indicate to B for each IP what format it is in, e.g. by an agreed-upon code in a field, by IP length, by a preceding IP, or whatever. An interesting variant of this is to use free-form data. There may be situations where you don't want

to tie the format of IPs down too tightly, e.g. when communicating between subsystems which are both undergoing change. To separate the various fields or sections, you could imbed delimiters into the data. You will pay more in CPU time, but this may well be worth it if it will reduce your maintenance costs. This is why, for instance, communication formats between PCs and hosts often use free-form ASCII separated by delimiters (binary fields present unique problems when uploading and downloading data). Lastly, the more complete FBP implementations provide mechanisms for attaching standard descriptions to IPs, called Descriptors, allowing them to be used and reused in more and more applications. Descriptors allow individual fields in IPs to be retrieved or replaced by name - I will be describing them in more detail in a later chapter.

The next concept I want to describe is the ability to group components into packages which can be used as if they were components in their own right. This kind of component is called a "composite component". It is built up out of lower-level components, but on the outside it looks just like any other component. Components which are not built up of lower-level components are called "elementary", and are usually written in some Higher Level Language (DFDM supports both PL/I and VS COBOL II), while THREADS supports C. DFDM as distributed also includes a small set of "starter set" components written in a low-level programming language for performance reasons, but it is not expected that the majority of users will need to code at this level.

To make a composite component look like other components from the outside, obviously it must have ports of its own. We will therefore take the previous diagram, and show how we can package it into a composite called COMPA:

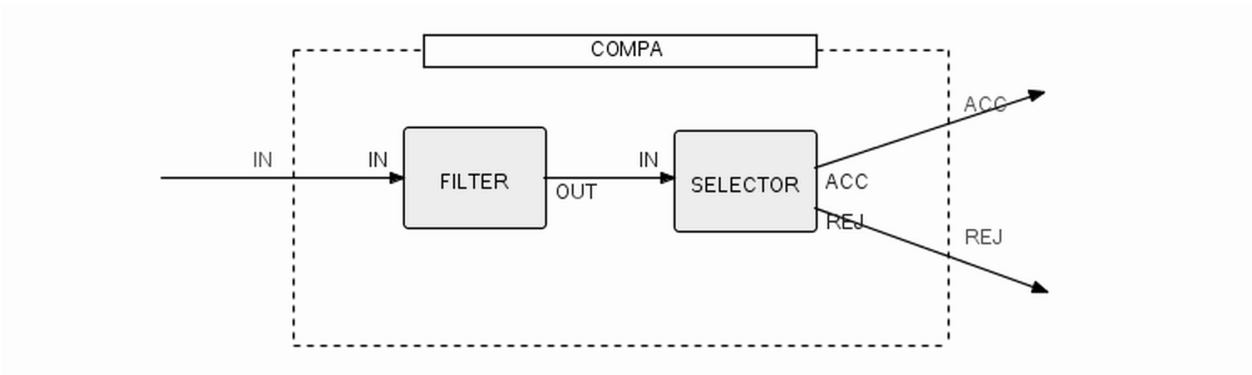



Figure 3.12

Once we have done this, COMPA can be used by anyone who knows what formats of data may be presented at port IN of COMPA and what formats will be sent out of its ACC and REJ ports. You will notice that it is quite acceptable for our composite to have the same port names as one

of its internal components. You might also decide to connect the ACC port inside to the REJ port outside, and vice versa - what you would then have is a Rejector composite process, rather than an Acceptor. Of course COMPA is not a very informative name, and in fact we probably wouldn't bother to make this function a composite unless we considered it a useful tool which we expected to be able to reuse in the future.

Notice also that we have shown the insides of COMPA - from the outside it looks like a regular component with one input port and two output ports, as follows:

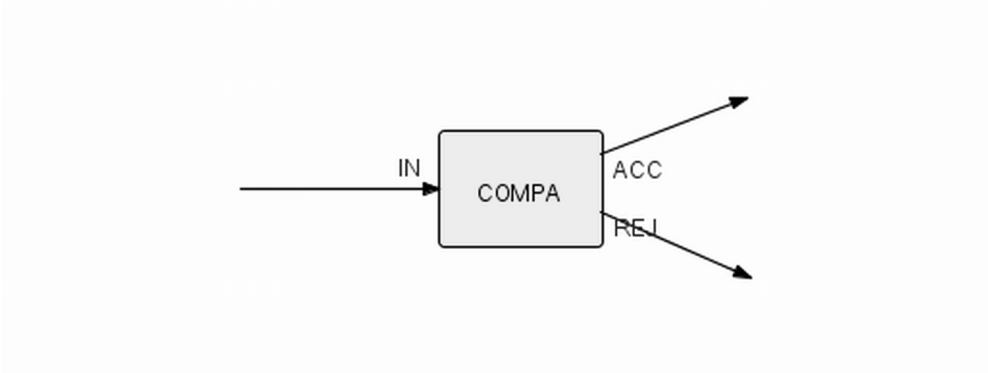



Figure 3.13

Now, clearly, any port on a composite must have corresponding ports "on the inside". However, the inverse is not required - not all ports on the inside have to be connected on the outside - if an inside component tries to send to an unconnected composite port, it will get a return of "unconnected" or "closed", depending on the implementation.

We have now introduced informally the ideas of "port", "connection", "elementary component", "composite component" and "information packet".

At this point, we should ask: just what are the things we are connecting together? We have spoken as though they were components themselves, but actually they are uses or occurrences of components. There is no reason why we cannot use the same component many times in the same structure. Let us take the above structure and reverse it. We then get the following:

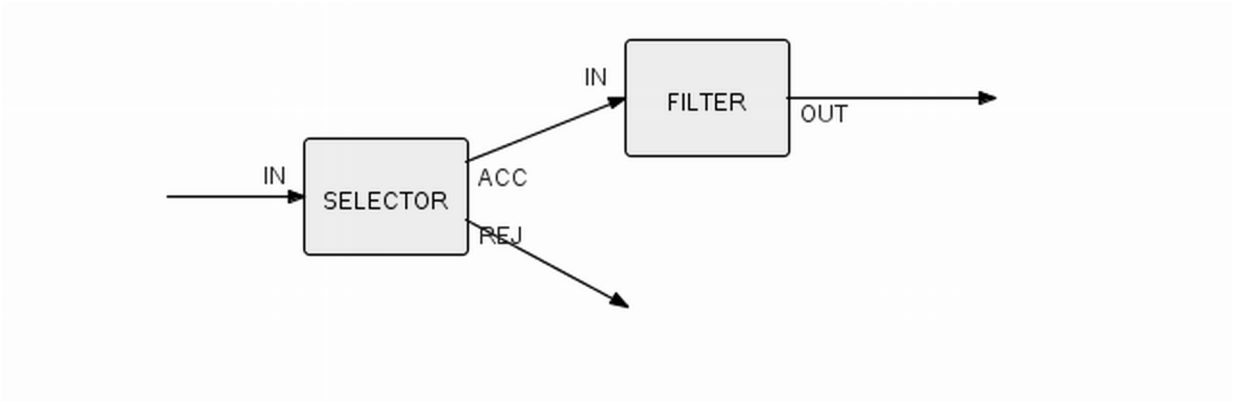



Figure 3.14

Let's attach another FILTER to the REJ port of SELECTOR. Now the picture looks like this:

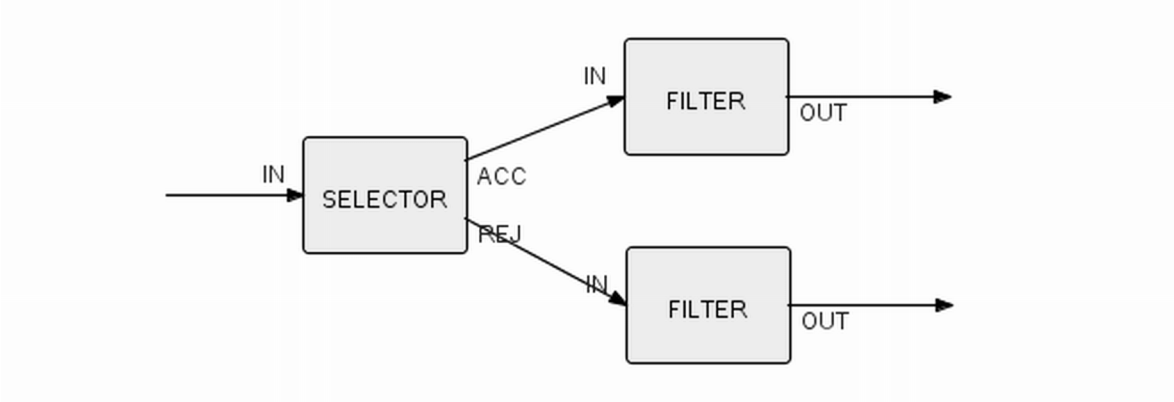



Figure 3.15

Here we have two occurrences of the FILTER component running concurrently, one "filtering" the accepted IPs from SELECTOR, the other filtering the rejected ones. This is no different from having two copiers in the same office. If we have only one copier, we don't have to identify it further, but if we have more than one, we have to identify which one we mean by describing them or labelling them - we could call one "the big copier" and the other "the little copier", or "copier A" and "copier B". In DFDM we took the function name and qualified it - in THREADS we make what might be called the "in context" function the name, and qualify it to indicate which component we are using. These component occurrences are called processes, and it is time to explain this concept in more depth.

In conventional programming, we talk about a program "performing some function", but actually it is the machine which does the function - the program just encodes the rules the machine is to follow. In conventional programming, much of the time we do not have to worry about this, but in FBP (as also in operating system design and a number of other specialized areas of computing) we have to look a little more closely at this idea. In FBP, the different components are executed by the CPU in an interleaved manner, where the CPU gives time to each component in turn. Since you can have multiple occurrences of the same component, with each occurrence being given its own series of time slots and its own working storage, we need a term for the thing which the CPU is allocating time slices to - we call this a "process". The process is what the CPU "multithreads" (some systems distinguish between processes and threads, but we will only use the term "process" in what follows). Since multiple processes may execute the same code, we may find situations where the first process using the code gets suspended, the code is again entered at the top by another process, which then gets suspended, and then the first process resumes at the point where it left off. Obviously, the program cannot modify its own code (unless it restores the code before it can be used by another process), otherwise strange things may happen! In programming terms, the code has to be read-only. In IBM's MVS, this kind of program is called reentrant, which is not quite as stringent as read-only, but read-only implies reentrancy, and I have found that making code read-only is a good programming discipline, as it makes a clear distinction between variable data, on the one hand, and program code together with constants, on the other. Although this may sound arcane, it is not really that far removed from everyday life. Imagine two people reading a poster at the same time: neither of them needs to be aware of the point in the text the other one has reached. Either one of the readers can go away for a while and come back later and resume at the point where he or she left off, without interfering in the least with the other reader. This works because the poster does not change during the reading process. If, on the other hand, one person changes the poster while the other is trying to read it, they would have to be synchronized in some way, to prevent utter confusion on the part of the reader, at least.

We can now make an important distinction: composite components contain patterns of processes, not components. This becomes obvious when you think of a structure like the previous one - the definition of that composite has three nodes, but two of them are implemented by the same component, so they must be different processes. Of course, they don't become "real" processes until they actually run, but the nodes correspond one-to-one with processes, so they can be referred to as processes without causing confusion. Here is the same diagram shown as a composite:

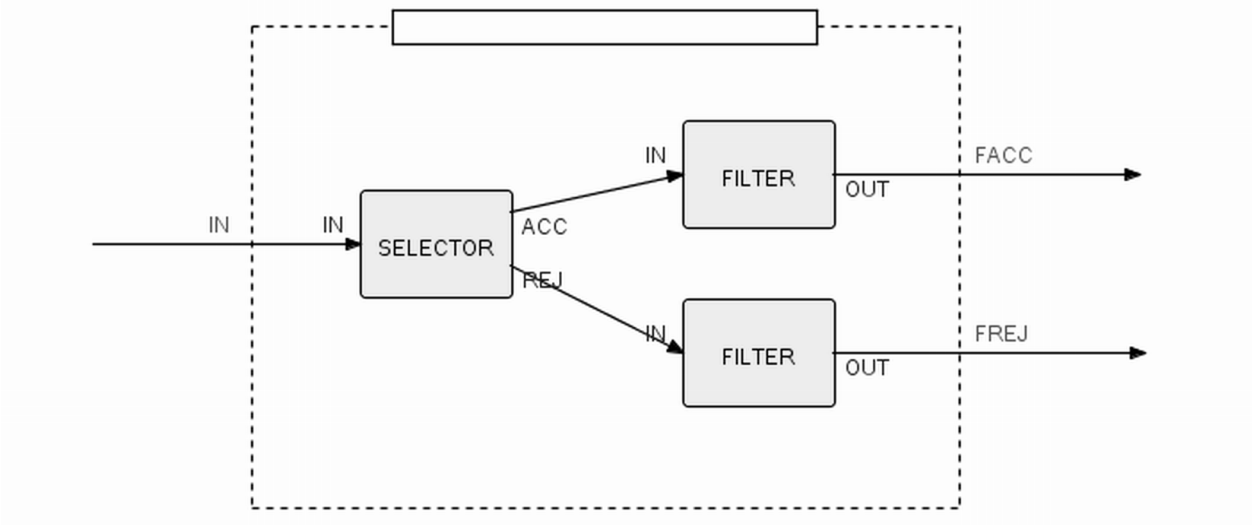
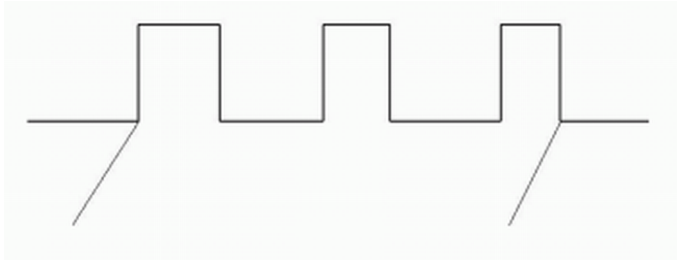



Figure 3.16

When this composite runs, there will be three processes running in it, executing the code of two components.

Lastly, I would like to introduce the concepts of data streams and brackets. A "stream" is the entire series of IPs passing across a particular connection. Normally a stream does not exist all at the same time - the stream is continually being generated by one process and consumed by another, so only those IPs which can fit into the connection capacity could possibly exist at the same time (and perhaps not even that many). Think of a train track with tunnels at various points. Now imagine a train long enough that the front is in one tunnel while the end is still in the previous tunnel. All you can see is the part of the train on the track between the tunnels, which is a kind of window showing you an ever-changing section of the train. The IP stream as a whole is a well-defined entity (rather like the train) with a defined length, but all that exists at any point in time is the part traversing the connection. This concept is key to what follows, as it is the only technique which allows a very long stream of data to be processed using a reasonable amount of resources.

Just as an FBP application can be thought of as a structure of processes linked by connections, an application can also be thought of as a system of streams linked by processes. Up to now, we have sat on a process and watched the data as it is consumed or generated. Another, highly productive way of looking at your application is to sit on an IP and watch as it travels from process to process through the network, from its birth (creation) to its death (destruction). As it arrives at each process, it triggers an activity, much like the electrical signal which causes your phone to ring. Electrical signals are often shown in the textbooks like this:



trailing edge leading edge

Figure 3.17

The moment when the leading edge reaches something that can respond to it is an event. In the same way, every IP has both a data aspect and a control aspect. Not only is an IP a carrier of data, but its moment of arrival at a process is a distinct event. Some data streams consist of totally independent IPs, but most streams are patterns of IPs (often nested, i.e. smaller patterns within larger patterns) over time. As you design your application, you should decide what the various data streams are, and then you will find the processes falling out very naturally. The data streams which tend to drive all the others are the ones which humans will see, e.g. reports, etc., so you design these first. Then you design the processes which generate these, then the processes which generate their input, and so on. This approach to design might be called "output backwards"....

Clearly a stream can vary in size all the way from a single IP to many millions, and in fact it is unusual for all the IPs in the stream to be of the same type. It is much more common for the stream to consist of repeating patterns, e.g. the stream might contain multiple occurrences of the following pattern: a master record followed by 0 or more detail records. You often get patterns within patterns, e.g.

```
'm' patterns of:
   city record, each one followed by
   'n' patterns of:
       customer record, each one followed by
       'p' sales detail records for each customer
```

Figure 3.18

You will notice that this is in fact a standard hierarchical structure. The stream is in fact a "linearized" hierarchy, so it can map very easily onto (say) an IMS data base.

To simplify the processing of these stream structures, FBP uses a special kind of IP called a "bracket". These enable an upstream process to insert grouping information into the stream so that downstream processes do not have to constantly compare key fields to determine where one group finishes and the next one starts. As you might expect, brackets are of two types: "open

brackets" and "close brackets". A group of IPs surrounded by a pair of brackets is called a "substream". In THREADS, we use IPs with "type" of "(" and ")" for open and close brackets, respectively.

We have one more decision to make before we can show how we might use brackets in the above example. We have two cases where a single IP of one type is followed by a variable number of IPs of a different type (or substreams). The question is whether the open bracket should go before the single IP or after it. In the former case, we might see the following (I'll use brackets to represent bracket IPs):

```
 < city1
     < cust11 detl111 detl112...>
     < cust12 detl111 detl112...>>
  < city2
     < cust21 detl211 detl122...>
     < cust22 detl221 detl222...>>
  etc.
```

Figure 3.19

In the latter case, we would see:

```
    city1 <
       cust11 < detl111 detl112...>
       cust12 < detl111 detl112...>>
    city2 <
       cust21 < detl211 detl122...>
       cust22 < detl221 detl222...>>
  etc.
```

Figure 3.20

From the point of view of processing, these two conventions are probably equivalent, but I tend to prefer the first one as it includes each customer IP in the same substream as its detail records, and similarly includes each city in the same substream as the customers that belong to that city. As of the time of writing, there is no strongly preferred convention, but you should make sure that all specifications for components which use substreams state which convention is being used. By the way, a very useful technique when processing substreams is the use of "control" IPs to "represent" a stream or substream. Both AMPS and THREADS and some of the versions of DFDM have the concept of a process-related stack, which is used to hold IPs in a LIFO sequence. In Chapter 9, I will be describing how these concepts can be combined.

We have now introduced the following concepts:

- process
- component (composite and elementary)
- information packet (IP)
- structure
- connection
- port and port element
- stream
- substream
- bracket

Normally at this stage in a conventional programming manual we would leave you to start writing programs on your own. However, this is as unreasonable as expecting an engineer to start building bridges based on text-book information about girders and rivets. FBP is an engineering-style discipline, and there is a body of accumulated experience based on the above concepts, which you can and should take advantage of. Of course, you will develop your own innovations, which you will want to disseminate into the community of FBP users, but this will be built on top of the existing body of knowledge. You may even decide that some of what has gone before is wrong, and that is standard in an engineering-type discipline also. Isaac Newton said: "If I have seen further than other men, it is because I have stood on the shoulders of giants." Someone else said about (conventional, not FBP) programming: "We do not stand on their shoulders; we stand on their toes!" Programmers can now stop wearing steel-toed shoes!

Before we can see how to put these concepts together to do real work, two related ideas remain to be discussed: the design of reusable components and parametrization of such components (see the next chapter).

# Chap. IV: Reuse of Components

> "Reuse in DFDM is natural. DFDM's technology is unsurpassed in its promotion of reuse as compared to other reuse technologies currently being promoted" (from an evaluation of DFDM performed by an IBM I/S site in the US in 1988).

So far, we have spoken as though components are created "out of thin air" for a specific problem. You may well have suspected that I selected my components to illustrate certain concepts, without worrying about whether they would be useful in a real application. Well, the good news is that the kinds of components we have run into are in fact the ones which experience shows are very useful for building real applications. The bad(?) news is that it requires a large amount of experience in programming and a certain creative flair to come up with useful components. This should not be so surprising when you think about useful tools you are accustomed to using in real life. Where and when was the first hammer invented? Imagine a whole series of "proto-hammers", for instance reindeer horns, rocks attached to sticks, etc., gradually evolving into what we are used to today, with a claw on one side, and balanced just so.... Perhaps we should qualify that by saying "in the West". Different cultures will come up with different tools or different forms of the same tool. I seem to remember from my anthropology classes that the Australian aboriginals have a wonderful tool which is a combination of all the things they find most useful in their everyday life, and yet is highly portable. It is a combination spear-thrower, shield, dish and fire-starter. These kinds of tools are not invented overnight - they take time and experience, requiring progressive refinement by many creative people. The best tools will always evolve in this way. Tools may also pass out of use as the need for them diminishes, or they are replaced by something more effective - buggy whips are the classical example, but usually it happens without our even noticing! When did they stop putting running-boards on cars? No, you don't need to phone right away! Clearly, culture and the tools we use are closely intertwined - we have strong ideas about what is the right tool for a given job, but another culture may in fact have a different definition of what that job is.... In the West we consider using a knife and fork the "proper" way to eat food - a few centuries ago, we were spearing it with the point of a dagger. Knives and forks

in turn mean that an acceptable Western meal might include some very large chunks of meat, or even half a bird. In the Orient, on the other hand, people have been using chop-sticks for a very long time, which requires that the food be served in bite-size pieces. Notice that the choice of tools also helps to determine what part of the serving is performed by the diner and what part by the cook behind the scenes.

One other thing we should consider is the need to be able to use the tool in unforeseen situations. A useful tool should not be too restrictive in the ways it can be used - people will always think of more ways to use a tool than its original designer ever imagined. Wayne Stevens (1991) tells a story about an airline attendant using a hearing set (that little plastic stethoscope you plug into the arm of your chair) to tie back some curtains. Elegant? No. Effective? Yes! We don't want to make a hammer so intelligent that it can only be used on nails.... Another example: why do some UNIX™ functions have non-obvious names? There are well-known cases where a tool was originally designed for one job, but people found that it was even more useful for some function the original designer did not foresee. This is in fact a testimony to the robustness of these tools. We will run into examples of this kind of thing in FBP.

Just as in the preparation and consumption of food there are the two roles of cook and diner, in FBP application development there are two distinct roles: the component builder and the component user or application designer. The component builder decides the specification of a component, which must also include constraints on the format of incoming data IPs (including option IPs) and the format of output IPs. The specification should not describe the internal logic of the component, although attributes sometimes "leak" from internal to external (restrictions on use are usually of this type). The application designer builds applications using already existing components, or, where satisfactory ones do not exist, s/he will specify a new component, and then see about getting it built.

Component designers and users may of course be the same people, but there are two very different types of skill involved. This is somewhat like the designer of a recent popular game, who admitted he was not particularly fast at solving it - his skill was in designing games, not in playing them. The separation between makers and users is so widespread in real life that we don't pay any attention to it unless it breaks down. In industry, as Wayne Stevens points out, we take for granted the idea that airplane builders do not build their own chairs - they subcontract them to chair manufacturers, who in turn subcontract the cloth to textile manufacturers and so on. In contrast, the world of conventional programming is as if every builder designed his own nails, lumber and dry-wall from scratch. Talk about "reinventing the wheel" - in conventional application development we reinvent the rubber, the nuts and bolts, and even the shape of the wheel!

I'd like to talk a little bit about how useful components are developed. They are unlikely to emerge from a pure "top-down" approach, or from a pure "bottom-up" approach. In the first case, you do not discover dry-wall by progressively breaking down an architect's drawing of a house.

In the second case, people who dream up useful components have to be willing to subject them to rigorous testing in real life situations. Even after this has been done, they still may not sell. Nobody in industry would bet the business on some untried tool which had never been evaluated in the field (well, usually not), and yet we do this frequently in application development. Another of Wayne Stevens' recommendations is not to build a generalized tool until you have found yourself doing the same thing three or four times. Otherwise, you may find yourself investing a lot of time and effort in features that nobody will ever use, and you may find yourself unable to respond to customer requests for features they really do want.

In FBP a lot of the basic components have analogues in an area which is no longer well-known, but has been very productive of generalized components over a number of years - namely, Unit Record machines. In those days we had specialized machines, such as sorters, tabulators, collators, etc., and people learned to wire (parametrize) them and link them together into applications very effectively. And you didn't need a college degree to get applications working. In fact, I once figured how to solve a problem with a tabulating machine plug-board, straight out of the bath, over the phone, dripping wet, without even notes or a schematic to look at!

Just as Unit Record machines worked with streams of punched cards, the corresponding FBP components work with streams of IPs. Let's call these "stream-based" components. Examples of such components are:

- sort
- collate
- split
- replicate
- count
- concatenate
- compare

These all have the characteristic that they process data streams and that they require very little information about the format of their incoming data streams. They typically have well-defined application-independent functions.

We might expand the list with some general-purpose components which get down to the data field level, but still do not "understand" business processing. One such component might be a generalized transform component. I believe such a component, properly parametrized, could in fact do a lot of the processing in a given business application. Nan Shu of IBM in Los Angeles has written extensively about a language which she calls FORMAL (Shu 1985) - its function is to take descriptions of files and the transformations between them and use them to do the transforming automatically. She has found that a large amount of business processing consists of

moving data around, changing its coding, and doing table look-ups, e.g. one application might use a number for each US state, while another might use a two-character abbreviation. This suggests that another type of function in this same class is a generalized table look-up function, and in fact we have built several for DFDM.

There is another general class of components known as "technology-dependent". These components usually require specialized knowledge to build, but, once created, can be used by people who are not as technically skilled. They thus encapsulate specialized knowledge. They will usually be written in a lower level language. We had an interesting example of this some years ago: we had a person who was an expert on paper tape. Paper tape is (was?) a medium with its own quirks. It has special codes for various purposes, and in particular has a convention for correcting data (you punch all holes across the offending character, which is then treated as though there was no character there at all). This individual was able to write some components which generated regular IPs, so that no other component in the application needed to know that the input was paper tape. You could build and debug an application using regular I/O, and then, after you had it working, you could unplug the reader module and replace it with the paper tape reader. This meant in turn that testing could be done without the tester having to monopolize a scarce (and slow) piece of equipment.

The two most widely used technology-dependent components are "Read Sequential" and "Write Sequential". As you might expect from the names, these convert file records into IPs, and IPs into file records, respectively. A matching Read/Write pair of components can be used to encode and decode any file format desired. For instance, you might decide that the medium you are using is so expensive that you want to compress the data as it is stored, and expand it as you retrieve it. Another useful function pair might be encryption/decryption. In fact, any Read/Write component pair can be thought of as embodying a data organization. Generalizing this thought, a sequential Read/Write component pair provides a conversion between a format suitable for processing and a linear format on some medium. For instance, we built a Read/Write pair which was specialized for dumping tree structures onto a linear medium and rebuilding them later.

You may have noticed by now that components very often come in matched pairs, e.g. split/merge, read/write, compress/expand, encrypt/decrypt, etc. This is characteristic of many FBP components - what one component does, another one undoes. In fact the combination of a component and its inverse should ideally result in a stream identical to the original input stream, just as in mathematics multiplying a number by its reciprocal results in unity, or composing a function with its inverse results in the Identity function.

Using separate Read and Write processes not only gives the separation between logic and I/O which is recommended by the majority of application development methodologies, but actually reduces elapsed time. The reason for this surprising result is that in FBP an I/O process which has to wait on I/O only suspends itself - other processes can proceed. This means that FBP applications tend to use as much CPU time as they are allowed to by the operating system. We

will be talking more about performance later on.

Another interesting group of components deriving originally from Unit Record are those connected with report generation. These include such functions as pagination and generation of page headings and footings, as well as totalling and subtotalling, cross-footing and other report generation functions. Reports are very often the main vehicle of communication between an application and the humans who use it, and the importance of these facilities to the average business is borne out by the remarkable longevity of IBM's RPG, which, while often regarded as old-fashioned, is still fulfilling a real need in the market-place. Later in this book, I will describe a Report Generation component which was used extensively in our shop.

A good guideline for the functionality of a component is that its specification should not exceed about a page. Some FBP experts have gone so far as to say that the summary of a component's function should not exceed one paragraph, and should not have the word "and" in it. Another guideline we have found useful is that generalized components should not have more than 4 ports (array ports only count as one port). Of course, these guidelines are not mutually exclusive, and they are only guidelines - some components bring so much function together that their parameters are essentially mini-languages, but their usefulness may outweigh any awkwardness in parametrization.

The last category of component is that of "business components". These embody business rules, and should hopefully be relatively simple, especially if you have used the other categories of component as far as possible. We can imagine business components for different business areas - banking, oil and gas, and so on. Some will be more mathematical, others less so. In all cases they represent the knowledge of some business expert.

After functionality, one of the major considerations in connection with designing business components is the likelihood of change. There are some types of business logic which hardly ever change, and others which are changed every time they are run. An example of the latter might be the logic to generate an employee's taxable income statement at the end of the year. It is changed every year and run once a year. It would be very nice if our governments could send out a single reusable component every year which companies could then just plug into their own payroll programs. This also gets back to the question of roles: who installs the new module? Application development or operations staff? If the former, you have an ongoing need for application developers indefinitely; if the latter, can you be sure that the new component will be adequately tested? On the other hand, given the backlog of work that application development usually faces, something which can just be loaded up and run by operational staff is certainly attractive.

It would help if such a component does as much validation of its input data as possible to make sure it is being used in the right context. Ideally a component should never crash - in practice, of course, it is almost impossible to prevent one component from destroying another's data, but it is

certainly possible to add validation logic to protect against (say) data format errors. The reusable module could also require that incoming data be tagged with a particular descriptor. Then if the required data format changes, you just need to change the descriptor name. Descriptor names are typically part of the specification of a reusable component, so this fits quite nicely.

The above discussion is really another form of the old compile-time versus run-time debate. In FBP, compile-time comes in two flavours: component-level and network-level. Actually parameters can be specified inside a composite component and still be outside the elementary component which they control! I predict that eventually a lot of business logic will be embodied in rules held in rules data bases. Such rules, written in a suitable language, can then be modified by people outside the normal application development group. These rules may not even be expressed in what a programmer would recognize as a programming language. A forerunner of this is the IBM Patient Care System, in which a surprising amount of the system logic (including screen layouts) was held in the form of tables, designed to be updated by senior clerical or nursing staff. This was very effective, as these were usually the people who had to use the system, and had the most operational experience with it. Again we see the separate roles of application-developer and application-user. If it bothers you to put so much control in the hands of end users, either implement authorization systems to make sure only the right people can modify key data, or specify the rules as tables hard-coded in the application definition, but outside the components that refer to them. This way, control remains in the application development group, but systems become much easier to modify and debug. However, we really should be moving away from requiring the DP department to do all systems maintenance.

If I am right that we will eventually see more and more business logic being either imbedded in reusable components or captured as explicit rules on disk, then the role of the current higher level languages in the future should diminish. We found in our experience that, given a powerful set of reusable components, people would go to enormous lengths to avoid writing HLL code. Much of the time, the resulting poorer performance does not matter - the Kendall Report (1977) contrasted the running time of the average program with the person-months required to develop it. Programs that took 6 person-months to develop might run for a few minutes over their entire lifetimes. So, most of the time, minor increases in the amount of CPU time really make no difference. Only in the case of long-running jobs run regularly is it worth-while to do performance tuning, and, as we shall see in a later chapter, it is much better to instrument an FBP application to find out where your real bottle-necks are than to try to guess ahead of time and waste time optimizing code which doesn't affect your system's performance much. There are in fact a number of ways to do performance tuning in the FBP environments, once we figure out where the real leverage is.

Far more important than CPU time is human time, and the fundamental question is really what is the best way to spend valuable human time. When deciding to develop a new component, you must take in account the expected return on your investment ("ROI" for short). Every new component must be documented, tested, supported, advertised and incorporated into educational

material (well, ideally - sometimes not all of these happen!). Small wonder, then, that our experience with FBP shows that application developers using FBP avoid writing new code - being responsible people, they are aware of the burden they take on the moment they start coding a new component. However, this realization isn't enough - we have to change the economics of the situation. Code is a cost item, as Dijkstra and others have pointed out, and someone who adds to the total amount of code when it is not justified is costing your company money, now and into the future. I have started suggesting, only half in jest, that programmers should be "penalized" for each line of code they write! In fact, some program improvements involve removing code - is this negative productivity?! It has been pointed out many times that people will modify their behaviour according to how you measure them - and companies which still measure productivity in Kloc (thousands of lines of code) get what they deserve! Conversely, someone who produces a useful reusable component improves the productivity of all of its users, and deserves to be rewarded - some companies have already started trying that - the key word, of course, being "useful". N.P. Edwards, who I mentioned in an earlier chapter, was a key player in getting IBM to move to reusable parts in the hardware area, and he has told me that the key breakthrough there also was in changing the economics of hardware development.

Someone who has talked and written extensively about the importance of reuse is T. Capers Jones (e.g. Jones 1992) - he has also been aware of my work for some years and has been supportive of it. He has been active in promoting the use of code-independent metrics, such as Allan Albrecht's now well-known Function Points, for measuring productivity and has done a lot of work on the potential of reuse for reducing the costs of application development.

How do we know whether a tool is useful? The only way is to measure its use. Will people use it? They will if it fits the hand, and if you provide support and education for it. That in turn means you have to have the infrastructure in place to allow your company to take advantage of this new technology, and measures and incentives to get people moving in the right direction.

There is also the opposite question: what if the tool is "less than perfect"? Just as with real tools, there is no perfect tool - there are only tools which fit your hand more or less conveniently. Like many programmers who tend to be perfectionists, you may be tempted to postpone putting something on the market because you feel it isn't finished yet. The question should be: is it useful as it is? You can always enhance it as time goes on, provided you keep the interfaces stable (or provide "expansion ports" but maintain upward compatibility). In fact, after it has been in use for a while, you may find that the extensions people really want are not at all what you expected. Since your reusable component will hopefully be in widespread use by this time, it is important that you allow extension while maintaining upward compatibility. In FBP, the fact that ports are named helps you to do this; also parameters (described in the next chapter) should be designed to be extensible. Parameters can be in string format, with delimiters, or, if fixed form, it is a good idea to insert a few zero bits - these can always to changed to ones to indicate that there is an extension portion.

Another kind of modification which will happen to your modules occasionally is error correction. It is certainly a pleasurable feeling to know that you have improved a component which many people are or will be using, and you might think that your users will welcome the change with open arms. Unfortunately some of your users may have adjusted to the error, and will need to be convinced that you know what is right for them. The other thing users do is take advantage of undocumented features. I talked about a tool fitting the hand - it may fit the hand, even with an error in it. One team found an error in one of the DFDM components, but instead of telling us about it, they carefully compensated for it. When we fixed it, their programs stopped working! I think they were quite indignant for a while until everybody realized what had happened. We had to spend some time explaining that everyone would be much better off if we fixed the bug rather than leaving it the way it was! There is a very important rule which you should impress on your users: If it isn't documented, don't trust it. IBM learned the value of this one by bitter experience and has accepted its wisdom since the day some bright user discovered an undocumented instruction on one of the 700-series machines. When IBM started making invalid instructions result in exception conditions, I'm told quite a few programs in universities and other places stopped working!

The next question is: how will people find out about these components? There is a common misconception that reusable componentry doesn't work unless you have an elaborate catalogue, which people can interrogate to find the tool they want. On the other hand, Wayne Stevens has pointed out that most examples of reuse in everyday life are done very naturally without any catalogue. We know by heart most of the things we use commonly. Let's say you go into a hardware store because you want to attach a wood base onto a ceramic pot - you will be familiar with half a dozen types of fastener: glue, nails, screws, rivets, etc. Most of the time you will know exactly what kind of glue to use. In this case, let's say you are not quite sure what is best. You still don't have to scan the entire store - most of the time, you can go right to the shelf and start reading labels. What do you do if you are not sure where in the store to go to? You ask a store clerk, who may in turn pass you onto someone who is an expert in a particular area. If your requirements are really unusual, the clerk may have to consult a catalogue, but this is likely to be a rare case. The point is that effective reuse doesn't require catalogues, although they can certainly help.

To try to measure the productivity gains we were getting from DFDM within IBM Canada, we kept statistics on the amount of reuse taking place over a number of projects. The figures for three projects are shown in the following diagram (the numbers relate to components):

| PROJECT | Type | Unique | Occurrences | Reuse Factor | 1 / Figure of Merit |
|---|---|---|---|---|---|
| A | Project | 133 | 184 | 1.4 | **3.7** |
| | Gen Purpose | 21 | 305 | 14.5 | |
| | Total | 154 | 489 | 3.2 | |
| | **GP/T** | **0.14** | **0.62** | | |
| | | | | | |
| B | Project | 46 | 48 | 1.0 | **7.7** |
| | Gen Purpose | 17 | 306 | 18.0 | |
| | Total | 63 | 354 | 5.6 | |
| | **GP/T** | **0.27** | **0.86** | | |
| | | | | | |
| C | Project | 2 | 54 | 27.0 | **135.0** |
| | Gen Purpose | 8 | 216 | 27.0 | |
| | Total | 10 | 270 | 27.0 | |
| | **GP/T** | **0.80** | **0.80** | | |

In this chart, "project" means 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 (about 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", to use Bob Kendall's phrase (Kendall 1988), 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 that 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 1 (Project C had a factor of 27.0). In the third example in the above chart, the programmer only had to write 2 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!).

[In Bob Kendall's "Figure of Merit", obviously smaller is better! In the on-line version, I have shown the reciprocal as it seems to be more intuitive to have the larger number indicate better reuse.]

Although we thought at first that this last case was just a quirk, we turned up quite a few applications which were not that different from this one (e.g. Rej's letter quoted in the Introduction).

Here are some figures from an evaluation of DFDM quoted from at the beginning of this chapter:

> All of the function in the DFDM pilot application is performed by 30 unique coroutines (this is the number of coroutines that an individual would need to be familiar with in order to understand the function of the application).
>
> A total of 95 occurrences of these 30 coroutines make up the application ..... providing a 3:1 reuse ratio.
>
> These 95 coroutines are leveraged through the use of subnets and CNS [Compiled Network Specification] networks to perform the equivalent work of 225 unleveraged coroutines.

Some companies have tried to encourage people to write generalized code by offering them money or kudos. One counsel I would give them is that you need to monitor not how many components someone has written, but how often it is used. An appropriate analogy is the system of royalties in the publishing industry. Every time a module is used, the author should get some kind of token, be it money or recognition. This will ensure that your company will not accumulate a collection of wonderful, Rube Goldbergish gadgets sitting on the proverbial shelf, gathering dust.

Let us say that you are all convinced that reusable code is the way to go - how do we get it adopted in your particular shop? You will find (unless all your people are super-altruists) that the biggest enemy of reuse is not technology - it is human psychology. While many people will accept the new ideas enthusiastically, others will resist them, and for several different reasons. People who have become good at delivering applications under time pressure very often feel that they must at all costs maintain control of everything they use, and in fact all their experience has taught them that this approach works. Components developed by others will be on their critical

path, and they will be pulled between the desire to reduce their own effort by using pretested components, and the fear that the components they are relying on will not be ready in time, will break or will not be maintained as the environment changes. They have to become convinced that someone will support these components - if necessary, on a 24-hour basis. This may not be necessary technically, but may be very necessary psychologically!

Another source of resistance is simply that some programmers love the bits and bytes and don't want to become mere combiners of precoded components. There is a role for these people, writing the components to specs. As we said above, two different roles seem to be emerging: component builders and component users. In my view the latter need skills very similar to those required by analysts. They need to be able to talk to users, gather requirements, and even build systems or prototypes of systems. For the more complex parts or parts which have to perform better, they can subcontract parts to the component builders. This is the domain where the programmer's programmers ("Merlins", as a friend of mine calls them) can shine. In some senses, a component becomes an encapsulation of their particular skill or knowledge. I have found that it makes sense to get "tighter" about the external specs and "looser" about how the code is built internally. This lets them express their creativity, while still serving the needs of the organization as a whole. Of course, it must not be so poorly written that it doesn't perform well! And it absolutely must deliver the function according to the specs! Once those are assured, then your only concern is maintainability. Generalized code should be maintainable, but you probably don't have to control the format of every internal label!

A programmer once said to me, "I don't like DFDM because I don't get dumps"! At the time I took this to mean that because programs built using FBP tend not to crash, it is hard for programmers to get a feel of how they work. Does not knowing how the engine of your car works make you nervous? It probably does affect some people that way, but most of us don't care. Later, I realized that it also brings up the very fundamental question of trust - if the users of a package don't trust the package or its vendor (same thing, really), they are not going to be happy... And trust is fragile: hard to build up, and easy to damage.

Let us suppose that your company has become convinced that developers should not keep "reinventing the wheel", but that, like most companies, you have only reached the stage where you are maintaining a library of shared subroutines. How do we get formalized sharing of components in place? Suppose I find out that Julia is working on a module which is pretty close to what I want but it needs some tweaking to fit my needs. In most shops, we don't even know what to call it. Companies that have just started to grapple with naming standards often think it's neat for module names to start with the project code. For instance, if I am managing project ABC, then I can name all my modules ABC-something. This way, I don't have to worry about my module names conflicting with those of other projects. Even the library names will often have ABC built into them! So, even to be able to find the code, we usually have to have some kind of enterprise-wide naming convention. Next question: who does the modification of the code and

who pays for it? What if Julia's schedule slips and starts to impact my schedules? Even if everything goes really well, who will maintain it, document it, and support it?

Many writers about reuse agree that the only solution is to set up an independent department to write and maintain components. This department must have enough resources to do the job properly, which also involves publicising and selling their product. One tendency which must be resisted is that such departments often get tied up producing complex, generalized tools for a few users, or even for none - they just figure the component would be neat and they'll worry about selling them afterwards. Remember the principle of ROI: the company as a whole will get more bang for the buck out of a lot of simple tools, especially if they communicate well with each other, rather than from a few very complex ones. Since good tools will often start as special-purpose modules which some other group has found useful, there must be a path for promoting such ad hoc components to a place where other people can find them and rely on them. Our centralized software support department must have ways to beat the bushes for new and interesting components and must then have ways to evaluate whether potential customers are interested (otherwise why go to all that trouble?). It must also avoid getting sucked into writing or upgrading complex tools which have only a small market. It is a service organization, so it must be service quality oriented, not just a group of self-styled experts who know what is best for everyone else. It must become entrepreneurial, but not exclusively bottom-line oriented. In short, it must follow good financial and engineering practices. If this takes a major shake-up in the way your organization is structured, then you should really get started as soon as possible!

I believe that, unless companies start to bring engineering-type disciplines to application development, not only will they fail to take full advantage of the potential of computers, but they will become more and more swamped with the burden of maintaining old systems. You can't do a lot about old systems - I know, I've seen lots of them - but new systems can be built in such a way that they are maintainable. It should also be possible to gradually convert old programs over to the FBP technology piece-meal, rather than "big bang". A director I once worked for called this "converting an iceberg into ice-cubes"!

I believe all true disciplines follow a sort of cycle through time, which may be represented using the following diagram:

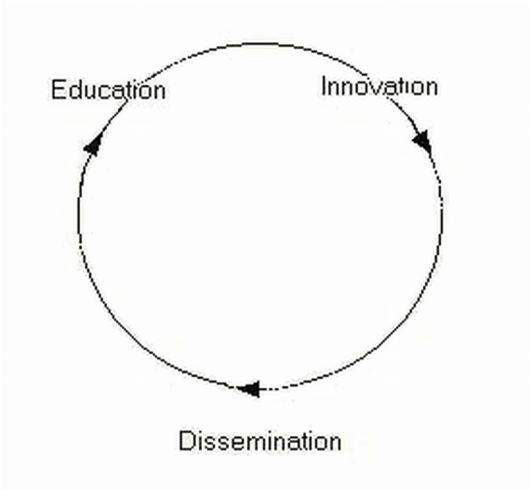



Figure 4.1

Innovation can only be founded on a base of solid knowledge of what went before - otherwise we are doomed to keep rediscovering the same old stale concepts. On the other hand, tradition without innovation cannot result in progress, and innovation is useless unless the word is gotten out to people who can make use of it. As I pointed out above, business application development has not really changed significantly since I started in the business in 1959 - but I really believe that now, at long last, we can start to see the promise of application development becoming a true engineering-style discipline.

# Chap. V: Parametrization of Reusable Components

I would now like to describe the way generalized FBP components are made reusable, using one additional, very powerful FBP mechanism, the "Initial Information Packet" or "IIP". IIPs were developed by E. Lawton of my old department at IBM in response to some of the problems we ran into using DFDM's parameter facility.

Let's say you have built a component like one of the ones described in the previous chapters. Let's call it "Select". It should be clear from the foregoing that this component can be used in a variety of contexts, as long as it is sent data in the format it expects. Because a component communicates with the outside world only through data being sent to or received from its ports, it can be held in object form, and never modified. Such reuse is often called "black box" reuse, to suggest the idea that the user cannot see the insides of the component. In addition, since the "black box" never needs to be modified, once its developer gets it working, it can be relied on to work correctly in any context. This is the converse of "white box" or "clear box" (source level) reuse, which is what most so-called reuse tools provide today. This type of reuse is easy to provide, but in my view doesn't buy its users much. It may reduce the cost of developing new code, but the amount of net new code which has to be maintained still increases. Furthermore, if a bug is found in the reused code, there is no easy way to tell whether it is safe to fix all instances of it - if you can even find them (with some reuse tools you can't even do that).

We will of course have to tell our black box Select component which fields to select on, as, otherwise, it will only be able to select on whichever fields were hard-wired into it. Suppose we want to tell it that it is to do its selecting on the contents of a 6-byte field starting at offset 23 in each incoming IP, and also want to give it a list of acceptable field values. In classical

programming, we do this kind of thing by having the calling program specify parameters. In FBP we do something similar, but there is no user-written calling program in which to specify the parameters. Instead there is a way for the application designer to specify this information, right in the application structure definition. This mechanism is called an Initial Information Packet (IIP). We also need an additional port on the component (let's call it OPTIONS - it can have any name we want). An IIP can be generated as part of the structure and associated with the chosen port.

Once the process is started, an IIP is turned into an ordinary IP by the component issuing a "receive" service call against the port the IIP is connected to. This has the added advantage that the OPTIONS port can also be fed by an upstream process instead of an IIP, so component options can either be decided at structure-building time or deferred until execution time, without any modification of the component being required. What the component sees when it does its receive from the OPTIONS port is an ordinary IP. In what follows, we will sometimes refer to this as an "options IP" - options IPs may start out life as IIPs or may be generated by upstream processes, but their function is primarily to control execution, rather than to carry data (obviously this is not a hard and fast distinction, and you may find a need for IPs which combine both functions).

One last point about options IPs: a major decision for the component designer is whether to make them free-form or give them a fixed layout - the former will generally be easier to specify, but is going to cost more processing time to scan for delimiters, convert numeric values, etc. However, since this processing is usually done just once, at the beginning of the run, it may not be significant in the context of total processing time. THREADS opted for free-form IIPs for its "off the shelf" components because of the ease of use factor. One other thing you should consider if you choose to use free-form IIPs is that you will need to decide on delimiter conventions, e.g. you might opt for commas or blanks, and you might decide to use brackets to group together sets of option values. This in turn means some convention will be needed for specifying character strings which might contain delimiter characters as valid values (the old "quotes within quoted strings" problem).

Fixed-form options, on the other hand, will take less machine processing time and avoid the above-mentioned problems of delimiter conventions. Conversely, they are more likely to result in alignment errors, and are harder to make open-ended. There are other options, however, within the fixed-form category:

- use some front-end tool to generate the fixed-form IIP, or

- use a descriptor (see Chapter 11 on Descriptors) to access the fields in the options IP.

Let's say we want to write a generalized selector where the column number, field length and permissible values are received from an OPTIONS port at execution time. Using a long, shallow rectangle to represent an IIP (by the way, you can't attach IIPs and regular processes to the same connection), we might represent our selector with its options IIP as shown below. The field being

selected on starts at offset 23, for a length of 1 byte. IPs with a value of A at that position will be sent to the zero'th element of port OUT, IPs containing a B go to the number 1 element, and so on.

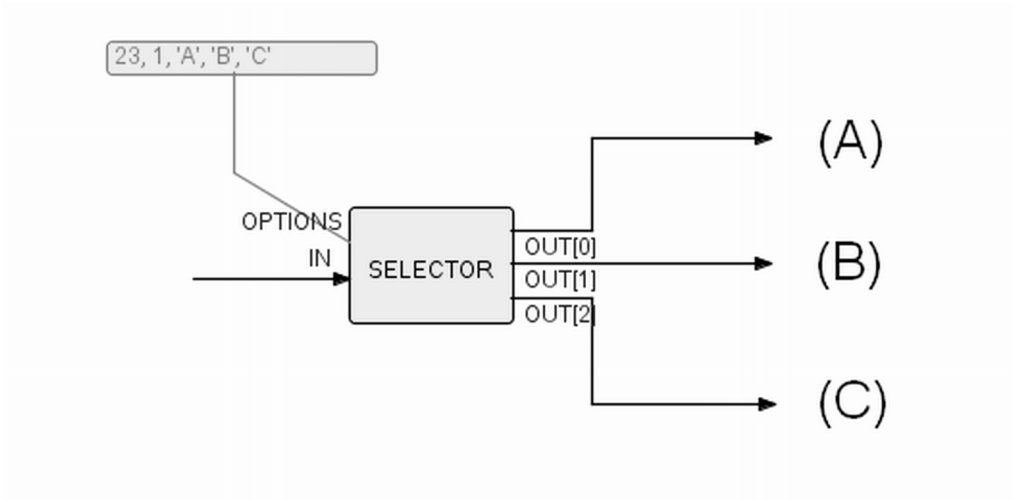



Figure 5.1

In a structure diagram, it is often useful to be able to show parameter values right in the picture. Note that in this example we have shown a free-form IIP.

To convert this diagram to use a connection instead of an IIP, change the block feeding OPTIONS to a component, and attach it to the OPTIONS port, as follows:

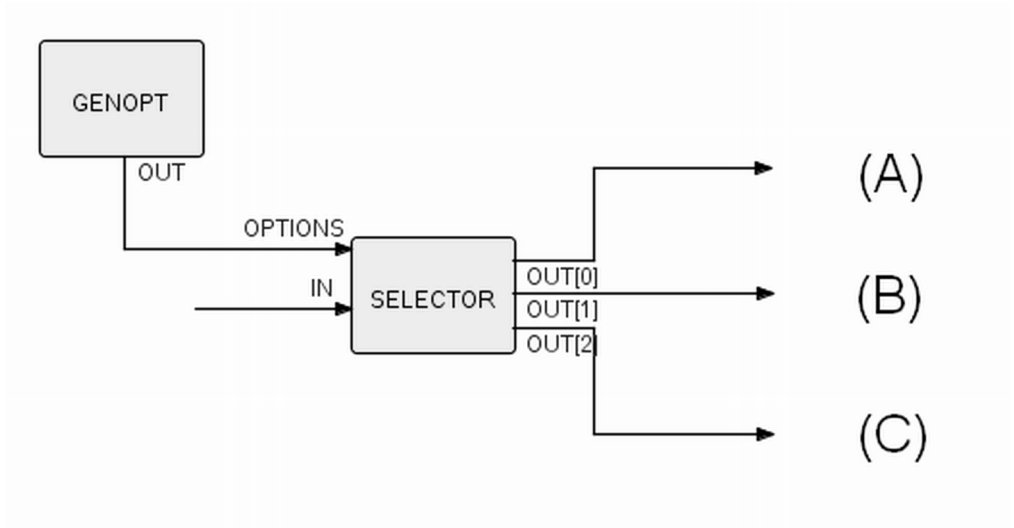


Figure 5.2

By way of comparison, in DFDM processes were parametrized by means of a variable-length character string (2 byte binary length followed by a character string) passed to the component at activation time (analogously to the way a parameter string is passed to a job step in IBM's MVS). This meant that parameters specified in the network and parameters coming from an upstream process could not be handled in a uniform manner.

There is a general problem of passing parameters from the "outside world" into networks. In DFDM's approach to parametrization, external parameters were only passed to the outermost structure, and there was a system of "inheritance" which allowed lower level structures or component access to parameters at the next higher level. Although we haven't tackled this problem in THREADS yet, the IIP technique should make this whole area much simpler as special components can be written to obtain external parameters and pass them into the network like any other IPs.

At a more general level, parametrization of components may be thought of as a spectrum, running from low to high. Low parametrization (few or no parameters) occurs when a component has no variability - either because it is custom-coded for a particular application, or because it is so simple that it always does the same work in the same way. For instance, there is a very useful component in all existing FBP systems which simply accepts and outputs all the IPs from its first input port element, followed by all the IPs from its second input port element, and so on until all the input port elements have been exhausted. In DFDM it was called the "Sequencizer" (some of my friends like to play fast and loose with the English language). This component is often used to force a sequence on data which is being generated randomly from a variety of sources. One example might be control totals being generated by different processes which you then want to print out in a fixed order on a report. You know the sequence you want them displayed in, but you do not know the time at which they are going to be created. Schematically:

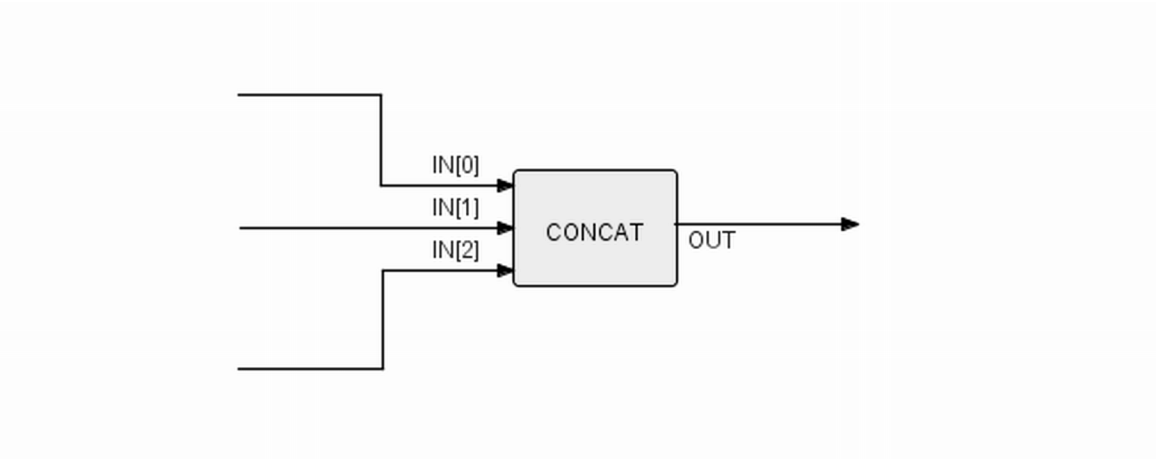



Figure 5.3

Its function is so simple that it doesn't need an options port. Now you may be asking, "Why not simply take all the incoming data streams and merge them into a single input port?". The answer is that you can do this, but the effect is somewhat different. What happens in that case is that the incoming IPs are merged in a "first come, first served" sequence, which is not what you wanted. However, there is a down side to this function: because the data from input port element 1 is held up until Concatenate knows that port element 0 has been closed, and so on for the remaining ports, there is the potential for deadlock. Deadlocks are not as bad as they sound, and there are well-established ways of detecting places where they might occur, and of preventing them before they can occur. In FBP, deadlocks are viewed as a design-time problem. We will be talking about the cause and prevention of Deadlocks in a later chapter (Chapter 16).

The Select component has a median level of parametrization, and most of the components we will be talking about in this book are similar, but there are occasional components which have so many parameters that they can almost be thought of as mini-languages. If parameters are replaced by, say, rules held on a data set, or even a large IIP, then you really do have a mini-language. A later chapter (Chapter 17) goes into more detail about this whole area. We will also be describing in Chapter 18 an approach to a generalized component of this type which should be able to handle a considerable part of the logic of a business application.

# Chap. VI: First Applications using Precoded Components

> "One of the things I like about AMPS is that there are so many more ways to do a job than with conventional programming" (a programmer at a large Canadian company).

We will start this chapter with the simplest network imaginable - well, actually, a network with one process only is the simplest, but this is equivalent to a conventional program! The simplest network with at least one connection might be a Reader feeding a Writer, as follows:

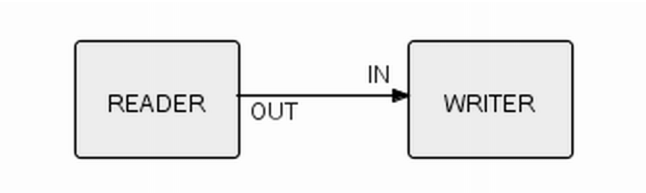



Figure 6.1

This network just copies one file to another, so it is equivalent to the kind of "copy" utility which is provided by just about every operating system. The difference is that FBP lets you combine these utilities into more and more complex functions. Utilities in my experience provide a number of functions, but one always wants something a little different. The functions that they have coded into them are often not the ones one needs. This is quite understandable given the difficulty of predicting what people are going to find useful. One alternative is to combine a bunch of utilities by writing intermediate files out to disk. FBP effectively allows you to combine multiple utility functions without requiring any disk space, or the I/O to read and write from and to disk (so you also use less CPU time, and more importantly, less elapsed time).

Suppose you wanted to combine a "copier" function with a selector, then sort the result before writing it to disk. Just string the functions you want together:

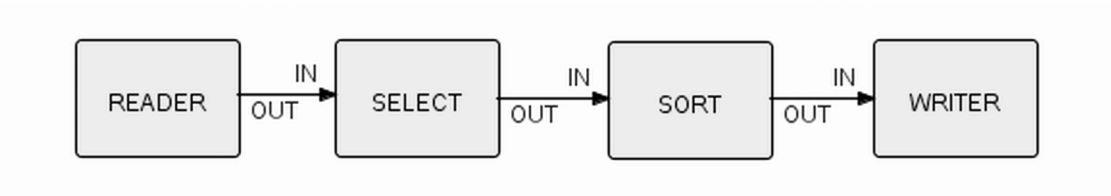



Figure 6.2

Actually, the network doesn't even have to be fully connected. For instance, the following is perfectly valid, and may even be useful!

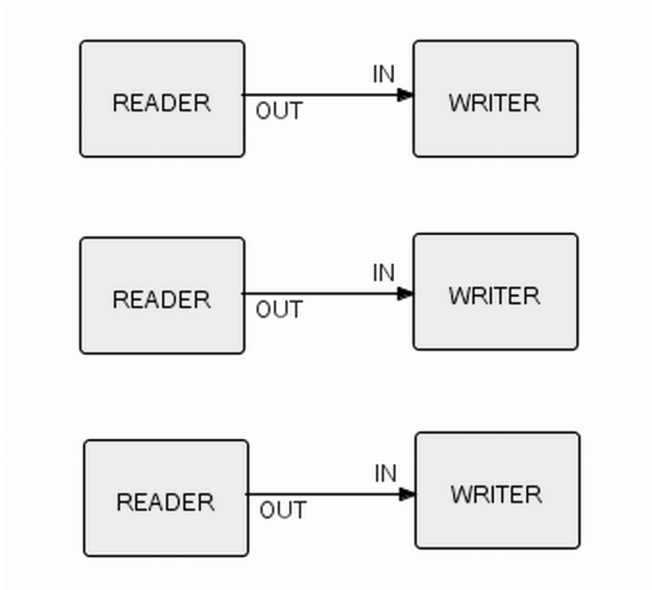



Figure 6.3

I can remember a time when being able to write a program which would simultaneously read cards and write them to tape, read a tape and punch out the records, and do some printing was considered the height of a programmer's ingenuity! With FBP, I discovered all you have to do is specify the connections between six processes as shown in the diagram!

Now, if you think about this diagram as a way to get something done, not to control when it happens, you will realize that the three pairs of processes shown above do not have to run concurrently. The point is that they can if there are adequate resources available, but they don't have to - it doesn't affect their correct functioning. Think of this network as three train-tracks, with a train running on each one. You just care that each train gets to its destination, not when exactly, nor how fast. In business applications it is correct functioning we care about - not usually

the exact timing of events. In the MVS implementations of FBP, the three "tracks" probably will run concurrently because I/O can be overlapped. In THREADS [a PC implementation], which has no I/O overlap yet, they may run sequentially. In none of these cases is the order defined. I think the bottom one will run first, but I'm not sure! Naturally, this makes old-guard programmers very nervous, accustomed as they are to controlling every last detail of when things have to happen. I will keep coming back to this point since it is so important: application development should be concerned with function, not timing control - unless, of course, timing is part of the function, as in some real-time applications. We have to decide what is worth the programmer's attention, and what can safely be left to the machine. Not knowing exactly when things are going to happen turns out to be liberating, rather than disorienting (for most people). But, yes, some programmers will find the transition rather hard!

By the way, when comparing 4GLs and FBP, I have been struck by the fact that you really can do anything in FBP! FBP's power does not come from restricting what programmers can do, but from encapsulating common tasks in reusable components or designs. Since a conventional program is in fact an FBP network consisting of a single process, the programmer is free to ignore all FBP facilities, if s/he wishes, and the result is a conventional program. This may sound glib, but it says to me that we are not taking anything away - we are adding a whole new dimension to the programming process. While some programmers do feel a sense of restriction with FBP, it comes from having to express everything as "black boxes", with well-defined interfaces between them, not from any loss of function.

In the rest of the chapter, we will put together some simple examples using reusable components, but first we should make a catalogue of some of the types of components which an FBP shop will probably have in its collection, a number of which we have already run into. The types which haven't been mentioned above are fairly obvious extensions of what went before. Some of the items in this list should be understood as representing types of component, rather than specific pieces of code. For example, a shop might have two or three Sort modules, with different characteristics.

- sort
- collate
- split
- assign
- replicate
- count
- concatenate
- compare

- generate reports
- read
- write
- transform
- manipulate text (this might be a large group)
- discard

Let's also throw in some components which have proven useful during development and debugging: a "dumper" (which displays hex and character formats) and a line-by-line printer component.

We haven't mentioned "Assign" before. I am going to use this for some examples, so we will go into a little more detail on this type of component. This component (or component type) simply plugs a value into a specified position in each incoming IP, and outputs the modified IPs. It has the same shape as a "filter", and can be drawn as follows:

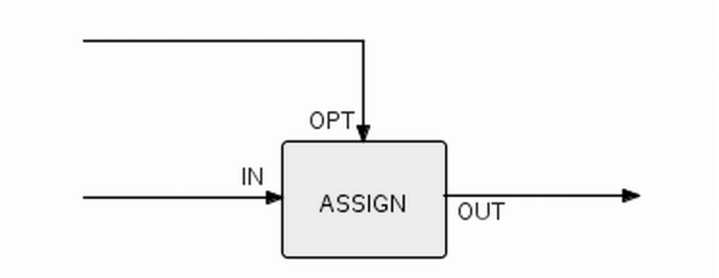



Figure 6.4

where OPT receives the specification of where in the incoming IPs the modification is to take place, and what value is to be put there. For instance, we might design an Assign component which takes option IPs looking like this:

```
3,5,ABCDE
```

This might specify that 'ABCDE' is to be inserted in the 5 characters beginning at offset 3 from the start of each IP. This may seem to be overly simple, but it can be combined with other functions to provide a broad range of function.

By the way, this component illustrates the usefulness of IIPs: if Figure 6.5 is specified in an IIP, you have essentially defined a constant assign with the value defined outside the Assign process, but fixed in the network as a whole. Now, instead, connect the OPT port of Assign to an

upstream process, and you now have a variable assign, where the values can be anything you want, and can be changed whenever you want.

Now let's use Assign to mark IPs coming from different sources. Let's suppose you want to merge three files and do the same processing on all of them, but you also want to be able to separate them again later. You could use Assign to set a "source code" in the IPs from each file, and use a splitter to separate them later.

In an earlier chapter we have used the idea of a two-way selector, looking like this:

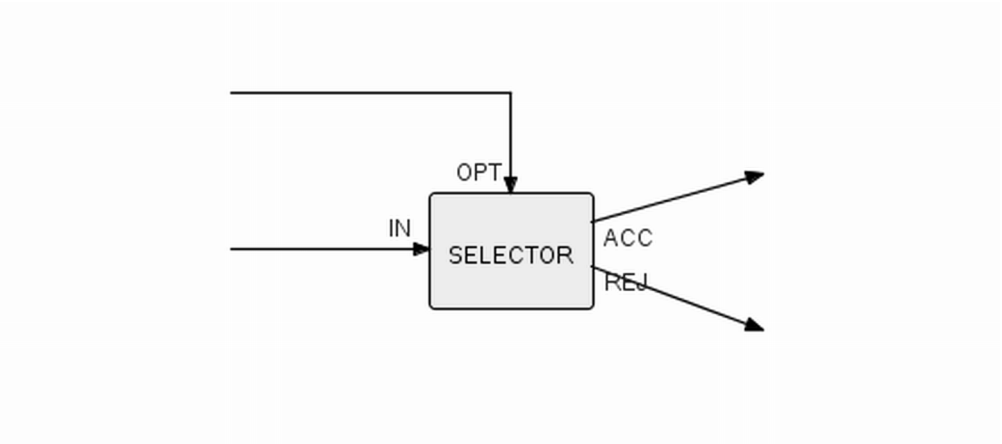



Figure 6.5

Now we can generalize this to an n-way splitter, where, instead of two ports with specific names, we have an array port, which essentially uses a number to designate the actual output connection. Let us show this as follows (using the THREADS numbering convention):

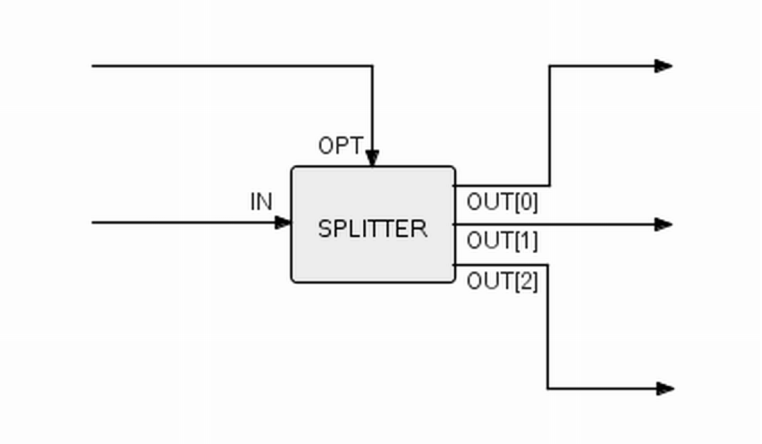



Figure 6.6

Combining the Assign components and a splitter, we could implement the above example of merging three files as follows:

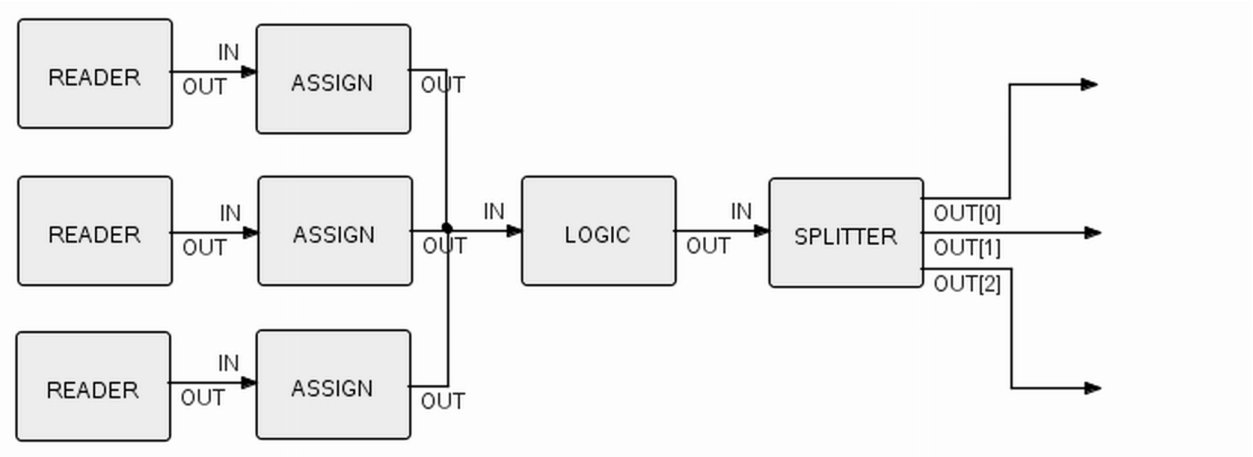



Figure 6.7

Splitters have to be parametrized by specifying a field length and offset, and a series of possible values. The above splitter might therefore be parametrized as follows:

```
54,1,'A','B','C'
```

assuming the codes A, B and C were inserted into the IPs coming from the three readers, respectively.

Let us construct a more complex example based on the provinces of Canada. Let's say we have a file of records with province codes in them. We want to arrange them by time zone, so that we can print them out and have a courier deliver them in time for start of business. The easternmost point in Canada is 4 1/2 hours ahead of the westernmost point, so most big Canadian companies have to wrestle with time zone problems.

Here we will also use a splitter to split the stream of IPs by province. Once the "province" streams have been split out, we could use Assign to insert appropriate codes into the different IP. This splitter might have an option IP looking like this:

```
6,2,'ON','QU','MA','AL',...
```

This will be read as follows: check the 2 characters starting at offset 6 from the incoming IP; if it is ON, route the IP to OUT[0]; if QU, to OUT[1]; if MA, to OUT[2]; etc. Of course, in both of the above cases, it would be much more friendly to be able to use field names, rather than offsets

and lengths. We will talk about this idea in the chapter on Descriptors.

One other question we have to answer is: what does the Selector do if the incoming IP does not match any of the specified patterns? In DFDM, the standard splitter simply sent unmatched IPs to array element 'n+1', where 'n' equals the number of possible values given (DFDM used 1-based indexing). Another possibility might be to send unmatched IPs to a separate named port.

The application might therefore look like this (I'll just show two Assign processes, and assume all IPs find a match):

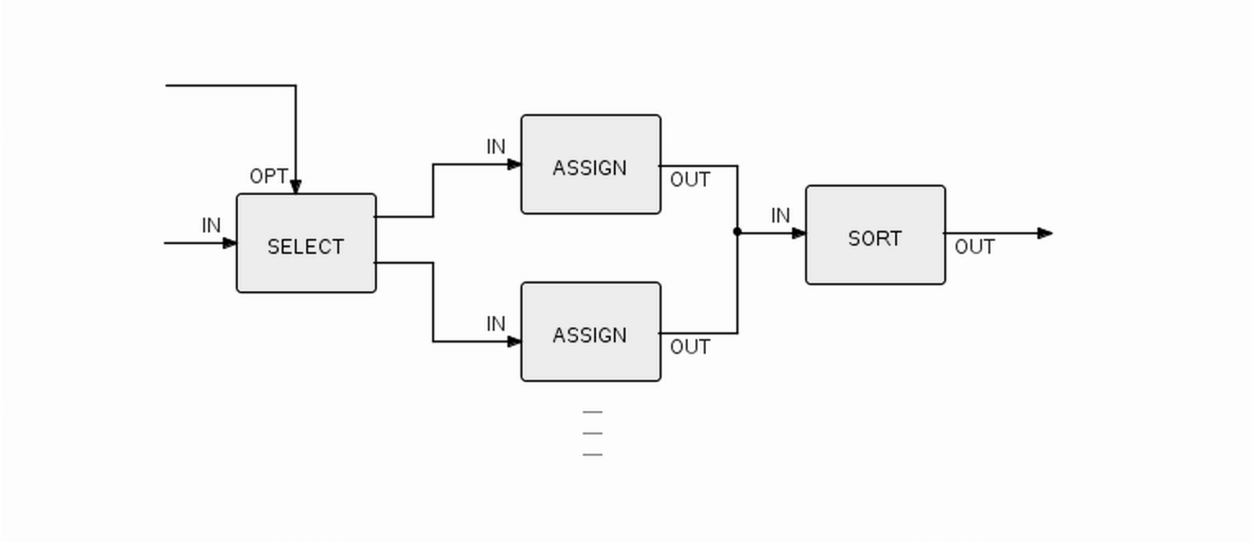



Figure 6.8

The Assigns can insert a code which ascends as one goes from east to west. Because Sort is going to rearrange all the data, we can feed all the modified records into one port on the sort process. This also avoids the possibility of deadlock (I'll be talking more about why this should be so in a later chapter).

I didn't show the option ports on the Assigns, but they will be necessary to specify what codes should be inserted, and where.

Now let's add the logic to handle unmatched IPs. Since they should probably be reported to a human, we will add a printer component, and used the named port technique, as follows:

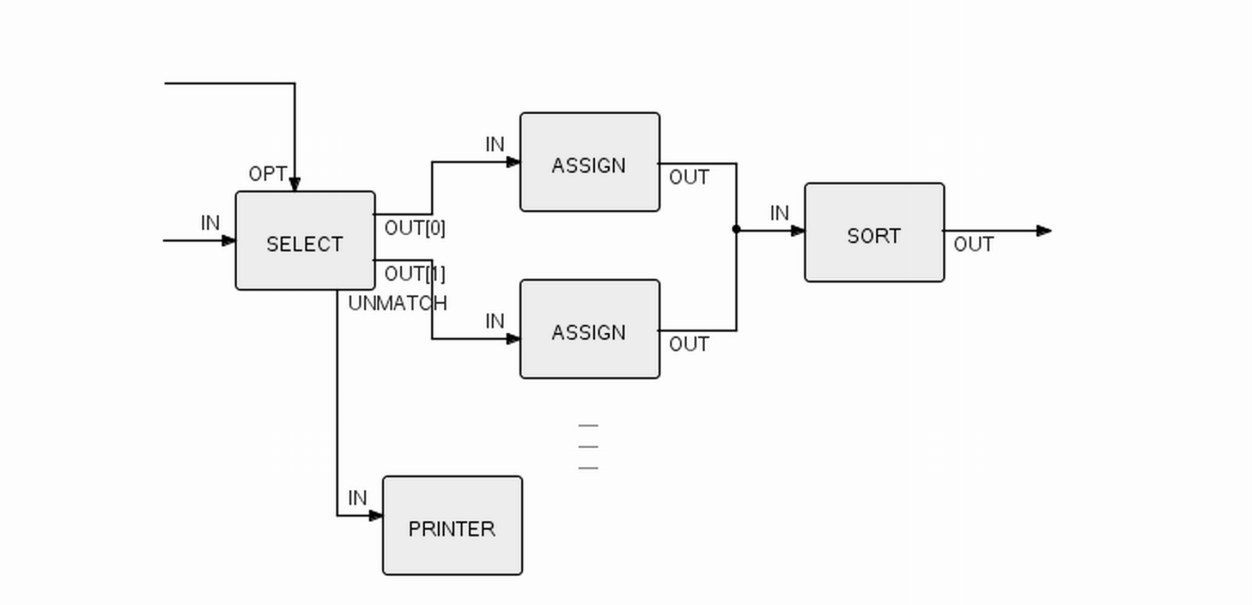



Figure 6.9

At this point we will simply have a list of unmatched IPs streaming out onto a print file. You will probably want to put out an explanatory title, and do some formatting. You will see later how to do this. For now, let's just say that you will probably need to change this network by inserting one or more processes where I have shown PRINTER above, e.g.:

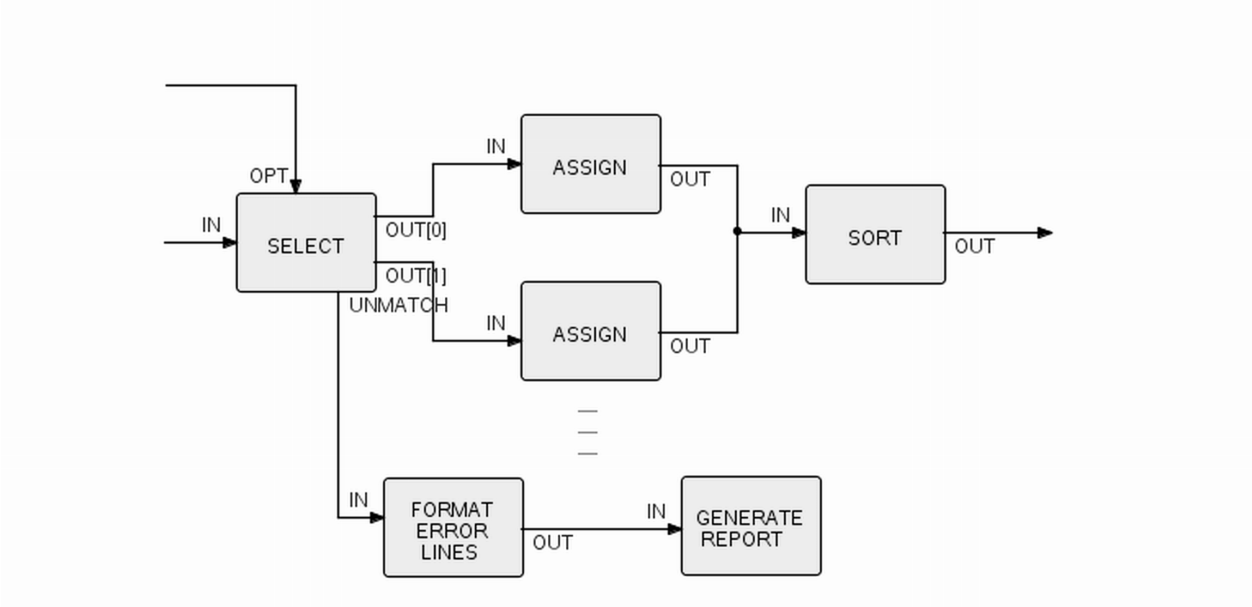



Figure 6.10

Let us suppose we now have our network working and doing what it is supposed to do - we have to ask the question: is this network "industrial strength"? There is a definite temptation, once something is working, to feel the job is finished. In fact, it is quite acceptable to use a program like the one shown above for a once-off utility type of application, or for a temporary bridge between two applications. But there is a fundamental question which the designer must answer, and that is (in this application), how long are there going to be exactly 13 provinces and territories in Canada? No, you did not suddenly jump into a book on Canadian politics! We have made our program structure reflect a part of the structure of the outside world, and we have to decide how comfortable we feel with this dependency. Yes, there will always be programmers, and we can always change this program... provided we can find it! Nobody can make the decision for you, but I would suggest that if this program may have to survive more than a few years, you might want to consider structuring your code to use a separately compiled table or data base, which can pull together all the attributes of provinces of interest to your application. Your application might then look like this (if you generalize SELECT to mean "determine which province it is", and ASSIGN to mean "insert the time zone code for each province"):

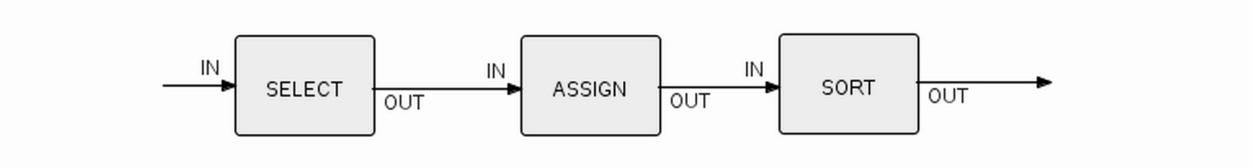



Figure 6.11

Note that this diagram has become simpler and the components more complex. It has the same general structure as the preceding diagram, but the shape does not reflect a (possibly changeable) political structure. Another approach might be to amalgamate the SELECT and ASSIGN components shown above, using either special purpose code or a generalized transformer module.

A more subtle generalization might be to keep the parallelism, but not tie it to individual provinces at network specification time. Let us decide there will never be more than, say, 24 provinces, so we will extend the earlier diagram to have 24 ASSIGN processes. SELECT will have 24 port elements on its output port, connected to the ASSIGNs. Now, since both ASSIGN and SELECT are option-driven, we can obtain their parameters from a file (being read by a Reader) or a table (using a Table Look-up component) and send them to their OPT ports. The possibilities are endless! The important decisions are not "how do I get this working?", but "what solution will give me the best balance between performance and maintainability?". When I was teaching FBP concepts, I used to tell my students that the machine will accept almost anything you throw at it - the real challenge in programming is to make your program comprehensible by humans (whether they are other people looking at your code years later, or your own self one week later)! This is far more of a challenge than simply getting the code working.

There is yet another way to look at this example: what we are really doing (in the last diagram) is converting one code to another under control of a table. One of the generalized component types that we found most useful was one (or several) table look-up components. The table could be held as a load module (in MVS), in which case it would have to be maintained by the programming department, but a better technique is to hold it on a file. The table look-up component will then look like this:

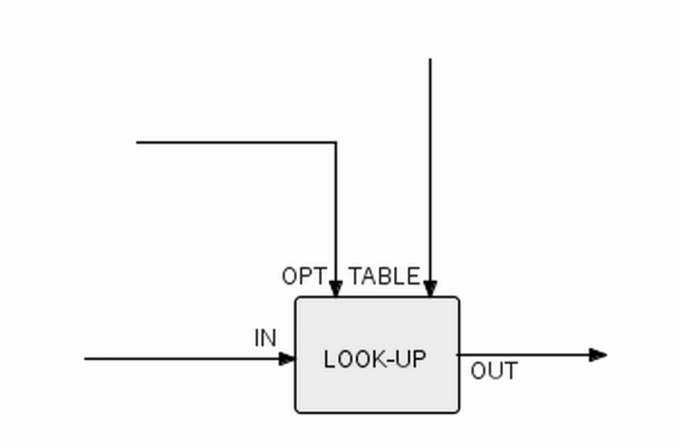



Figure 6.12

What will happen here is that at start-up time the component will read all the IPs from the port named TABLE and build a table in storage. It then starts a process of receiving IPs from IN, looking up a specified field in the table, inserting the found value into the IPs, and then sending them to OUT. Of course, this needs to be parametrized - probably we will need to specify the offset and length of the search field in the incoming IPs, and the offset and length of the field which is to be inserted. We will need a Reader to bring in the table IPs from a file, so the resultant network will look like this (partially):

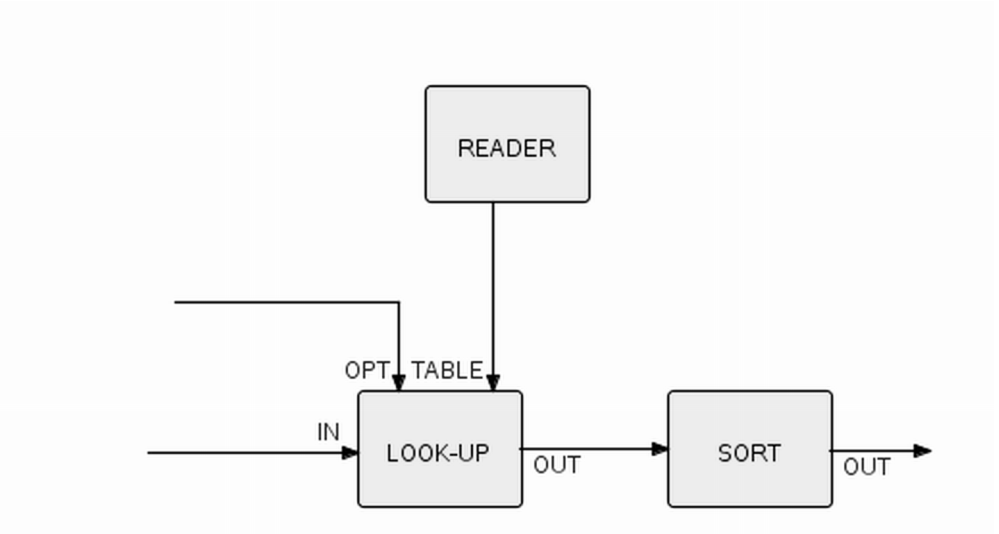


Figure 6.13

Table look-up components have been found to be very useful in the various dialects of FBP. In DFDM, the system was distributed with two "off the shelf" table look-up components. One of them was much like the one we've just described. However, the other one gives you some idea of what can be provided in the form of reusable code, with a little more imagination! It's pretty complex, but it is a "black box" piece of reusable code, and I am showing it just to indicate what can be done with a single component. Of course it violates some of the rules we gave above, so maybe this should sound some warning bells! However, I believe you will agree that it is still a "stream" process, rather than just a complex algorithm. Basically, it does table look-ups on a table which is being refreshed from some kind of backing store - it doesn't care what kind, so long as the other process conforms to a certain protocol. It therefore acts as both a table look-up component and as a buffering device.

Here is the picture:

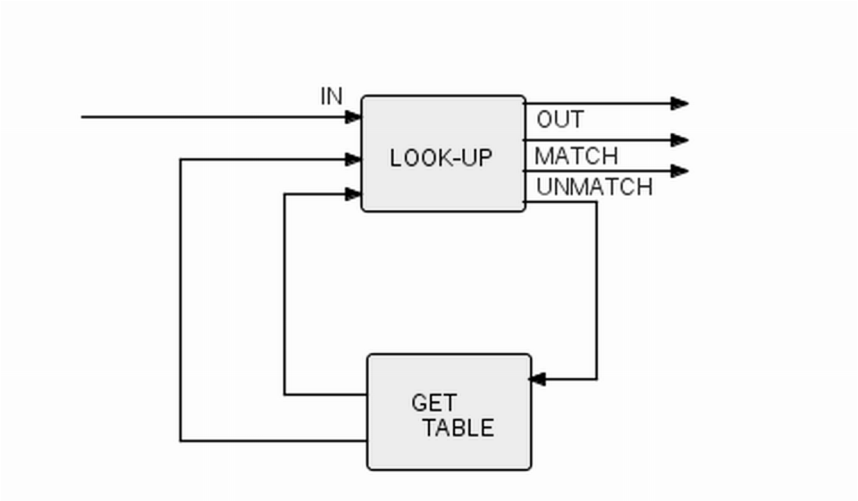



Figure 6.14

This look-up component always works with another process (in this case called *GET TABLE*) which is used to access a direct access file or data base. These two components work together as follows:

- the top component builds the table, which starts out empty
- as each search request comes in, it is checked against the table; if a match is found, the search request goes out of OUT and the table entry out of MATCH

- if a match is not found, the search request is sent to the other process, which will either find it or not; if it finds it, the search request and the found table entry are sent back to the top component, to be respectively sent to OUT and added to the table
- if it does not find it, a zero-length IP is sent to the top component, together with the search request; the zero-length IP tells it not to add an entry to the table, and the search request is sent to UNMATCH

This may seem complicated, but once set up, it is very easy to use, and any process can occupy the "bottom" position, provided that it behaves as described. This DFDM component (the "top" component) only needed 4 parameters, of which one was the (optional) maximum number of entries in the table. If this limit was specified, and was reached, entries would start to be dropped off the table in FIFO sequence. Another possibility could have been to use an LRU (Least Recently Used) sequence.

This chapter has tried to give some idea of what can be done using only reusable components. Of course, the number of these is going to grow steadily, and, especially if you have made some of the organizational changes we suggested in an earlier chapter, you may wind up with a sizable number of useful, reusable tools. We don't know what this number is, but my guess is that it will taper off in the low hundreds. The amount these are used will follow a curve with the following general shape:

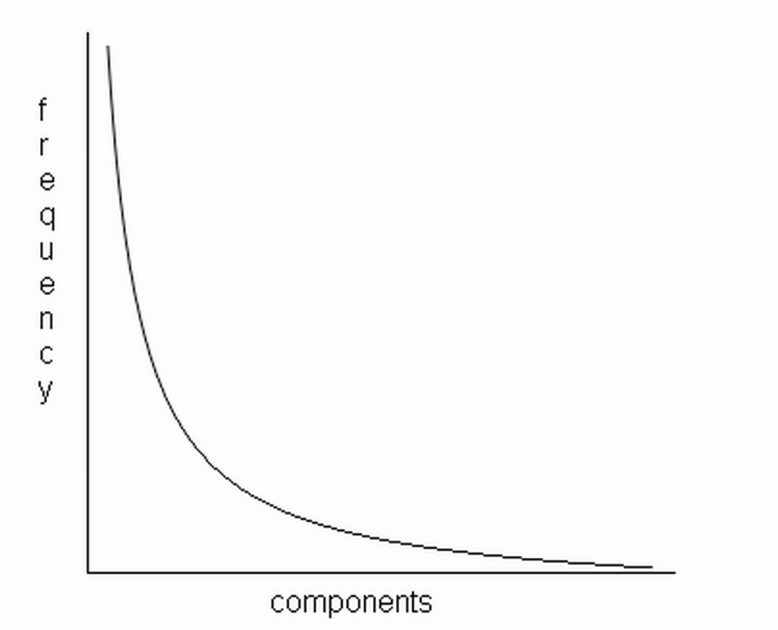



Figure 6.15

where the less frequently used components may be used only in a few applications. Given this kind of distribution, you may wonder whether it is worth maintaining the low-frequency components in a support department, or whether they should be made the property of the using departments. The answer will depend on whether specialized knowledge is encapsulated in them, whether you are trying to sell them outside the company, and so on.

This kind of rarely used component has similarities with what we have called "custom" components. Until all code is "off the shelf" (if that ever happens), there will be a need for customers to write their own components. This is not as easy as just hooking reusable components together, but it is also pretty straightforward. In the next chapter, I will talk about the idea of composite components, and then in the following chapter start to discuss some concepts for building systems by combining "off the shelf" and "custom" components.

# Chap. VII: Composite Components

In Chapter 3 (Concepts), we talked about hierarchic structures of substreams - there is obviously another type of hierarchic structure in FBP, which has been alluded to earlier: the hierarchical relationship between components. Although the processes in an FBP application are all little main lines, cooperating at the same level to perform a job of work, it is easier to *build* the total structure hierarchically, with a "top" structure comprising two or more processes, most of which will be implemented using composite components. Each composite component in turn "explodes" into two or more processes, and so on, until you reach a level where you can take advantage of existing components, or you decide that you are going to write your own elementary components, rather than continuing the explosion process. Apart from the emphasis on using preexisting components, this is essentially the stepwise decomposition methodology of Structured Analysis.

The application of this approach to FBP is largely due to the late Wayne Stevens, arising from his work on Structured Programming and application development methodologies in general. From a hierarchy point of view, a running program built using AMPS (the first FBP dialect) was "flat" - it had no hierarchic structure at all. People drew application networks on *big* sheets of drawing paper, and stuck them up on their cubicle walls. These drawings would then gradually accumulate an overlay of comments and remarks as the developers added descriptions of data streams, parametrization, DDnames, etc. When the time came to implement the networks, the developer simply converted the drawing into a list of macro calls.

Wayne realized that a better way to develop these networks was to use the decomposition techniques of Structured Analysis, but that, unlike conventional programming, *there was no change in viewpoint as you moved from design to implementation*. In conventional program development there is a "gap" between the data flow approach used during design and the control flow viewpoint required during programming, which is extraordinarily difficult to get across. When building DFDM we therefore provided a way for developers to grow their systems by stepwise decomposition, but at structure build time the hierarchy got "flattened" into the

conventional AMPS-type network. This approach turned out to be very successful - I alluded earlier to a colleague who built a 200-node network without once drawing a picture of a network!

So DFDM networks can be built up hierarchically, but are flat at run-time. This approach lets developers build up their applications layer by layer. As we have shown in some of the examples, you could also take a simple component and replace it by a subnet - this gives an additional dimension of expandability. In DFDM we have seen that subnets can be stored on reusable libraries and reused. However, they must be stored in interpretable form, so their internal structure is visible (although they are built out of black boxes). It would be even better for someone marketing an application to be able to store some or all of an application as "black box" subnets, so that the customer cannot see inside the subnets. Remember that DFDM has the ability to take an interpretable network and convert it into a directly executable load module. It should be possible to do this with a subnet as well, so that a *part* of an application is held as a black box as well. The original motivation for doing this, however, was performance of interactive systems.

Consider an application comprising 200 processes - when you link all the code together into a single module, the result is a pretty big load module. The particular application mentioned above ran under IMS/DC, and the network was executed once for each transaction. However, when we started measuring performance, we found that, on each pass through the network, even though IMS loads in the whole network for most transactions, only about 1/3 of the processes actually got executed for a given transaction. We wondered if we could just build a framework for an interactive application and load in the required chunks of logic dynamically. This would have the advantage that it would take less time to load in the application on each transaction, because the individual load modules would be smaller, thus improving response time. Also, on each transaction you would only need the framework and the particular dynamic chunk involved, so the framework could actually be made resident, improving response time even more. It should also make it easier to expand the application and even change it on the fly. The problem was: how do you run a network which modifies itself dynamically without losing track of your data?

For a long time, I resisted this idea as I had a vision of a complex subway system like the London underground, but with the added complexity that stations would be appearing and disappearing at random. How would it feel to be a passenger in such a system?! I had experimented with loading in individual components dynamically under AMPS: I was able to read in or load a piece of code, treating it as pure data, and let it travel through a network, until it arrived at a "blank" process (one which was connected to other processes but had no code assigned to it - a sort of "tabula rasa" process), at which point it would get executed, so I felt that dynamic process modification could work under controlled circumstances. Wayne Stevens had also proposed a particular case of dynamic network modification which is simpler than what we eventually landed up with, but we never got around to trying it out: his image was of engineers repairing a dam. The water has to keep flowing, so the engineers divert the water through a secondary channel. After the dam has been repaired, the water flow can be restored to its original channel. This seems like it would be a

good way to do maintenance on an FBP system which has to keep running 24 hours a day, like a banking system.

I myself came up with a different and somewhat more complicated approach, which however was also safe and manageable, and which also solved the problem of load module size I described above. The trick I discovered was to have a "mother" process load a subnet (in compiled and linkedited form), start up the processes in the subnet, and then go to sleep until all of her "daughter" processes had terminated. At subnet start time, the daughters are counted, so the subnet is finished when the count of active daughters has gone down to zero. While the mother is sleeping, some of the daughter processes can be given control of their mother's input and output ports. There will be no conflict over who has control of the ports, as the mother and the daughters are never awake at the same time.

We called these subnets in DFDM *dynamic subnets*. The "mother" was a generalized component called the Subnet Manager, which continuously iterated through the following logic:

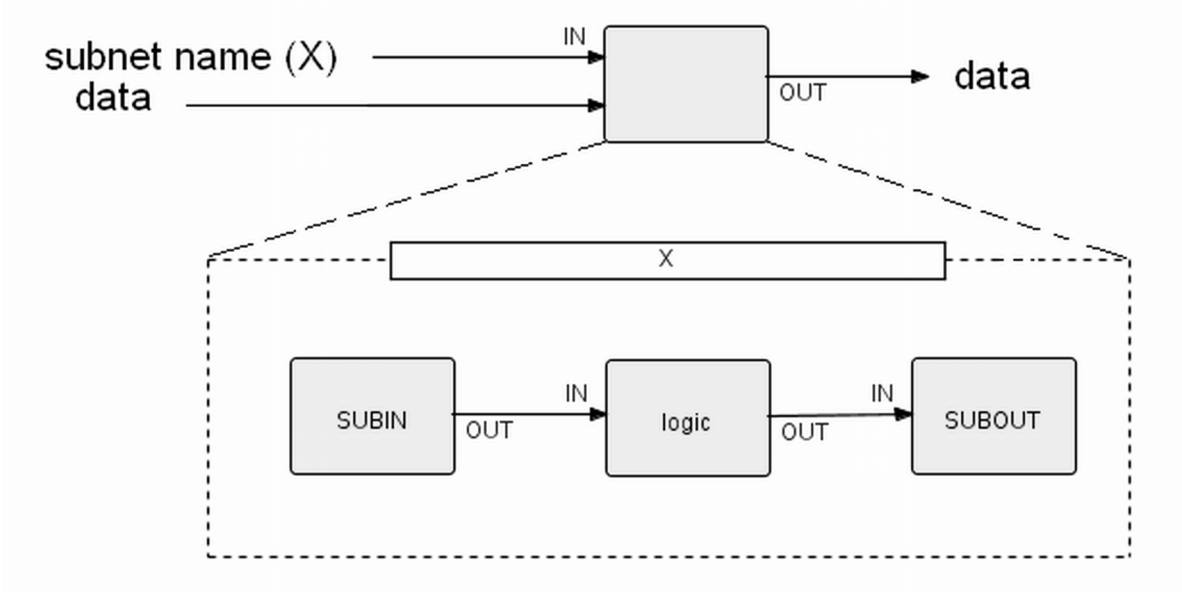
- receive the name of a subnet in an IP,
- load the subnet load module,
- "stitch" it into the main network,
- start it up and go to sleep,
- wake up when all the daughter processes have terminated,
- dispose of the subnet load module and repeat these steps

In addition to the Subnet Manager, we added special precoded components (Subnet In and Subnet Out) which were used for input and output handling by the subnet. Here is a picture of a very simple dynamic subnet (with one input data port and one output port):

Figure 7.1

When 'X' is given control, it behaves just like a mini-application: technically SUBIN has no input ports, so it gets initiated. The other two processes have input ports, so they will not be initiated until data arrives. SUBIN and SUBOUT have the ability to use their mother's input and output ports respectively. They have to be separate processes as they have to be independently suspendable. Thus, if mother had two input data ports, the subnet would have to have two SUBIN processes to handle them.

Now we noticed a strange thing: normally, once a process terminates, it can never be started again. We saw that the Subnet Manager had to have the unique ability of being able to restart terminated processes. This was the only function in DFDM which had this ability, and it was in a very special off-the-shelf component.

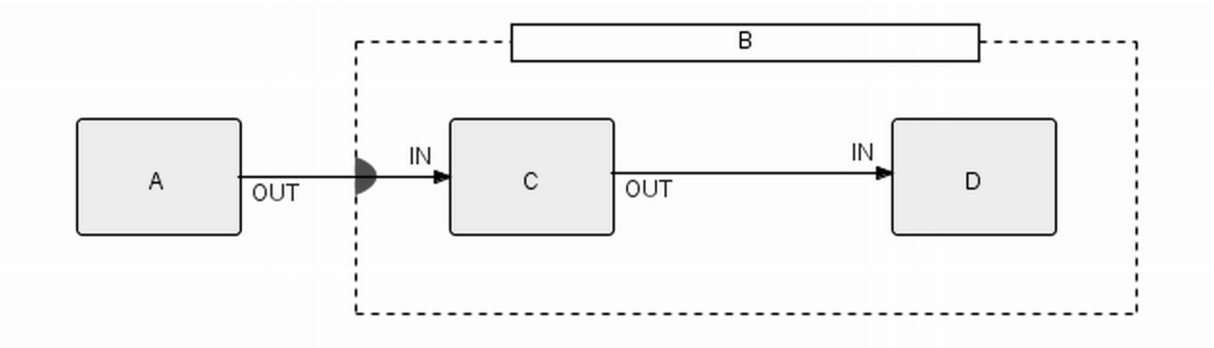



In the work which followed DFDM (referred to above under the name FPE), we realized that these characteristics of dynamic subnets could be extended to static networks as well. We moved this ability into the infrastructure (removing the subnet names port), so that all subnets had a built-in monitoring process. This approach naturally coordinated the hierarchy of processes with the stream hierarchy. In addition, since the monitoring process's other job is to stitch the composite into the main network, we could now have "black box" composite components. This facility would allow subnets to be packaged as separate load modules for distribution, which could later be linked with other components by a developer to form the full application network. This seems very attractive, as a software manufacturer will be able to sell a composite component without having to reveal its internals (as would be required by DFDM)! We will also see later (in Chapter 20) that these concepts give us an intuitively straight-forward way of implementing fairly complex situations such as checkpointing long-running applications.

We mentioned above how, in dynamic subnets, we have a "mother" process which monitors the execution of its daughter processes, and can "revive" them after they have all closed down. However, since the subnet cannot close down until all IPs have been received at all input ports, what would be the point of ever waking up the subnet again? Well, if that was all we could do, it would just be a performance improvement. However, we came up with an idea which we thought dovetailed in pretty neatly. Why not put markers in the data stream, such that the internal subnet thinks it is seeing end of data, and will terminate, but in fact there is more data to come, so it will be revived? We did this by adding an option to composite components called *substream sensitivity*. This was implemented in DFDM for dynamic subnets, and in FPE as an option on all

composite components. Sunbstream-sensitive composites essentially keep track of the bracket nesting levels at each of their input ports, and whenever this level drops to zero for a given port, the port involved is closed, resulting in an "end of data" indication next time the daughter process does a receive from that port. Essentially they make substreams on the outside look like streams on the inside. Since you can nest subnets within subnets, each level of nesting strips off one level of bracketing (call it "the application onion").

Let us start with one input data port only. Suppose we have a "substream-sensitive" composite B, which contains C and D, as follows:




```
The point shown with a solid semicircle is a substream-sensitive port
on B.

[This is sort of a shorthand - the solid semicircle will be implemented as an
"external port", which has no real input ports, but can access the input port
of the subnet.]

Figure 7.2
```

Suppose that A generates a stream as follows:

```
(a b c d e f)  (g h i)  (j k l m)...
```

reading from left to right. Then C is going to see IPs *a*, *b*, *c*, *d*, *e* and *f,* and then end of data. At end of data it terminates, as it has no upstream processes at the same level *within its enclosing composite*, enabling D to terminate also. If D was a Writer, it would then close the file it was writing to. However, we know (although the subnet doesn't) that C and D are not permanently "dead" - when the next open bracket arrives at B's input port, they will both be revived. As far as C and D are concerned, IPs *a* through *f* constitute a complete "application", but, as far as B is concerned, each substream, e.g. *a* through *f* results in a single activation of the internal subnet.

What happened to the brackets? Well, we could add a process to remove the enclosing brackets

of a substream, but it seemed a good idea to add the ability to substream-sensitive ports to drop the brackets if the designer wants. However, you may not always want this: for instance, if D was outputting IPs to B's output port, you might have to be able to put the brackets back on again.

In this example, you can see the insides of a composite working like a complete application within each activation of a composite. The power of this concept is that you can match levels of composite component to levels of nesting of substreams. So, *substream* structure can be related to *subnet* structure. I would like to record here the fact that this very powerful idea came from Herman van Goolen, of IBM Netherlands, and I feel it is very elegant. You can probably now see why we use brackets both to delimit substreams and as the delimiters which substream-sensitive composites respond to.

If we have more than one input port on a substream-sensitive composite component, as described above, our composite will process one substream from each input port successively until all the input ports are exhausted. Processing of these input streams will therefore be synchronized at the substream level. Other kinds of synchronization are also possible, of course, but we have found this type to be the most generally useful. It also ties in nicely with the requirements of Checkpointing (see Chapter 20).

# Chap. VIII: Building Components & Some More Simple Applications

I am now going to describe some simple applications mixing reusable components and custom ones. We start with a fairly simple text processing application to make a few points about the design of applications in FBP. This is a classical programming problem, originally described by Peter Naur, commonly known as the "Telegram problem". This consists of a simple task, namely to write a program which accepts lines of text and generates output lines of a different length, without splitting any of the words in the text (we assume no word is longer than the size of the output lines). This turns out to be surprisingly hard to do in conventional programming, and therefore is often used as an example in courses on conventional programming. Unless the student realizes that neither the input nor the output logic should be the main line, but the main line has to be a separate piece of code, whose main job is to process a word at a time, the student finds him or herself getting snarled in a lot of confused logic. In FBP, it is much more obvious how to approach the problem, for the following reasons:

- words are mentioned explicitly in the description of the problem
- since we have to select our IPs between each pair of processes, it is reasonable for the designer to treat words as IPs somewhere in the implementation of the problem. It would actually be counter-intuitive to deliberately avoid turning words into IPs, given the problem description
- there is no main line, so the student is not tempted to turn one of the other functions into the main line.

Let us dig into the coding of this problem more deeply. We should have IPs represent words somewhere in the application. You will have realized also that we should have a Read Sequential on the left of the network, reading input records from a file, and a Write Sequential writing the new records onto an output file. Here is a partial network:

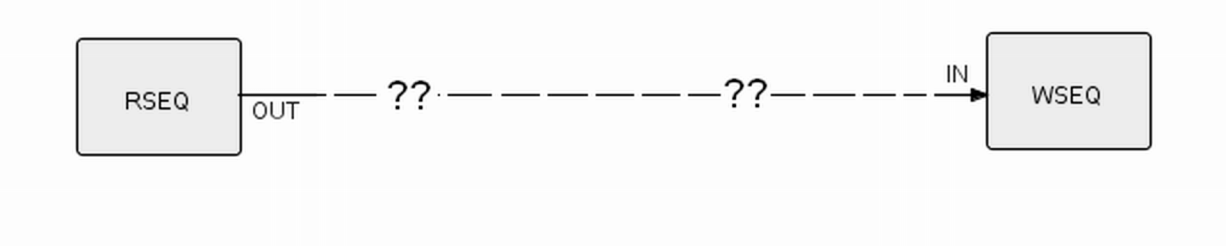



Figure 8.1

Now the output of Read Sequential and the input of Write Sequential both consist of streams of IPs containing words and blank space, so it seems reasonable that what we need, at minimum, is a component to decompose records into words and a matching one to recompose words back into records. Given the problem as defined, I do not see a need for any more components, but I want to stress at this point that there is no single right answer. Remember ROI? What you select as your basic black boxes depends on how much they are going to be used versus how much it costs to create them.

Let us add our two new components into the picture:

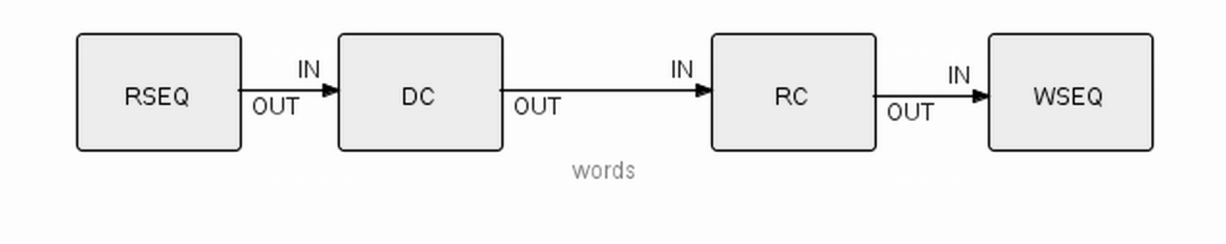



Figure 8.2

Now we have another matched pair of components - in the diagram I have labelled them DC (for DeCompose) and RC for ReCompose). Components can always find out the actual size of any IP, so we do not have to provide the size of the incoming IPs to DC as a parameter. However, RC cannot know what size of IPs we want it to create, so this size must be passed as a parameter to its OPTIONS port (I didn't have to call it that - but it is good a name as any). So let's show an options IIP on RC. RSEQ and WSEQ will also need to know the identifiers of the files they are working with, so our diagram now looks like this:

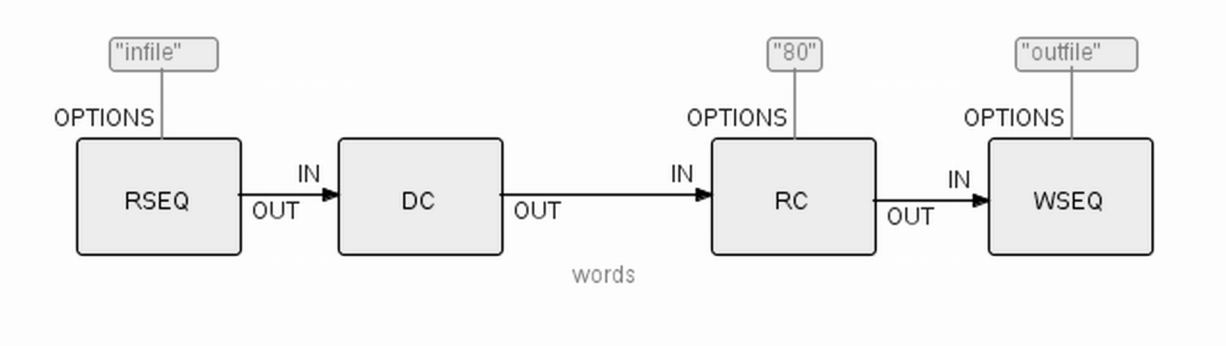



Figure 8.3

For completeness, I will give some possible pseudo-code for DC and RC. Remember that, once you have written and tested DC and RC, you have them forever. So it is worth the effort to get them "perfect" (as close to perfect as software ever gets!). One of the great advantages of FBP is that you can simply insert a Display process on any connection, e.g. between DC and RC, to see if the IPs passing across that connection are correct. So testing is very easy.

In what follows, I have assumed (but not shown in the diagram) that the input port of both components is called IN and the output port is called OUT.

```
DC (Decompose into Words):
        define switch "in word" - initially off
        receive from IN using a
        do while receive has not reached end of data
                start at beginning of input IP
                do while not yet end of input IP
                        if "in word" off and current char blank OR
                                "in-word" on and current char non-blank
                                do nothing
                        else
                                change state of "in word"
                                if "in word" now on
                                        save pointer
                                else
                                        create word IP of length =
                                          (current pointer - saved pointer)
                                        copy that number of chars to new IP
                                        send created IP to OUT
                                endif
                        endif
                        look at next character
```

```
                enddo
                drop a
                receive from IN using a
        enddo
```

**RC (Recompose Words into Records):**

```
        receive word IP from IN using a
        do while receive has not reached end of data
                create output IP and set it to all blanks
                start at beginning of output IP
                do forever
                        if received word will not fit into output IP
                                send output IP to OUT
                                create output IP and set it to all blanks
                                start at beginning of output IP
                        endif
                        move contents of word IP into output IP
                        if there is room for 1 more character
                                move in single blank
                        endif
                        drop a
                        receive word IP from IN using a
                        if receive reached end of data,
                                leave innermost loop
                        endif
                enddo
                send output IP to OUT
        enddo
```

Maybe this logic can be simplified, but a component does not have to be simple on the inside - it should be simple on the outside, and above all it must work reliably! This point really illustrates a fundamental difference between conventional programming and FBP: I have just shown some pseudocode, and you may be feeling that we are back to conventional programming. However, another way of putting what I am trying to say is that, because we can program, it does not mean that we should. Most conventional programming, and even some of the new Object-Oriented approaches, still stress the production of new code. Many reuse approaches are based on finding what source code is available, and reusing it. Because code is such a malleable medium, and does not have an inherent component structure, we are always creating new stuff and we forget that the results of our work may live long after us and that there is a cost to maintaining them, documenting and managing them. How many times have we heard, "It's less trouble to write my own than to find out what's out there"? In FBP, the orientation is the exact reverse: use what's out there, and only build new if you can justify the effort in terms of ROI. This is where experience

becomes valuable: after you have done the same job many times, you know whether other people will find components like DC and RC useful. If you know they won't, find another way to do the job!

Now we have a matched pair of useful components, but, of course, you don't have to use them together all the time. Let's suppose we simply want to count the number of words in a piece of text. We have already mentioned the Count component in the Concepts chapter - it simply counts all its incoming IPs and generates a count IP at the end. It has an option which is substream-sensitive (it generates one count IP for each incoming substream), but for now we will use it in its most basic form. In this form, it simply sends the count it has just calculated out via one output port, while the original incoming IPs are sent out of another one if it is connected (this is an example of an optional port). This type of component is sometimes called a "reference" component, meaning that the original input IPs are passed through unchanged, while some derived information is sent out of another output port. So the resulting structure will look something like this (we won't bother to connect up Count's optional output):

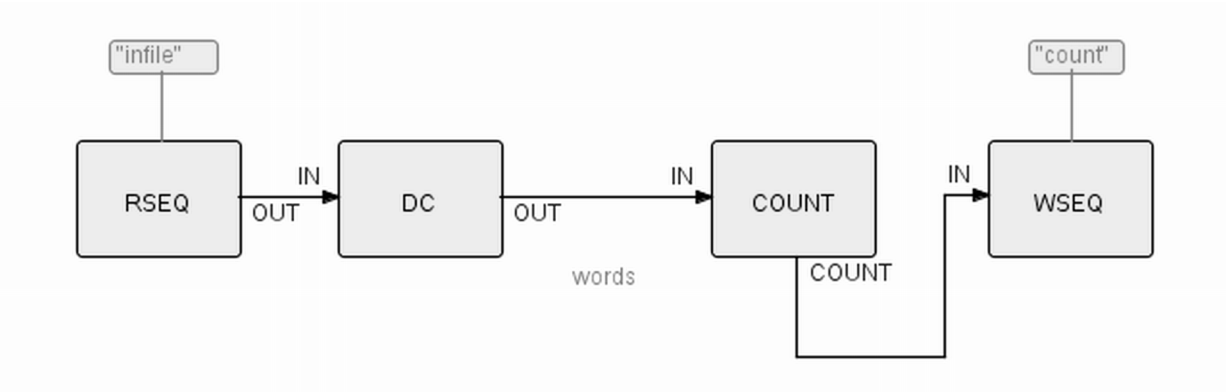



Figure 8.4

We can keep on adding or changing processes indefinitely! These changes may result from changing requirements, new requirements or simply the realization that you can use a component that was developed for one application on another. We talked in the chapter on Reuse about some of the principles behind designing components for reuse.

As another more elaborate example, instead of counting the words, we could decide to sort them, alphabetically or by length. Once we have the words sorted alphabetically, it might be nice to be able to insert fancy heading letters between groups of words starting with the same letters, like some dictionaries.... Of course, once we have sorted them, we should eliminate duplicates. The resulting diagram would then look like this (partially):

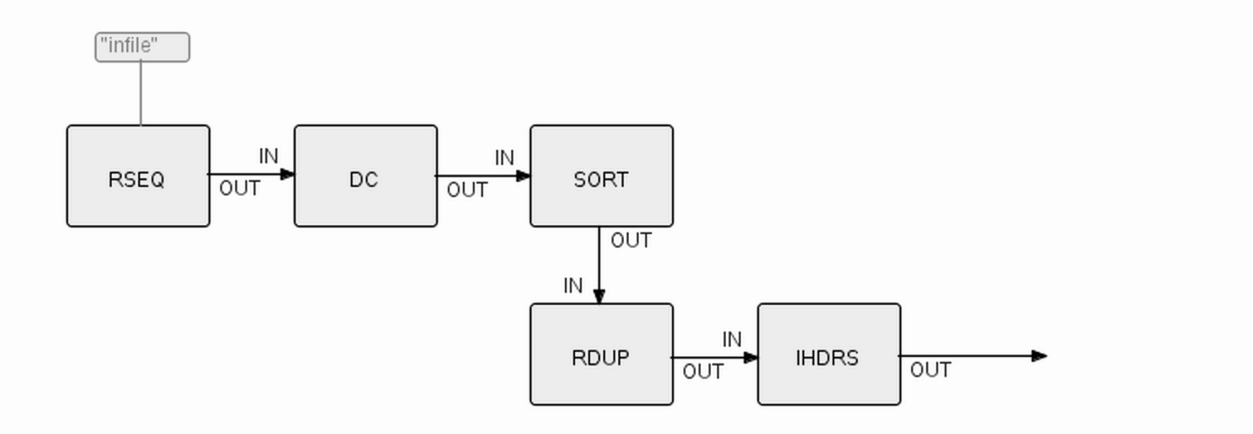



Figure 8.5

where RDUP means "Remove Duplicates" and IHDRS means "Insert Header Letters".

It will come as no surprise that text processing applications have been very productive of generalized components. This is also the application area where the UNIX™ system has proven very productive. The UNIX pipe mechanism is very similar to FBP's data streams, except that UNIX communication is based on using streams of characters, whereas FBP's communication is by means of structured IPs.

An excellent example of this kind of text-processing application is P.R. Ewing's publication a few years ago of a Concordance to the Ukrainian Bible (1988) - this was programmed using DFDM, and Philip found DFDM to be very well suited to this type of work. He has recently completed a Biblical Concordance for Xhosa (one of the languages of South Africa), using the PC-based THREADS software. He reports that, after spending between 100 and 150 hours trying to develop this Concordance using conventional (non-FBP) software, he eventually had to abandon it unfinished. With THREADS he was able to produce a completed Concordance with about 40 hours of work (of course this does not include the time needed to input the Bible text). He told me that, as far as he was concerned, the big advantage of FBP is the fact that it simplifies the complexity of an application, and 40 hours vs. more than 100 hours certainly seems to bear this out.

We have used the Sort function as a component a number of times in the foregoing examples. In conventional programming, Sort is usually packaged as a stand-alone utility, although various exits are provided to allow its behaviour to be modified. What are the advantages of packaging it as an FBP component? People very often have the reaction that Sort cannot be a good FBP component because it is too "synchronous" - all the input has to be read in before any of it can be sorted; then the sort proper takes place; then all the sorted records are output by a final merge

phase. However, we have found a stream-to-stream sort to be a very effective FBP component for the following reasons:

- Performance: a Sort which is run as a separate job step has to use files for its input and output, and the control fields have to be in the same place in every input record. If we provide Sort as a stream-to-stream component, then data IPs which are to be sorted no longer have to be written to a file first (and do not have to be retrieved from a file afterwards), but can simply be sent across a connection to the Sort, which in turn sends them on to the next process when it is finished, resulting in a considerable savings in I/O overhead. Actually, the "central" sort phase is the only part of the sort which cannot be overlapped with other processes.
- Flexibility of positioning control fields: if the control fields are not in a standard place for all the input IPs, you can simply insert a transform process upstream of the Sort to make the key fields line up.
- Eliminating unnecessary sorting: sometimes, some of the IPs are known to be already sorted, in which case they can bypass the Sort, and be recombined with the sorted IPs later. This is often not practical when the Sort is a separate job step.
- Improved sorting techniques: if you know something about the characteristics of your keys, you may be able to build more complex networks which perform better than a straight sort. For instance, if you are sorting on a name field, it might make sense to split the data 26 ways, sort each stream independently, then merge them back together - I don't say it will definitely, but you can try it out. The sort process can thus be implemented with other components or subnets for purposes of experimentation.

Although some sorts are faster, a good rule of thumb is that the running time of most sorts is proportional to n.logn, where 'n' is the number of records. Since this is a non-linear relationship, it may be more efficient to split your sort into several separate ones.

Here is a picture of a Sort with some IPs going around it, and with sort tags being generated on the fly by an upstream process (GTAG):

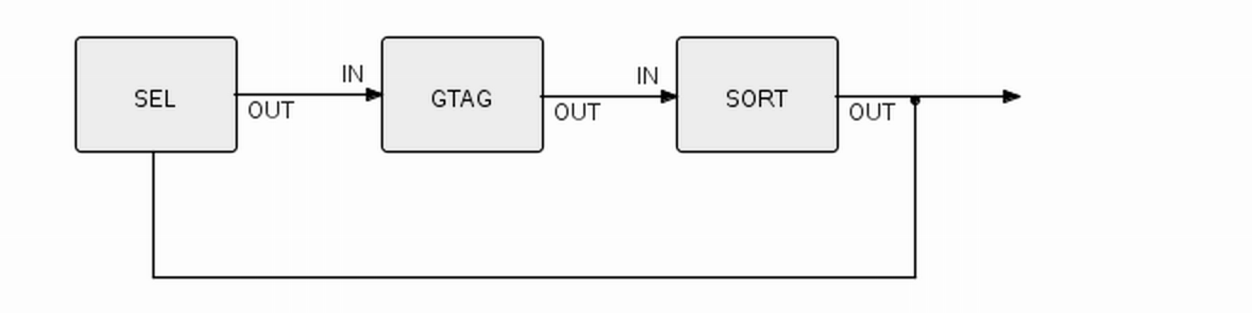



Figure 8.6

Actually Sort is an example of something I discovered quite early in the work on FBP: FBP enables you to tie together things which didn't expect to be tied together! If you can persuade something to accept and generate data packets, it can talk to other things which talk in terms of data. For instance, once you have converted Sort to a stream-to-stream component, it can talk to other utilities, HLLs, DB2, etc. They don't necessarily have to be callable - they just have to be able to accept and generate data once they are given control. I have written networks which contained Assembler, COBOL and PL/I programs, all in the same network. I also built an application which used a screen manager written in Assembler, REXX for all of its calculation logic, used reusable components for screen management and GDDM/PGF for drawing charts when requested by the user. Another network spanned two CMS virtual machines, communicating by means of VMCF. This is a point we'll come back to later: you can design a big network, and then split it across different machines, processors, software systems, etc.

Incidentally, there is a flip side to this ability to tie things together 'without their knowledge': we alluded above to the fact that components have to be reentrant if they are going to multithread with each other. Strange things happen if you try to multithread processes which are not reentrant! After we had turned Sort into a stream-to-stream process, it seemed reasonable to want to run more than one Sort process in a single network. As long as one had fully completed before the other one started, we had no problems. When we tried to overlap two Sorts in time, strange things happened. We suspected that reentrancy was being violated, but it did not consistently cause problems. Some people reported that everything ran fine, others that it always crashed! After we ruled out the possibility that it was programmer-induced, we eventually figured out that some of the sort techniques used by Sort were reentrant (probably the more recent ones), and some not, so some file formats or volumes caused problems (because they triggered different sort techniques), while others did not. We decided that the safest thing to do was interlock the Sorts so that one couldn't start until the other had finished (this is not a great solution as it requires deciding which Sort should run first, which you will probably have realized by now runs counter to the philosophy of FBP).

This anecdote reminds me of a pitfall when using Writers to write to PDS's under IBM's MVS (even if you are not an MVS user, other operating systems support similar structures, so it may still be instructive). PDS's are data sets which contain a series of subfiles (members), with a directory at the front. A number of programmers used multiple instances of the Writer component in DFDM to try to write more than one member at the same time. What these programmers saw was that the resulting members seemed to have a mixture of each other's data! In MVS, we are so accustomed to treating PDS members as if they were ordinary files that it is not immediately obvious why writing two or more at the same time should cause problems. Of course, the moment it is pointed out, it becomes obvious: PDS's only "grow" at the end. If a PDS already has a member "A", and you decide to write to member "A", what actually happens is that you write the new data at the end of the PDS, and afterwards, when the new member is complete, the directory is updated to point at the new member (leaving what is often referred to as "gas" in the middle [this happens if you are updating a PDS member, or a member has been deleted]). If you try writing to two members concurrently, the two writer processes will see the same "end of PDS", and will both start writing at the same spot. The two processes then write blocks alternately, after which both directory entries are updated to point to the old end of the PDS!

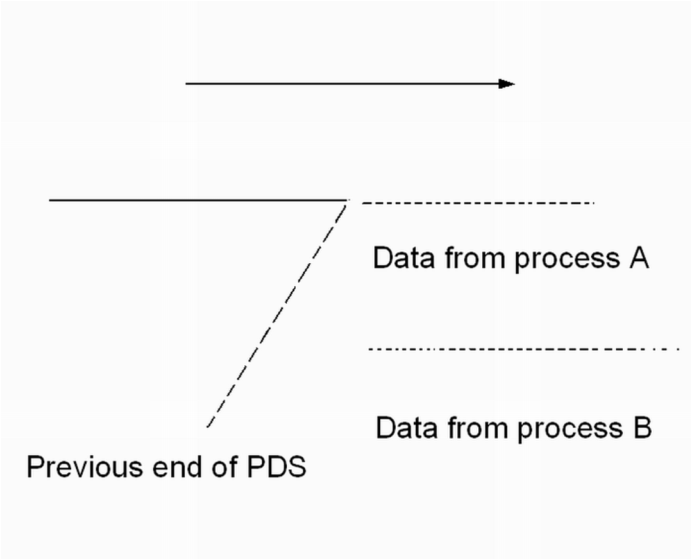



Figure 8.7

Now let's go back to the original Telegram problem, but first we are going to program it using conventional (control-flow) programming. Hopefully the reason for doing this will become clear soon.

From the above discussion, we see that words are the key concept needed to make this problem tractable. Once we realize this, we can go ahead and code it up, using something like the

following call hierarchy:

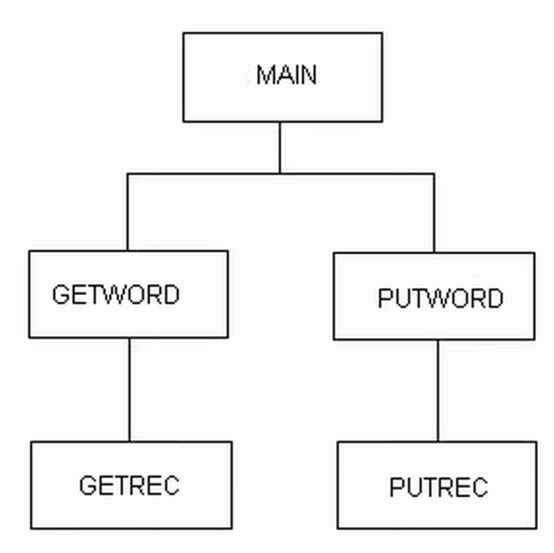



Figure 8.8

As we have said above, it is not at all obvious at first in conventional programming that this is the right way to tackle this job. Most people who tackle this problem start off by making GETWORD or PUTWORD the "boss" program, and promptly get into trouble. So we now realize that we have to "bring in a boss from outside" (call it MAIN), instead of "promoting" GETWORD or PUTWORD to boss. Now MAIN can call GETWORD and PUTWORD to retrieve and store a single word at a time, respectively. To do this GETWORD must in turn call GETREC and PUTWORD must call PUTREC, to look after the I/O. Now note that all four of these subroutines have to "keep their place" in streams of data (streams of words or streams of records). In the old days we did this by writing them all as non-reentrant code, so the place-holder information essentially became global information. This is quite correctly frowned on as poor programming practice, as it has a number of significant disadvantages, so today we normally manage this kind of place-holding logic using the concept of "handles". The general idea (for those of you who haven't had to struggle with this kind of logic) is for MAIN to pass a null handle (in many systems this will be a pointer which is initially set to zero) to GETWORD. When GETWORD sees the null handle, it allocates a block of storage and puts its address into the handle. Thereafter it uses this block of storage indirectly via the handle, and at the end of the run, frees it up again. Although this block of storage is allocated and freed by GETWORD, it is

considered to be owned by MAIN. This same logic is also used between MAIN and PUTWORD, between GETWORD and GETREC, and between PUTWORD and PUTREC. At the basic level, our problem is that subroutines cannot maintain internal information which lasts longer than one invocation. In contrast, FBP components are long-running objects which maintain their own internal information. They do not have to be continually reinvoked - you just start them up and they run until their input streams are exhausted. Thus, FBP components can do the same things subroutines do, but in a way that is more robust and, something of considerable interest for our future needs, that is also more distributable. It is important to note that FBP does not prevent us from using subroutines, but my experience is that they are most appropriate for such tasks as mathematical calculations on a few variables, doing look-ups in tables, accessing data bases, and so on - in other words tasks which match as closely as possible the mathematical idea of a function. Such subroutines are said to be "side-effect free", and experience has shown that side-effects are one of the most common causes of programming bugs. Hence subroutines which rely on side-effects for their proper functioning are a pretty poor basis on which to build sophisticated software!

At this point, to get you back in the data flow mood, let's talk about the text-processing example I talked about above. You remember the above example, where we want to take some text, split it into individual words, sort them, remove duplicates, insert fancy letters on every letter change, and print out the result. Oh, and let's print it out in two columns. Using FBP, this is quite simple - actually, we have just described the FBP network structure! I will leave this as an exercise for the reader. It also illustrates a point [the late systems architect] Wayne Stevens has made frequently - namely, that we very often want to string a whole bunch of functions together in a serial manner. We say "do A and B and C and...", which is basically the same as "take the output of A and feed it to B; now take the output of B and feed it to C, and so on..." This is exactly the same as the pipelines of the UNIX system or MS DOS. This is a very natural and important function, and it is essential to be able to express this kind of linkage with a minimum number of key-strokes. In the DFDM and THREADS interpretable notation we use two key-strokes (->) to represent this relationship.

So far, we have worked with quite simple structures which either string together "filters" in what is sometimes called a "string of pearls" pattern, or we had one data stream generate more than one ("divergent" patterns). There is a point to be made about divergent flows: once two streams diverge, they will no longer be synchronized. Some software systems go to a lot of trouble to keep them synchronized, but our experience is that, most of the time, it is not necessary or desirable. There are ways to resynchronize them if you really have to, but you may find it's not worth the trouble! For one thing, in the chapter on deadlocks we will find that resynchronization is a potential cause of deadlocks.

People very often split data streams so that different parts of their networks can handle different IP types - this works best if you do not need to retain any timing relationships between the

different types. If you do, there is a variant of the "string of pearls" technique that you may find useful: have each "pearl" look after one IP type and pass all the other ones through. Its (partial) pseudo-code would then look as follows:

```
receive from port IN using a
if type is XXX
        process type XXX
endif
send a to port OUT
```

You can then string as many of these together as you want and each pearl will look after its own data type and ignore all the others.

Now it's time to talk about various types of Merge function. There is a basic merge built right into FBP - the first-come, first-served merge. This is done very simply by connecting two or more output ports to one input port, as follows:

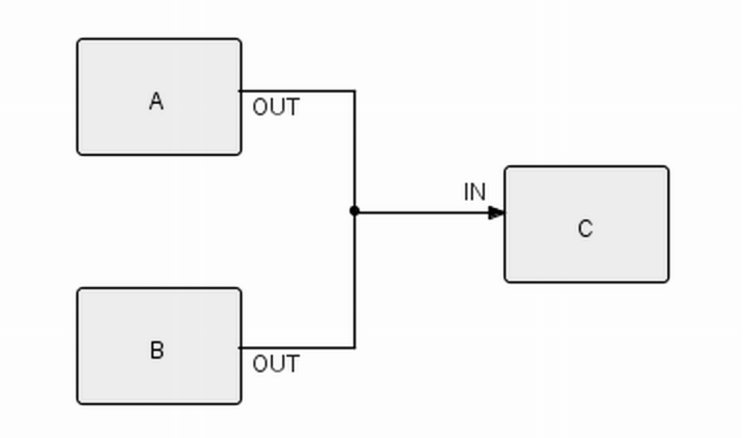



Figure 8.9

where ports OUT of A and OUT of B feed into port IN of C. IPs being sent out along this connection to the IN port of C will arrive interleaved. The IPs sent out by a single process will still arrive in the correct sequence, but their sequencing relative to the output of the other process will be unpredictable. Why would you use this structure? Surprisingly often! C might be a Sort, so the sequence of its incoming IPs is going to be changed anyway. The IPs from the two output streams may be easily distinguishable, so we can always separate them out later. C may be interested in receiving its input data as fast as possible - any additional sequencing may cause delays, and in fact may even cause deadlocks.

Now perhaps the first-come, first-served merge may not be adequate, in which case we will need a process at the junction of the two streams. This may be a custom-coded merge component, or you may be able to use one of the ones supplied with the FBP software, for instance Collate or Concatenate. Both of these use one or more port elements of a single array port to handle their input streams, so we will use this convention. The reason for this is that they can handle any number of input streams up to the implementation maximum. Of course, custom components could call one input port JOE and the other one JIM - it's up to the developer (of course assuming the component's users will put up with it!).

Our diagram now looks like this:

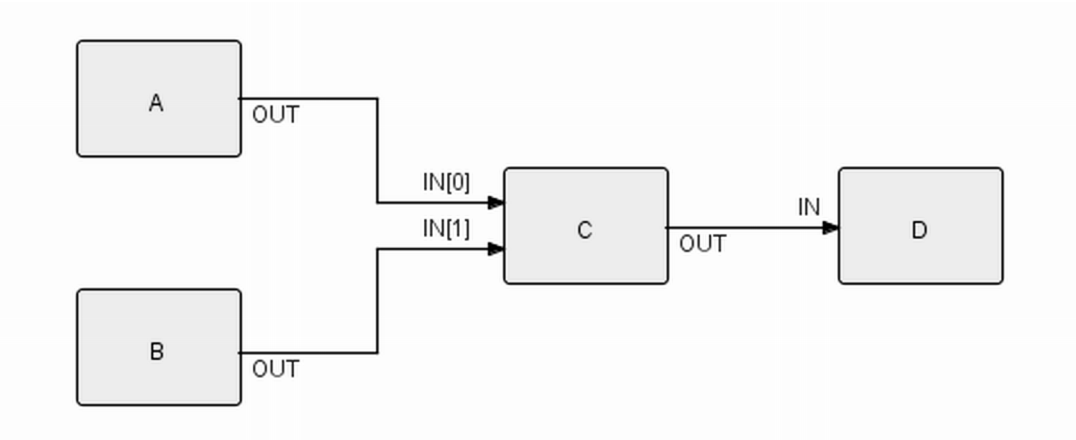



Figure 8.10

Now we have:

- OUT of A connected to IN[0] of C
- OUT of B connected to IN[1] of C
- OUT of C connected to IN D

If C is a Collate, then the output of A will be merged with the output of B according to key values - usually the key fields are specified to Collate by means of option IPs. Alternatively the Collate may use one or more named fields which will be related to the IPs by means of descriptors (see that chapter). One might also visualize Collates which are substream-sensitive - we might want to merge substreams based on a particular field in the first IP of each substream.

Now, if we were to use Concatenate instead of Collate, then what we are saying is that we want C to send all of A's output on to D before it accepts any of B's output. As we said above, there are some situations where this can be useful, too. Collate, however, is a very powerful component,

and in conjunction with the ideas described in the next chapter, significantly simplifies application programs which would be extremely complex using conventional programming techniques.

# Chap. IX: Substreams and Control IPs

We are now going to expand on the use of the Collate component mentioned in the previous chapter. This chapter will also show how Collate, substreams and control IPs can be combined to address one of the most difficult types of conventional business batch application. The main function of Collate, just as it was for the Collator Unit Record machine from which it gets its name, is to merge the IPs in its incoming data streams based on values in key fields (the definition of these key fields is normally specified in an options IP). In most applications, we have more than one key field, which are used to specify different levels of grouping. As an example, let's take a file of bank accounts within branches. In this particular bank, we'll say that account numbers are not guaranteed to be unique across branches. Another way of saying this is that to make an account number unique across the whole bank, we must specify the branch number.

Suppose we have an application where a stream of banking transactions must be run against a stream of account records: we first sort both streams by account number within branch number. Now in conventional programming, we have to write an "Update" program. I once figured that something like a quarter of all business programs running today are Updates! Whether or not that is the right figure, Update programs are hard to code and harder to modify, and yet the only assistance programmers have ever received is a piece of paper, handed down from father to son, showing the basic logic for Updates, which is a pattern of moves and compares usually called the "Balance Line" technique. This logic then has to be modified by hand to suit the particular situation you are faced with. However, it is still only a description of an approach - to adapt it to a particular application you have to massively modify it. I once had the dubious pleasure of having to modify an update program whose author had written the client an explanation of why his request could not be satisfied, which started: "Owing to the limitations of data processing,..." My clear recollection, after almost 25 years, is that modifying that program (and, for a conventional program, it was really quite well-written) was *not quite* impossible!

Now imagine what you could do if you had a prewritten, pretested component for collating data streams. Let us imagine that we have a stream of transactions and a stream of accounts, both sorted by account number within branch. We will now collate them into a single stream, specifying branch number and account number as major and minor control fields, respectively. When Collate finds two equal records from two different port elements, it outputs the one from the lowest-numbered element first. The resulting output stream will contain the following sort of pattern:

```
IP type      branch    acct #       date        amount    DEP/WD
 account         1         1
 trans           1         1       1992/3/12      12.82      DEP
 trans           1         1       1992/3/12     101.99      WD
 trans           1         1       1992/3/12      43.56      WD
 trans           1         1       1992/3/26      54.77      WD
 trans           1         1       1992/3/26      12.26      WD

 account         1         2
 trans           1         2       1992/3/03      34.88      DEP
 trans           1         2       1992/3/03      10.00      WD
   .
   .
   .

 account         2         1
 trans           2         1       1992/2/29      25.99      DEP
 trans           2         1       1992/3/25      87.56      DEP
 account         2         3
 trans           2         3       1992/3/01      34.88      WD
 trans           2         3       1992/3/17      88.22      DEP
   .
   .
```

Figure 9.1

Notice that the effect of Collate operating on sorted input streams is to give us a nicely sequenced and grouped data stream, consisting of two kinds of data IP. The job of the process downstream of Collate is therefore much simpler than the conventional Balance Line, which has to do this grouping as well as implement the required business logic. A conventional Update also has to worry about what happens if one file is exhausted before the other. Instead, in our FBP solution, the actual business logic (as compared with all the logic of synchronizing the two data files) sees one IP at a time, determines its type, and decides what to do with it. In what follows, we will call this process UPDATE_ACCTS. One rule of thumb in conventional programming is that the complexity of a program is roughly proportional to the square of the number of input files. Just one reusable component, Collate, therefore can reduce the complexity of UPDATE_ACCTS

significantly!

So far we have talked about the branch and account levels. Now let's assume we want to group transactions by date - bank statements often only show one subtotal per day. This then gives us three grouping levels, most of which are only recognized by changes in control fields. A change of account number is recognizable by the arrival of a new Account IP, but this cannot tell us when we have started a new branch. So a lot of the logic in a conventional Update is keyed to *changes in value* of control fields. Now, Collate has to look at all the control field values anyway, so it would be nice if we could have Collate figure out the groupings and pass that information to downstream processes, which would therefore be relieved of all this comparing to see when a given group starts or finishes. How does Collate pass this grouping information downstream? You guessed it! We use the "bracket" IPs mentioned in Chapter 3.

Bracket IPs have a recognizable type which follows a special convention, so that they can never conflict with user-defined types. Brackets come in two flavours: open and close brackets. They may also contain real data (if their IP length is non-zero), which by convention we use for the name of the group they delimit. Let's get Collate to insert some brackets into its output data stream, resulting in a collated data stream that looks like the following diagram. As before, I will use bracket symbols to represent open and close bracket IPs, but this time we will show the names of the groups they refer to in the data part of the bracket IP ("date" means a group comprising all the deposits and withdrawals for a given date). To make things a bit clearer, I will show a "level number" (L) in front of each IP - open brackets increase the number, close brackets decrease it). I will just show the first few IPs of the collated stream:

```
L  IP type     branch    acct #     date     amount   DEP/WD
0 |<            branch
1 |<            account
2 |account          1        1
2 |<            date
3 |trans            1        1    1992/3/12    12.82    DEP
3 |trans            1        1    1992/3/12   101.99    WD
3 |trans            1        1    1992/3/12    43.56    WD
3 |>            date
2 |<            date
3 |trans            1        1    1992/3/26    54.77    WD
3 |trans            1        1    1992/3/26    12.26    WD
3 |>            date
2 |>            account
1 |<            account
2 |account          1        2
2 |<            date
3 |trans            1        2    1992/3/03    34.88    DEP
3 |trans            1        2    1992/3/03    10.00    WD
2 |>            date
```

Figure 9.2

Generally, the logic of UPDATE_ACCTS will consist of a "case" statement based on the type of the incoming IP. An open bracket will cause counters and totals for the correct level to be initialized to zero; a close bracket will cause an IP containing the counters and totals for that level to be sent to an output port. We won't even have to reinitialize the counters and totals at this point because we know that another open bracket will be coming along shortly (or end of data). We could either update the counters and totals at every level for every incoming data IP, or just roll the values into the next level up at close bracket time - it seems simpler to choose the latter. There is some redundancy in the data structure, as an account IP is always immediately preceded by an account open bracket, but this is much better than not having enough data! Since we will be needing information from the account IP, we can just ignore the account open bracket, or we can do a cross-check that they are both present ("belt and braces" programming - that's "[belt and] suspenders" for American readers!).

So far the main piece of logic for UPDATE_ACCTS (the process downstream of the Collate) looks roughly like this:

```
receive incoming IP
   begin cases based on IP type
       case: open bracket for branch
          initialize counters and totals for branch
       case: open bracket for account
          initialize counters and totals for account
       case: open bracket for date
          initialize counters and totals for date

       case: account
          pick up account info
       case: transaction
          increment counter for debit or credit
          add amount to debit or credit total
       case: close bracket for date
          output IP containing counters and totals for
            date
          roll these values to account level
       case: close bracket for account
          output IP containing counters and totals for
            account
          roll these values to branch level
       case: close bracket for branch
          output IP containing counters and totals for
            branch
   end cases
```

Figure 9.3

Notice that these groupings are perfectly "nested", and at any point of time we are only looking at one level or at most two adjacent ones. This suggests that we could handle this kind of logic very elegantly using a push-down or "last in first out" (LIFO) stack. You will remember that this is a storage structure which is added to and removed from at one end only. We also have to decide what kind of data structure to hold all these counters and totals in. One possibility would be an array, but for each level we have to eventually output an IP containing the values for that level, so why not hold the values in IPs all the time? This kind of IP is called a *control IP*, and it can be described as an IP whose lifetime corresponds exactly to the lifetime of a substream, which it can be said to "represent".

Now we can put these techniques together and deduce that we will be working with a *stack containing control IPs* (actually their handles). So the above logic becomes much simpler. It now reads something like this:

```
receive incoming IP
  begin cases based on IP type
      case: open bracket
         create IP for this level
         initialize counters and totals for this level
         push IP onto stack
      case: account
         pick up account info, insert into account IP
      case: transaction
         increment counter for debit or credit in IP
           currently at head of stack
         add amount to debit or credit total in IP
           currently at head of stack

      case: close bracket
         pop IP off stack
         roll counters and totals in this IP into IP
           currently at top of stack, if there is one
         output IP which was just popped off stack
  end cases
```

Figure 9.4

When we say "create an IP" in the above logic, we may simply use the incoming IP if there is a characteristic type at the beginning of the group. Bear in mind, however, that the IP must have data fields in it for the totals. The most obvious technique is to create a control IP, copy fields such as identifiers across from the incoming IP (frequently the one after the open bracket) into the control IP, stack the control IP and throw away the original incoming IP. A faster technique, though, which can sometimes be used, is to arrange for the incoming data to be put into a bigger IP at the time it is read in by the application. It can then be stacked immediately, without the tedious "create new, copy, destroy old" sequence.

Now we have our control IPs, what do we use as a stack? AMPS had a stack mechanism specifically for this kind of logic, which was actually a kind of connection (one that was only attached at one end, like a cul de sac in a city). The later versions of DFDM did not provide stack mechanisms, but, since we are only dealing with one component, it was easy enough to set up an array of IP handles in the process's working storage and "push" and "pop" (the basic stack operations) by incrementing or decrementing an index over the array. However, I happen to like stacks, and we shall see in a later chapter there are striking similarities between the way components parse their input streams and the way compilers parse their input. In both cases stacks are the natural mechanism for keeping track of nested structures. We have accordingly provided a stack mechanism in THREADS.

You may have noticed that Figures 9.3 and 9.4 did not have the familiar "do while receive has not reached end of data" - this is because they are written in a style which assumes that they end execution after processing each IP. In FBP an inactive process will be invoked the next time an IP arrives at any of its input ports, so this kind of component will be invoked once for each incoming IP. A result of this is that it cannot maintain continuity across multiple IPs, but this is where the stack comes in. Since the stack is outside the process's local storage, continuity can be maintained across the invocations using the stack. This style of component is called a "non-looper", as opposed to components written in the "do while receive has not reached end of data" style, which are referred to as "loopers". This is not an externally detectable attribute of a component, but just depends on when and how often the component decides to end processing - as long as there is data to be processed, it will continue being reinvoked.

You may be wondering what the advantages of non-loopers are, if any. Well, for one thing, non-loopers end execution more frequently, so IPs which have not been disposed of are detected sooner, making such errors easier to find. Also, non-loopers' local storage is only used within one invocation, so there is less opportunity for one IP's logic to interfere with another's. Also we shall see in the chapter on Checkpointing (Chapter 20) that there is an advantage to having processes yield control as often as possible. As always, there are pros and cons.

Obviously, there is still some application logic left to be written for UPDATE_ACCTS, but I have tried to show that the approach of holding control IPs in a stack (together with a generalized Collate) does significantly simplify the remaining piece of logic you do have to write. One of the things that also makes this logic simpler is the fact that every action is associated with the arrival of a distinct type of IP (this is even more obvious in the case of non-loopers), rather than a change in value - this is what allows one to cleave the logic of such a component into distinct self-contained cases. When talking about such logic, I often find it useful to refer to "open bracket time", or "detail time". An incoming IP triggers an action, which starts up and then finishes, readying the process for the arrival of the next IP (or end of data). In a later chapter, I will try to show that the code we still have to write has such a simple structure that we can begin to think about generating it semi-automatically from some kind of specification other than a HLL

program.

By the way, as you work through this kind of logic, you may notice a characteristic flavour of FBP coding: very little of the data you will be dealing with actually resides in a process's working storage - the vast majority of it will be in IPs, very often control IPs like the ones we have just been discussing. When you think about it, this should not be that strange - in a real life office, most of your data is in files, memos or on the computer - how much data do you have to hold in your personal short-term memory? I don't know about you, but I try to hold onto as little data as possible for as short a time as possible (after which I destroy it, pass it on it or file it - just like IPs).

# Chap. X: Some More Components and Simple Applications

In this chapter, we will be working with a more complex example, the Sales Statistics application described in (Leavenworth 1977). The referenced paper describes an application in which a sorted detail file of product sales is run against a product master file, producing an updated master file and two reports: a summary by product and a summary by district and salesman. The figure on the next page, which originally appeared in (Morrison 1978), shows the FBP process network for this application.

In the conventional approach to building this application, we would first split off the district/salesman summary into a separate job step preceded by a Sort. This leaves us with a function which accepts two input files and generates three outputs (updated master, product summary and Sort input, also referred to as extended details). This function must pass details against masters, take care of the fact that one of the files will usually terminate before the other, handle control breaks, detect out-of-sequence conditions, etc., etc.

As we said in the previous chapter, the Collate component is key to simplifying this kind of application. The resultant diagram looks like this:

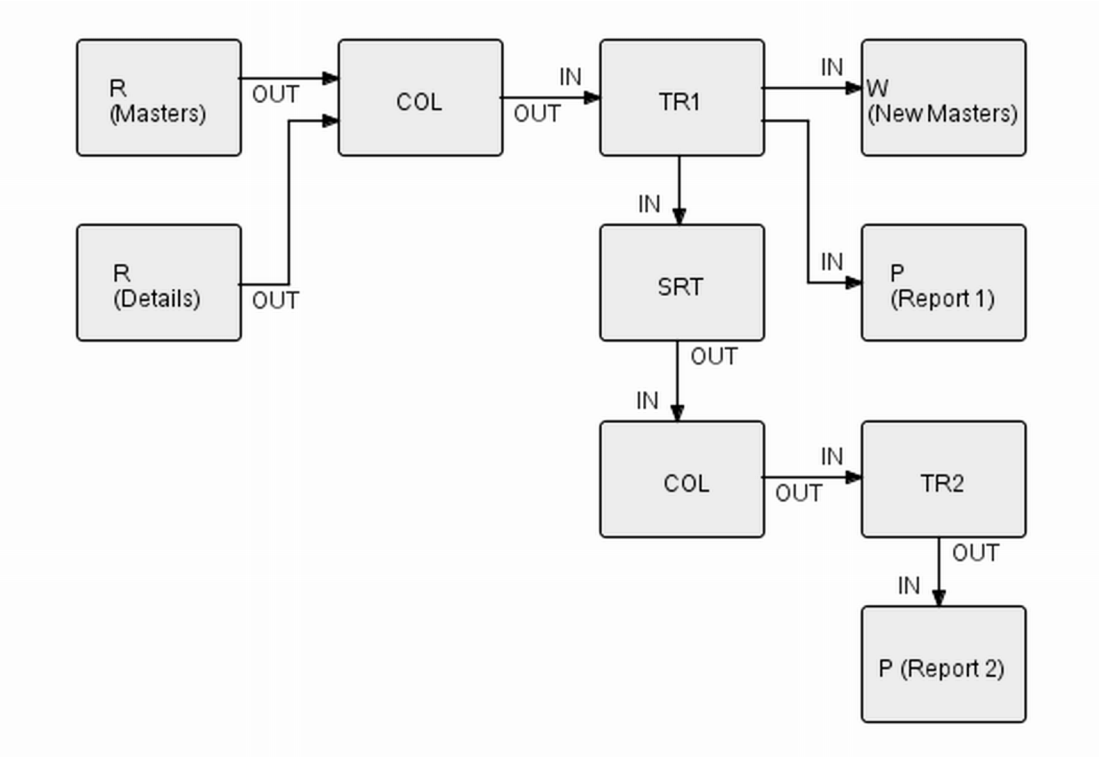



Figure 10.1

where

- R is a Read component.
- W is a Write component.
- COL is a generalized Collate which merges two or more streams on the basis of specified control fields and inserts bracket IPs between IPs with different control field values. (If used with only one stream, it simply inserts bracket IPs - this is the case in the second occurrence of COL).
- P is a Print component.
- TR1 and TR2 correspond to Tran-1 and Tran-2, respectively, in B. Leavenworth's paper
- SRT is a generalized "Sort" component which sorts the Extended Details coming out of Tran-1, by Salesman within District.

The output of the Collate component consists of sequences of groups, called "substreams", each

consisting of an open bracket, a master, zero or more details, and a close bracket.

This can be shown schematically as follows:

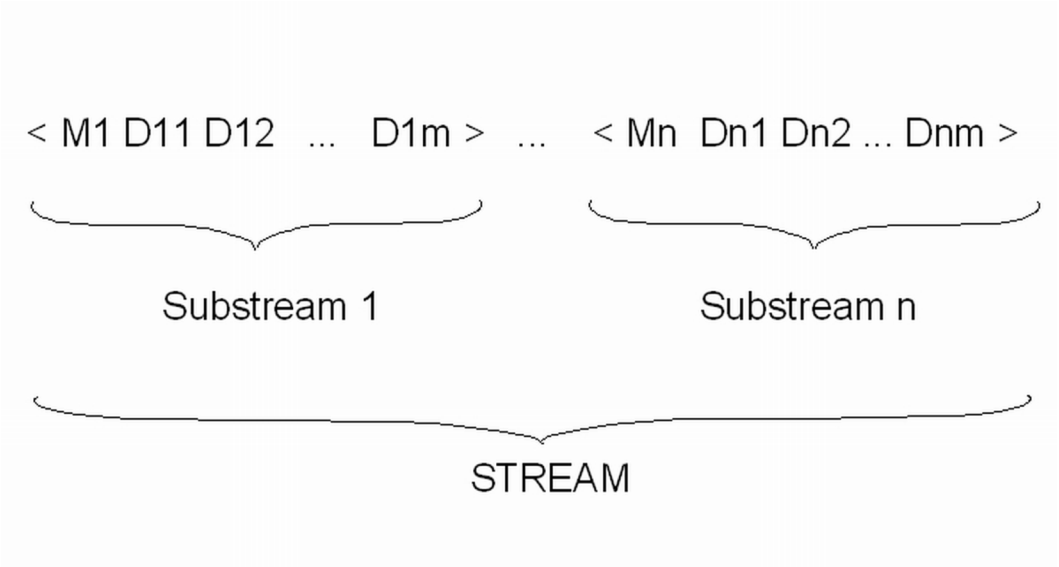



Figure 10.2

J-D. Warnier (1974) uses a vertical form of the above diagram to define the input and output files of an application, and uses this to determine the structure of the code which has to process them. Unfortunately, control flow programming requires that one of the files has to become the driver in terms of the overall program structure, so that, if there are any significant differences between the structures of the different files, the program structure becomes less and less easy to derive and understand, and hence to maintain. In FBP, this structure tells us important things about just those components which receive or send this particular stream structure, so it remains an extremely useful device for understanding the logic of the application.

The input stream for TR1 is shown as it might be expressed using an extension of J-D Warnier's notation:

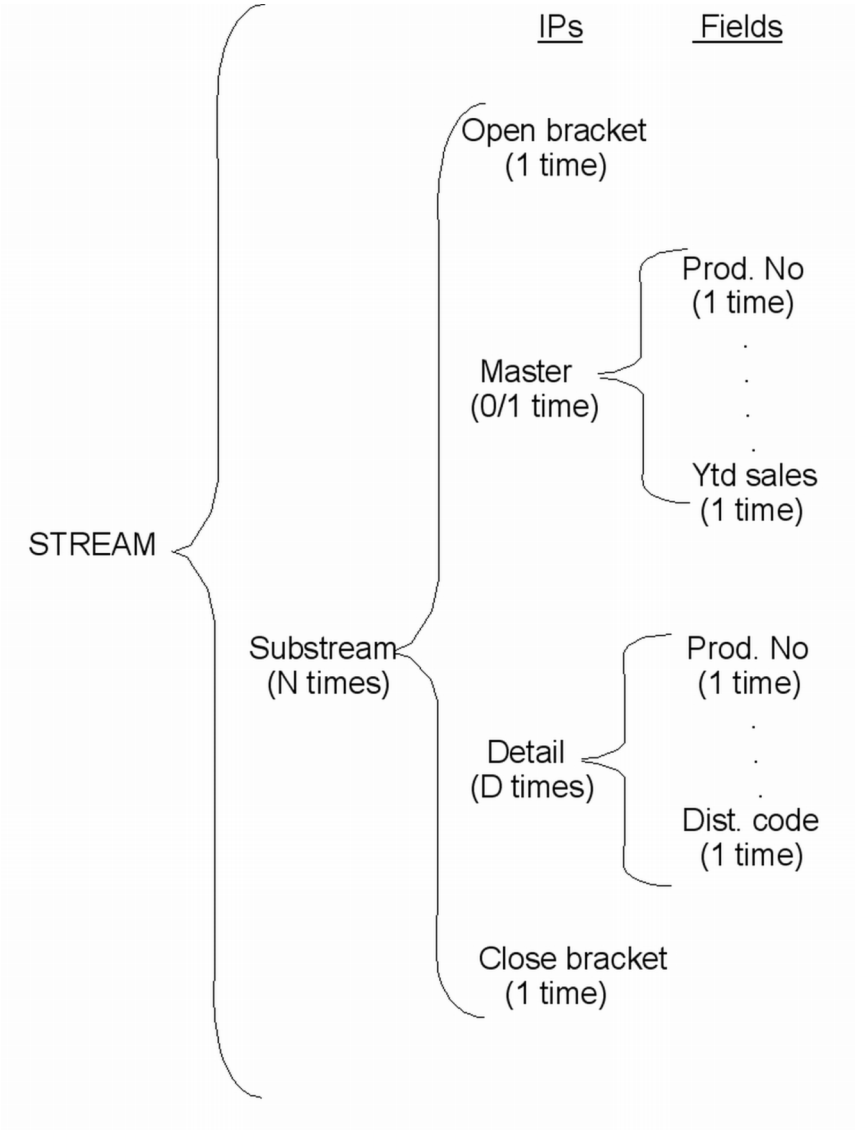



Figure 10.3

Of course, in Warnier's book, the above type of diagram is used to describe actual files, rather than FBP streams, but I believe it generalizes quite nicely to IPs, substreams and streams. The last column, of course, is fields within IPs.

Another methodology with close affinities to Warnier's is the Jackson methodology, already

alluded to. He uses a horizontal version of this notation, using asterisks to indicate repeating items. Using his notation, this diagram might look as follows:

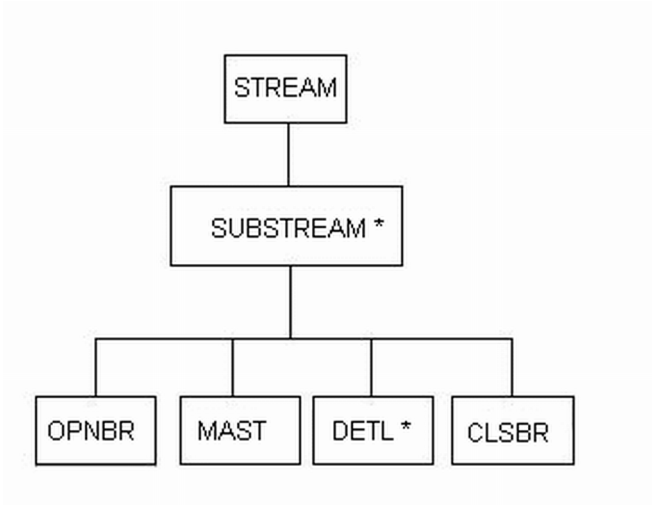



Figure 10.4

Now, going back to our example, TR1 generates three output streams - one consisting of updated master records, one of summary records, which are similar to masters but have a different format (they are intended for a report-printing component), and one of "extended" details: detail records with an "extended price" field (quantity times unit price) added.

The following computations must be performed:

```
extended price (in detail):=
        quantity from detail * unit price from corresponding
            master record
product total (in summary):=
        sum of extended prices over the details relating
           to one product master
year-to-date sales (in summary and updated master):=

        year-to-date sales from incoming master record +
           product total
```

Figure 10.5

We have talked above about using non-loopers with stacks to handle nested streams and

substreams. If we add a stack to our TR1 components, we get the following "blown up" picture of TR1 (the stack is not shown in a network definition - I just show it because it is "external" to the process):

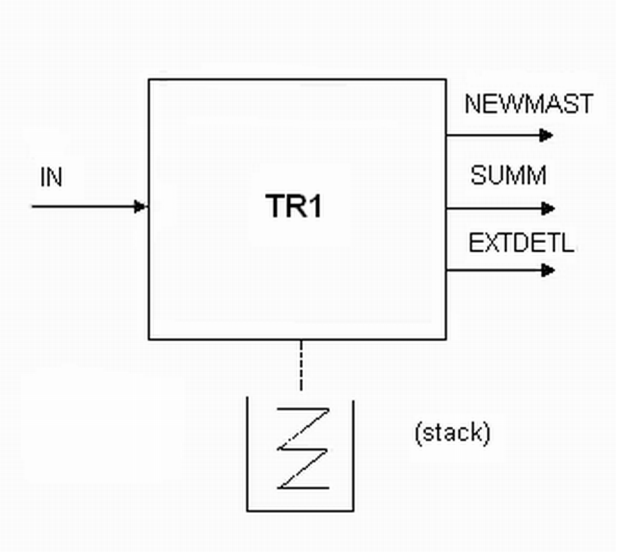



Figure 10.6

Here is the logic that needs to be performed for each incoming IP (as you can see, it is very similar to the logic we showed in the previous chapter):

- At "open bracket time",
  - create a control IP
  - clear the total quantity field in this new IP
  - store the IP in the stack
- At "master time",
  - obtain the control IP from the stack
  - copy the information from the incoming master into the control IP, such as unit price, year-to-date sales, etc.
  - discard the master IP
  - replace the control IP in the stack (you have to remove an IP from the stack before

it can be processed)

- At "detail time",
  - obtain the control IP from the stack
  - update the total quantity by the quantity in the detail IP
  - calculate an extended price for the detail
  - put out the extended detail to its own output port
  - return the control IP to the stack
- At "close bracket time",
  - obtain the control IP from the stack
  - create a summary
  - calculate the product total (dollars)
  - format the summary IP and put out to the summary port
  - create a master IP with the information from the control IP, and put to the "updated master" output port
  - discard the control IP
  - the stack is now empty, so that when the next open bracket arrives, a new control IP can be "pushed" onto the stack, preserving the stack depth
- At end of data,
  - the process closes down, resulting in its output ports being closed, which in turn allows its downstream processes to start their own close-down procedures.

Note that summary and updated master IPs are not output until "close bracket time", as all the details for a given master have to be processed first.

FBP is chiefly concerned with dynamic IPs, rather than with variables. Although it might at first glance seem that this would result in undisciplined use and modification of data, in fact we have better control of data because each IP is individually tracked from the moment of creation to the time it is finally destroyed, and it cannot simply disappear, or be duplicated without some component issuing a specific command to do this. During an IP's transit through the system, it can only be owned by one process at a time, so there is no possibility of two processes modifying one IP at the same time. We in fact monitor this at execution time, by marking an IP with the ID of its owning process: the act of getting addressability to an IP, if successful, confers "ownership"

of that IP on the process doing it.

In fact, in this example almost all modifiable data is in IPs, and there is no global data at all. FBP did not require a global facility during its earlier years, and, although it has been added to some dialects of FBP, it is still only used very occasionally.

We can also use Figure 10.1 to illustrate how easy it is to modify this kind of network, whether it is to satisfy business requirements, improve performance, or for whatever reason. Here is what I said in my article (Morrison 1978) about how this diagram might be modified (this article uses the term DSLM for the cluster of concepts which we now call FBP):

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

I apologize for the use of "he" and "salesman", but this was 15 years ago [when the book was written - 1994]! I then went on to suggest that the subnet demarcated by the dots in the diagram below could be replaced by a network which updates totals at random, then signals a scan and report function to display the resulting totals, i.e.

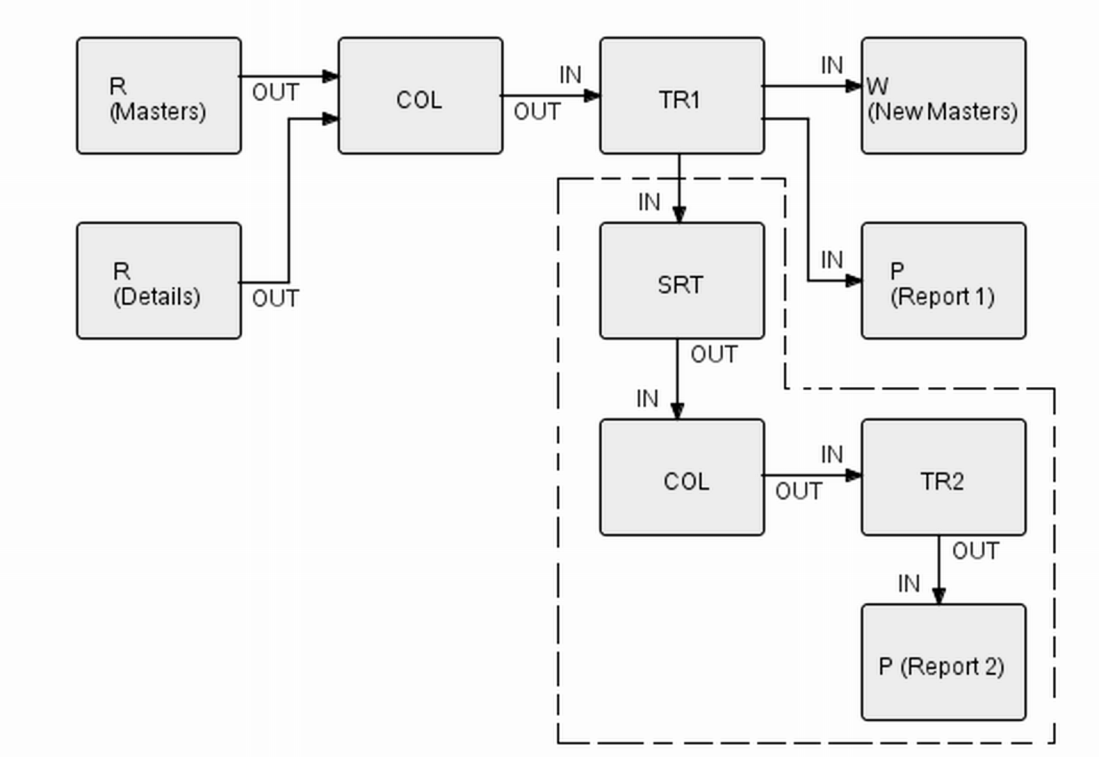



Figure 10.7

could be modified as follows:

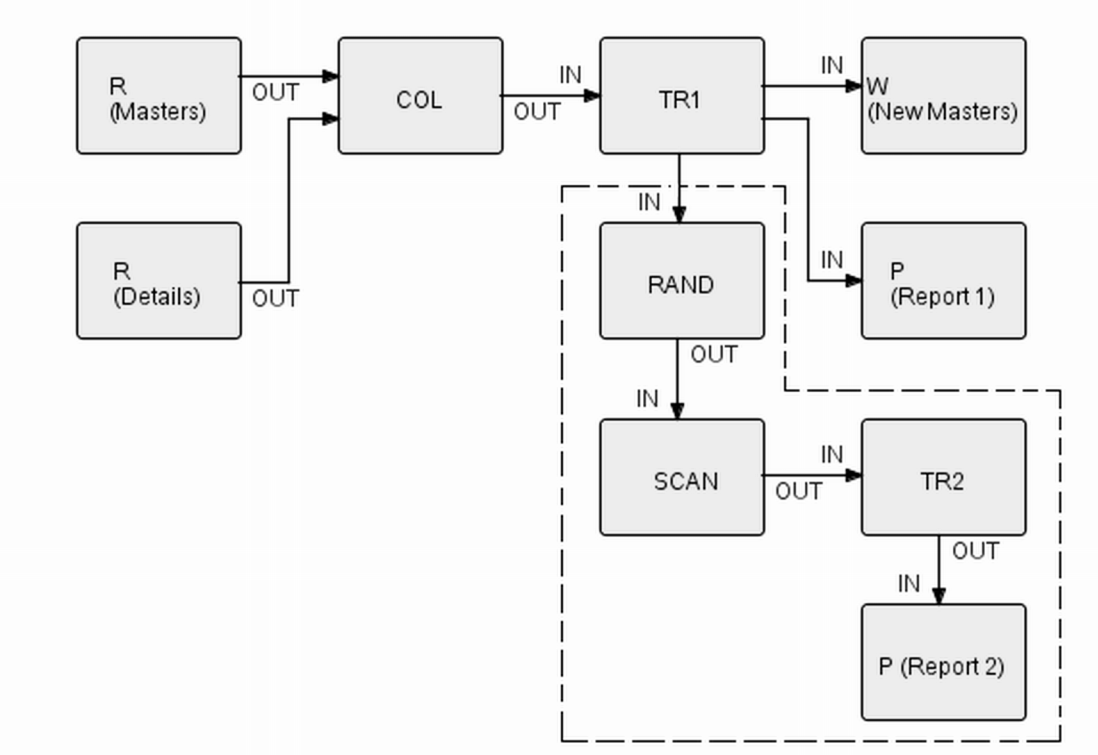



Figure 10.8

where RAND is a component which updates totals in a table at random, while SCAN goes through all the totals at end of job, generating report lines.

How does SCAN get triggered and, once triggered, how does it get access to the table which has been built by RAND? Well, since, in FBP, all that moves through connections are the handles for IPs, why not have RAND just "send" the whole table? This ensures that SCAN doesn't start until RAND has finished, plus it takes care of SCAN getting addressability to the table - *at the right time*! In conventional programming, tables don't move around - and, in fact, in FBP they don't either, but it is very convenient to make them appear to!

I mentioned a "report generation" component above. You will find this essential for your batch business applications, and it illustrates the power of FBP and also FBP's ability to help you divide function into manageable components. For almost all business applications we found that we needed a component which would accept formatted lines and combine them into report pages. An upstream process can have the job of generating the formatted lines, and this type of function should be kept separate from page formatting. FBP is a "black box" reuse tool, and in fact your Report Generator component can be used as a black box by your applications. As such, it can

implement the standards for reports in your installation, and it can make it easier for programmers to conform to your standards by making it less work (definitely the best way to encourage adherence to standards)! So far, so good. However, every shop has a different report standard, so you will have to build your own black box to embody your own standards. Luckily it is easy to build new components using the FBP API calls. If we decide that we want to distribute this kind of component more widely, it seems to call for a different distribution technique: either in source format, or implemented as a pure black box, but with installation-provided exits, or as a black box supporting a mini-language. Perhaps we should call this a "grey box".

Here is a list of the facilities we provided in the Report Generator we used with DFDM in our shop:

- accept two lines of permanent title information as run-time options, and combine these with date information in an installation-standard format
- accept two dates for the run: the actual date and the effective date ("as of" date)
- accept additional title information from an additional IP stream which could be changed dynamically (on receipt of a "change title" signal)
- generate page numbers
- accept a signal to reset the page number
- generate an "end of report" box at the end of the report (this lets the human receiving the report know it is complete)
- generate a "report aborted" box on the report on demand
- support all of the above features in English, French or bilingual English and French (under control of an option).

This may seem like a long list, but it basically embodied a preexisting set of shop standards, some of which were supported by subroutines, but some weren't. Now, instead of having to mandate a standard which people see as extra work, we had a component which did it "automatically". In our experience, it is much easier to enforce a standard which saves people work. If they use your component, their reports will follow shop standards, and your systems will also be more reliable and cheaper to build and maintain. You can tell your developers, "We don't mind if you don't follow standards, but it'll cost you, and, if you miss your deadlines, we'll be asking for an explanation!". The same philosophy was followed in the early days of hardware development. Designers were perfectly free to create new components, but they had to carry the whole cost of development, testing, etc., themselves. Today, it is no coincidence that the vast majority of personal computers are built around a very small number of different CPU chips. This is a variant of a point we will come back to often - only by changing the *economics* of application development will we get the kind of behaviour we are trying to encourage.

One last point may appear obvious at first: this Report Generator assumes that its input IPs are fully formatted report lines. Formatting of these report lines may logically be split off into separate components. Now, programmers rooted in conventional programming may feel that such rigid separation is not possible, but they have not had the experience of using a separate component which makes such separation attractive economically, as well as logically. Once such a thing exists, people find that they can make intelligent decisions balancing esthetic considerations against economic ones. Without such a component, you don't even have a choice!

A component like the Report Generator can be added to your application incrementally - that is, you first get your application working with a simple line-by-line printer component, then replace that with the report generation component to produce a good-looking report. Although you are introducing more function into your application, you can predict very accurately how much time it is going to take to do this. In conventional programming, many writers have commented on the exponential relationship between size of application and resources to develop it. This graph typically has the following shape:

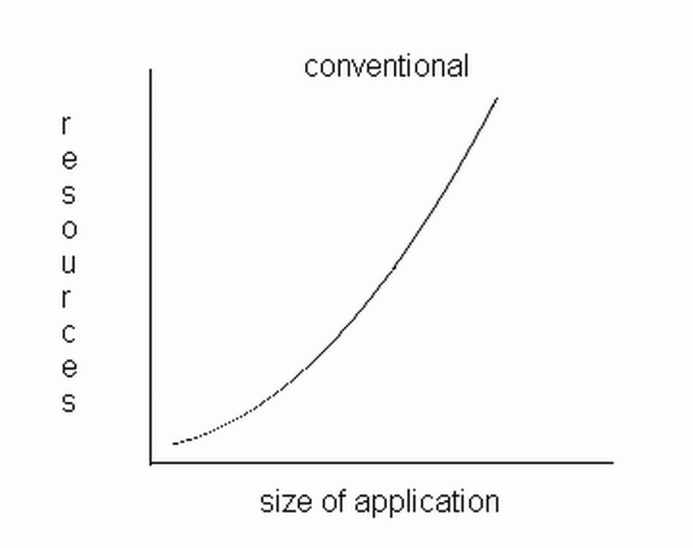



Figure 10.9

Development using FBP shows an essentially linear relationship, as follows:

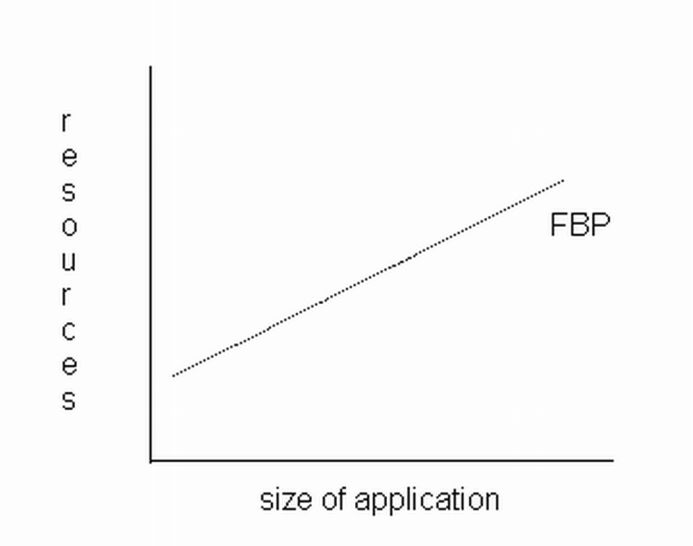



Figure 10.10

Superimposing the two graphs (even if we allow for the possibility that start-up costs may be slightly higher - in FBP you tend to do more design work up front), we get the following picture:

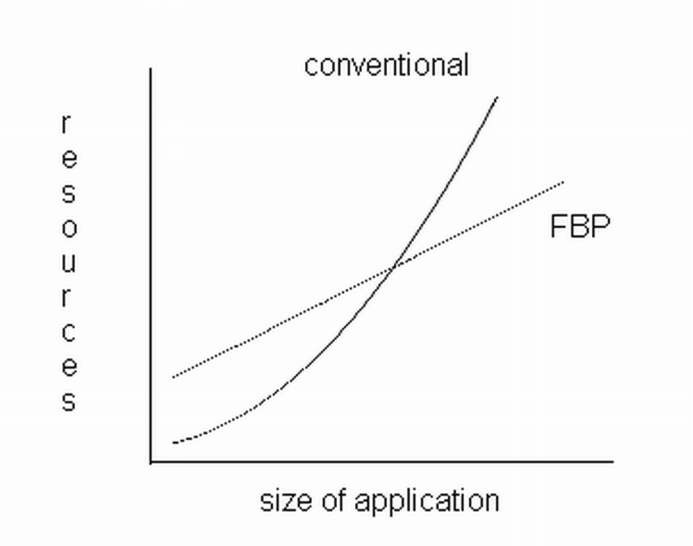



Figure 10.11

Clearly at some point (and we have found this to occur even with surprisingly small

applications), FBP's productivity starts to overtake that of control flow, and in fact gets better the larger the application. *FBP scales up!* My colleague Chuck (he of the 200-process application) didn't have to worry that his application was going to become harder and harder to debug as it got bigger - he just built it methodically, step by step - and it has since had one of the lowest error rates of any application in the shop. Finally, in case you think that this only worked because he was a single individual who could hold it all in his head, ask yourselves what are the requirements for successfully managing a big project. Surely, some of the more important ones are exactly what FBP provides so well: a consistent view, clean interfaces and components with well-defined functions.

# Chap. XI: Data Descriptions and Descriptors

[*Recent addition: The reader will note similarities between this chapter and Object-Oriented concepts. We in fact implemented a number of the concepts described below in a recent e-business application using the JFBP package. This was a layer of function below FBP (it didn't care how the higher levels modules were hooked together), and it totally prevented currency amounts from being multiplied together! To achieve this, of course, you have to hide the internal representation of a currency amount, which in turn means being willing to continually add more and more methods to each class as people come up with new uses. Maybe one day we will have identified all possible uses of currency amounts (or any other business data type)... but I'm not holding my breath!*]

Up to now, you have probably noticed that we have been assuming that all components "know" the layouts of the IPs they handle. The layouts of all IP types that a component can handle and the IP types that it can generate become part of the specification of that component, just as much as the overall function is. If you feed something raw material it can't digest, you are bound to have problems, just as in the real world!

In conventional programs, you usually use the same layout for a structure or file record in all the subroutines of a program - in FBP actually only each pair of neighbouring processes has to agree on the layout. This means that a process can receive data in one format and send it on in a different format. If two neighbouring processes (perhaps written by different suppliers) are using different layouts, all you have to do is add a transform process in between.

Now suppose you at first wanted to have two neighbouring processes communicate by means of 20-element arrays. You then decide this is too restrictive, so you stay with the arrays, but allow them to communicate the size of the arrays at run-time. This is a type of "metadata", data about data, and can be as much or as little as the two processes involved want. For instance, their agreement might specify that the array size is to be positioned as a separate field (it could even be

a separate IP) ahead of the array.

Now most higher level languages don't support metadata very well, so you are dependent on having the data formats imbedded in a program. Also, it is easy for old data to get out of step with the programs that describe them. There is a perhaps apocryphal story that somebody discovered a few decades ago that the majority of the tapes in the US Navy's tape library were illegible. It wasn't that the tapes had I/O errors - they were in perfect shape physically - but the problem was that the layouts of the tape records were hard-coded within code, and nobody knew which programs or copy code described which tapes! I'm sure that such a problem, if it ever existed, will have been remedied long ago! [tongue firmly planted in cheek!]

In DFDM we extended the idea of metadata by providing run-time descriptions which could be attached to IPs passing across a connection. DFDM allowed the creator of an IP to attach a separately compiled descriptor to the IP, which was used by special DFDM services which allowed fields to be accessed by name. Whenever an IP was created, the "allocate" function could optionally specify a descriptor which was to be permanently associated with that IP. All the IPs of the same type would share the same descriptor [shades of OO].

In DFDM these access services were called GETV and SETV ("get value" and "set value"). They had the advantage that, if you ever wanted to move a field (call it AMOUNT) from one place to another in the structure containing it, you didn't have to recompile all the programs referring to it. Another advantage was that, with a single call, a component could access a field which might be at different offsets in different types of incoming IP. For example, the field called AMOUNT could be at different locations in different IP types, and the component would still be able to access it or modify it, as long as the IPs had descriptors. The DFDM GETV and SETV services (and their descendants) were designed to be called from S/370 Assembler or from the HLLs we supported. They also provided limited conversion facilities between similar field formats - for example, between 2-byte and 4-byte binary fields, between different lengths and scales of packed decimal, or between varying and non-varying character strings. Thus you could specify that you wanted to see a binary field as 4 bytes in working storage, even though it was only 2 bytes in the IP. As of the time of writing, these services have not been implemented for THREADS.

Without such a facility, the layout of incoming IPs has to be part of the specification of all components. This points up another advantage of GETV and SETV: if a component is only interested in three fields, only those specific field names have to be mentioned in the specification for the component, rather than the whole layout of the IPs in question.

When you add this facility to the idea of option IPs, you get a powerful way of building more user-friendly black boxes. For instance, you could write a Collator which specified two field names for its major and minor keys, respectively. It would then use GETV to locate these fields in the correct place in all incoming IPs. "Collate on Salesman Number within Branch" seems much more natural than "Collate on cols 1-6 and 7-9 for IP type A, cols 4-9 and 1-3 for IP type

B," and so on. Thus it is much better to parametrize generalized components using symbolic field names, rather than lengths and offsets. The disadvantage, of course, is performance: the component has to access the fields involved using the appropriate API calls, rather than compiled-in offsets. However, the additional CPU time is usually a negligible cost factor compared with the cost of the human time required for massive recompiles when something changes, or, still worse, the cost of finding and correcting errors introduced while making the changes!

["Smart" Data...]

When we looked at the problems of passing data descriptions between components, we rapidly got into the problem of what the data "means": in the case of our conventional Higher Level Languages, the emphasis has always been on generating the desired machine instruction. For instance, on the IBM S/370, currency is usually held as a packed decimal field - 2 digits per byte (except the last byte, which has one digit and a sign), and usually amounts are held with 2 decimal places (these usually have special names, e.g. cents relative to dollars, new pence relative to pounds, and so forth - are there any mixed radix currencies left in the world?). Since the instructions on the machine don't care about scale (number of decimal places), the compiler has to keep track of scale information and make sure it is handled correctly for all operations. (You could conceivably use floating point notation, but this has other characteristics which make it less suitable for business calculations). Now suppose a component receives an IP and tries to access a currency field in it based on its compiled-in knowledge of the IP's layout. If the compiled code has been told that the field has 2 places of decimals, that's what the component will "see". So we can now do arithmetic with the number, display it in the right format (if we know what national currency we are dealing with), and so forth. But note that the layout of the IP is only defined in the code - the code cannot tell where the fields really start and stop. So we have a sort of mutual dependency: the only definition of the data is in the code, and the code is tightly tied to the format of the data. If you want to decouple the two, you have to have a separate description of the data which various routines can interrogate, which can be attached to the data, independent of what routines are going to work with that data. If this is powerful enough, it will also let you access older forms of your data, say, on an old file (the "legacy data" problem).

Apart from format information, you also have to identify what type of data a field contains. For instance, the hexadecimal value

```
019920131F
```

might be a balance in a savings account ($199,201.31), but it could also be a date (31st January, 1992). Depending on which it is, we will want to perform very different operations on it. Conversely a function like "display", which applies to both data types, will result in very different results:

```
$199,201.31
```

versus

```
31st January, 1992
```

Our traditional HLLs will see both data types as FIXED DECIMAL (or COMPUTATIONAL-3). Again, only the program knows which kind of data is in the field (by using the right operations). There is also the issue of which representation is being used. We have all run into the problem of not knowing whether 01021992 is January 2 (American convention) or February 1 (British convention). So we have to record somewhere which digits represent the day, which the month and which the year. Thus a complete description of our field has what we might call "base type" (signed packed decimal), length (and perhaps scale), domain and representation. Some systems use a standard representation for "internal" data, or at least a format which is less variable than "external" formats, but the "data about data" (metadata) items which I have just described are pretty basic. (Base type could be treated logically as part of the more general domain information, but it turns out to be useful for designing such functions as "dumb" print processes). In some systems, domain is referred to as "logical type", and representation as "physical type".

Let us now look at another example of the pitfalls of programs not knowing what kind of data they are working with. Suppose that you have coded the following PL/I statement in a program by mistake:

```
NET_PAY = GROSS_PAY * TAX;
```

The compiler isn't going to hesitate for a microsecond. It will blithely multiply two decimal values together to give another decimal value (assuming this is how these fields are defined), even though the result is perfectly meaningless. Remember: computers do what you tell them, not what you mean! A human, on the other hand, would spot the error immediately (we hope), because we know that you can't multiply currency figures together. The compiler knows that the result of multiplying two numbers each with 2 decimal places is a number with 4 decimal places, so it will carefully trim off the 2 extra decimal places from the result (maybe even rounding the result to the nearest cent). What it can't do is tell us whether the whole operation makes any sense in the first place!

One other (real world) problem with currency figures is that inflation will make them steadily larger without increasing their real value. If 11 digits seemed quite enough in 1970, the same sort of information may need 13 or 15 digits in 1992. It is disconcerting to have your program report that you mislaid exactly a billion dollars (even though it is usually a good hint about what's wrong)! It would be nice if we could avoid building this kind of information into the logic of our applications. If we had an external description of a file, we could either use this dynamically at run-time, or convert the data into some kind of standard internal format - with lots of room for

inflation, of course! The other place where this affects our systems in in screen and report layouts. As we shall see, these are also areas where it makes a lot of sense to hold descriptions separately, and interpret them at run-time. What we don't want to do is have to recompile our business systems every time some currency amount gets too big for the fields which hold it.

This is another legacy of the mathematical origins of today's computers - everything is viewed as a mathematical construct - integers, real numbers, vectors, matrices. In real life, almost everything has a dimension and a unit, e.g. weight, in pounds or kilograms, or distance, in miles or kilometres. If you multiply two distances together, you should get area (acres or hectares); three distances give volume, in cubic centimetres, bushels or litres. Currency amounts can never be multiplied together, although you can add and subtract them. You could also theoretically convert currency from dollars to pounds, or francs to marks, but this is a little different as it would need access to up-to-date conversion rate tables, and it was recently pointed out to me that banks usually like to charge for converting from one currency to another, so that it's not just a simple matter of multiplying a currency amount by a conversion factor [although we built just such a facility for a recent e-business application]. Dates can't even be added, although you can subtract one date from another. There is a temptation with dates to just convert them into a canonical form (number of days from a reference date - for instance, Jan. 1, 1800) and then assume you can do anything with them. In fact, they remain dates, just represented differently, and you still can't add them.... On the other hand, you can do things like ask what day of the week a date falls on, what is the date of the following Monday, how many business days there are between June 30 and August 10, etc.

We have used the term "representation" quite a lot so far. Some of the above complexities come from confusing what the data is with how it is represented. The data is really a value, drawn from a domain (defined as a set of possible values). We really shouldn't care how the data is represented internally - we only care when we have to interface with humans, or a file is coming from another system. However, we also have to care every time we interface with current higher level languages. The requirements for interfacing with humans involve even more interesting considerations such as national languages and national conventions for writing numbers and dates, which should as far as possible be encapsulated within off-the-shelf subroutines.

Multilingual support is an increasingly important area. Some Asian languages involve double-byte coding, which differs from machine to machine, as well as from language to language. Computer users no longer feel that they should have to learn English to use an application, although most programmers are still willing to do so! This attitude on the part of programmers still sometimes laps over into what they build for their customers, but the more sophisticated ones know that we are living in a global market, and that computers have to adapt to people, rather than the other way around. In many ways, Canada has been at the forefront of these changes, as it is an officially bilingual country, and French Canadians have historically been very insistent, and rightly so, that their language be written correctly!

If we separate the representation from the content of the data, we can look at the variety of possible representations for any given chunk of data, and consider how best to support conversion from one to another. I once counted 18 different representations of dates in use in our shop! These are all inter-convertible, provided you are not missing information (like the century). I imagine a lot of retired programmers are going to be called back into harness around the year 2000, converting programs with 6-digit dates to make them able to cope with the 21st century! [Yup! We were!]

As we said above, the representation of data inside the machine doesn't really concern us. It is of interest when we are talking about external uses of that data (being read by humans, being written to or read from data bases, or being processed by HLLs). In some ways, this resembles the Object-Oriented view. However, when displaying data (e.g. numeric fields) we found that we needed "global" information to control how the data should be presented, such as:

- currency symbol (whether required and, if so, which)
- whether currency symbol is floating or fixed
- separators between groups of three digits (whether required, and, if so, what symbol)
- separators between integer part and fractional part (what symbol)
- whether negative values should be indicated and, if so, how (DR, preceding -, following -, etc.)

In addition, these options usually come in "layers": there may be an international standard, a national standard, and a company standard, and a particular report may even use one or more such representations for the same domain, e.g. amounts with and without separators. Today it is not enough just to provide one conversion facility in each direction. Representations occur at the boundaries between responsibilities and, I believe, require sophisticated multi-level parametrization.

Given that we have to work with existing HLLs [this was pre-Java!], the best we can probably do is to describe fields and use smart subroutines as much as possible for all the conversion and interfacing logic. This will let us implement all known useful functions, but it cannot prevent illegal ones. Object-oriented and some of the newer High Level Languages with strong typing are moving in this direction, but the older HLLs do not provide any protection. I suspect we have to go further, though, as some form of dimensional anlysis will probably be necessary eventually. Hardware designs may emerge which take into account the needs of business users, and when this happens, we will not have to have these drastic conversions back and forth between such different paradigms.

So far we have only talked about static descriptions of data. DFDM also had another type of data description which proved very useful, which we called "dynamic attributes". Dynamic attributes

were also a form of metadata, but were attached to IPs, as well as to their descriptions. The first example of this that we came up with was the "null" attribute. We took this term from IBM's DB2, in which a column in a table may have the attribute of "may be null". This means that individual fields in this column will have an additional bit of information, indicating whether or not the particular value is null or not, meaning either "don't know" or "does not apply". Some writers feel that these two cases are different, and in fact the latter may be avoided by judicious choice of entity classes, but the former is certainly very useful. In DFDM interactive applications, we often used "nullness" to indicate fields which had not been filled in on a panel by the end-user.

The "null" attribute also works well with another dynamic attribute which we also found useful: the "modified" attribute. Suppose that an application panel has a number of fields on it, some of which do not have values known to the program. It is reasonable to display "null" fields as blank, or maybe question-marks. If the user fills one or more fields in, their attributes are changed to "modified" and "non-null". This information can then be used by the application code to provide user-responsive logic. We found that this kind of logic often occurred in the type of application which is called "decision assist": here you often see screens with a large number of fields and it becomes important to know which ones have been modified by the user.

Many applications encode null as a "default" value, e.g. binary zeros, but there are a number of formats which do not have an unused value, e.g. binary, so how do you tell whether you have zero eggs, or an unknown number? Does a blank street name and number in an address mean that we don't know the house-owner's full address, or that she lives in a rural community, where the mailman knows everyone by name? Also, we saw no advantage to confusing the idea of null and default - what is an appropriate default number of eggs?

Where DB2 has specific handling for the "nullness" attribute only, DFDM generalized this idea to allow you to attach any kind of dynamic attribute data to any field of any IP, e.g. "null" and "modified", but also "colour", "error number", etc. Since we felt we couldn't predict what kinds of dynamic attribute data we might want to attach to the fields of an IP, we built a very general mechanism, driven by its own descriptor (called, naturally, a Dynamic Attribute Descriptor or DAD). It allowed any number of attributes to be attached to each field and also to the IP as a whole. Thus, we had a "modified" attribute on each field, but, for performance reasons, we had a "modified" attribute on the IP as a whole, which was set on if any fields were modified.

"Null" and "modified" are of course Boolean, but we allowed binary or even character dynamic attributes. One character-type dynamic attribute which we found very useful for interactive applications was "error code". Suppose an editing routine discovered that a numeric field had been entered by the user incorrectly: it would then tag that field with the error code indicating "invalid numeric value". Any number of fields could be tagged in this way. When the IP containing the screen data was redisplayed, the display component would automatically change all the erroneous fields to some distinctive colour (in our case yellow), position the cursor under

the first one and put the corresponding error message in the message field of the screen. Without leaving the component (under IBM's ISPF environment), the user could then cycle through all the error fields, with the correct error message being displayed each time. This was really one of the friendliest applications I have ever used, and it was all managed by one reusable component which invoked ISPF services (under IBM's IMS you can do the same thing but it would probably take a set of components working together).

Some writers have objected to the "null" attribute on the grounds that it introduces 3-valued logic: yes, no and don't know. Our experience was that, in practice, it never caused any confusion, and in fact significantly reduced the complexity of the design of our end user interfaces.

A last point has been suggested by our study of Object-Oriented Programming: IP types are often related in a superclass-subclass relationship. This comes up frequently in file handling: one may know that one is dealing with, say, cars, but not know until a record is read what kind of car it is. It would be very nice to be able to attach a "car" descriptor to each record as it is read in, and then "move down" the class hierarchy for a given record, based on some indicator in the record. This is in turn related to the question of compatibility of descriptors: what changes are allowed between descriptors? It seems reasonable to be able to change a generic car into a Volkswagen as one gains more information about it, but not a car into, say, a beetle [note - this is similar to the OO class/superclass relationship, except that it is not acceptable to have to modify code as new models of car are produced]. In the later versions of DFDM we decided that IP lengths should be completely specified by IP type - i.e. if you created an IP with a given type, the length would be obtained from the type descriptor, and couldn't be changed. The question then arises: if you change a car into a Pontiac (move from superclass to subclass), does the IP become longer, or does an "unstructured" portion of the IP become structured? Let's leave this as an exercise for the reader!

# Chap. XII: Tree Structures

Up to this point, I have been talking about single IPs travelling through networks, like cars and buses in a system of highways. I have talked about how streams of IPs can be treated as higher-level entities, and how these can in turn be given more complex structures by the use of "bracket" IPs. I hope I have shown that, using these concepts, quite complex applications can be handled in a straightforward manner.

However, what if we want to build more complex structures and move them through the network as single units? Streams take time to cross a connection, and you may want a whole data structure to be sent or received at a single moment in time. Since, as we said before, only the handles travel through the net, it is just as easy for an IP containing one bit of data to be transferred from process to process as one containing a megabyte, so why not allow complex structures to move as a unit? It turned out that there was a natural analogue to this idea in real life (which always tends to reassure us that we are on the right track!). We mentioned before the idea that IPs are like memos - you can dispose of one in one of three ways: you can forward it (send), discard it (drop), or keep it (using the various methods of holding onto IPs, e.g. stacking, saving on disk, etc.). Well, of course there is a fourth thing you can do with a memo (no, not wad it up and throw it at a neighbour) - clip it to another piece of paper, and then do one of the previous three things with the resulting composite memo. Make that four things, actually, since you could clip the composite memo to another piece of paper, or to another composite memo, and so on.

Just as the composite memo can be sent or received as a unit, so the structure of linked IPs, called a tree, can be sent or received as a single object. Once a process has received the tree, it can either "walk" it (move from IP to IP across the connecting links), completely or partially disassemble it, or destroy it. For instance, the receiver might walk the tree looking for a particular kind of IP, and, for each one it finds, it could unlink it and dispose of it in one of the standard ways. The following diagram shows two processes, one assembling trees and one disassembling them again:

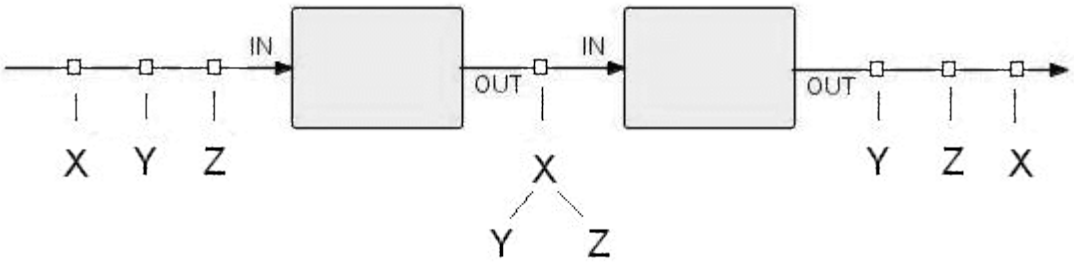



Figure 12.1

*A* converts a series of three IPs of different types and sizes into a tree of IPs, and *B* disassembles the trees and outputs the component IPs (in a different order).

In all implementations of this concept so far, no node could descend from more than one parent node, and we did not allow any loops - much like a real live tree! I do not believe we lost much expressive power by doing so. The main reason for doing this is to allow FBP's ownership and disposal rules for IPs to work.

You will probably have realized that we don't need any special mechanisms as long as the tree is assembled and disassembled within one activation of one process (activation is described in detail in the chapter on Scheduling Rules). In fact you could read in a set of IPs and build an array of pointers for sorting, say - this is in fact how a number of components work. It is only when a tree has to be passed from one activation to another, or from one process to another, that the IP disposition rules have to be taken into account. Thus, in the picture above, process *A* receives (or creates) three IPs, assembles them into a single tree of 3 IPs and sends it out - in this case *A* starts with an "owned IP count" of 3, and reduces it to 0 by attaching 2 IPs to the root (count is now 1) and then sending the tree as a whole out (count of 1 goes to 0). *B* receives one IP (count goes from 0 to 1), and detaches the attached IPs (count goes from 1 to 3). These detached IPs must then of course be disposed of by the normal rules. If we allowed any violations of the above rules about tree shape, we wouldn't be able to map so easily between trees and streams.

In one application we had a striking example of how useful trees can be in the FBP environment: in a batch banking application, bank accounts were represented by complex sequential structures on tape. Each account record consisted of an account header, followed by a variable number of different trailer records belonging to a large number of different types, e.g. stops, holds, back items, etc. The problem was that, most of the time one could just process these sequentially, but sometimes processing later in the stream resulted in changes which should be reflected earlier in the stream. For example, an interest calculation, triggered by a particular type of trailer record, might require the account balance in the header record to be updated. You could always hold on

to the header and put it out later, but then it would have to collated back into its correct position. And anyway there was quite a lot of this "direct" access going on, including adds and deletes of trailer records. Using conventional programming, this application became fiendishly complex because all the logic had to be coordinated from a timing point of view, and there were timing conflicts between when things were required and when they became available! (We know this because originally it was coded using conventional logic, and it was very complex!) We were also talking about large volumes of these structures - we had to be able to process about 5,000,000 every night. When we did this application using FBP, we realized that we could implement this application very simply and naturally by converting each sequential record into a tree structure. Once the tree had been built, "direct access" type processing could jump from one IP type to another within the tree structure, add or delete IPs, etc., and then the whole thing could be converted back to linear form when we were finished. This solution turned out to be simple to understand, easy to code and easy to maintain.

Now let's draw a picture of a simple tree of four IPs:

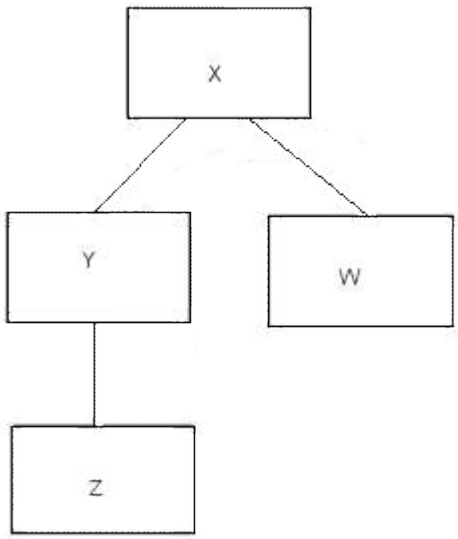



Figure 12.2

The top box represents the "root" IP, and all the other IPs are descended from it directly or indirectly. In the above diagram (mixing our metaphors a bit):

- X is the root
- Y and W are the daughters of X

- Z is the daughter of Y
- Z and W are leaves (terminal IPs)
- Y is a non-terminal IP

AMPS and the earlier versions of DFDM took the approach that an attached IP required a pointer field to be specified as part of the mother's data. This meant that a tree structure could only be traversed using languages which supported pointers, e.g. PL/I, C, etc., but it did have the advantage that, using these languages, we did not have to have any explicit tree traversal services. In fact, we only needed an "attach" service and a "detach" service. The more fundamental problem was that the layout of a given IP had to allow for the maximum number of children that it might have - thus, in the above diagram, X would have to provide (at least) two pointers, and Y one. Schematically (with the pointers shown as 'x's):

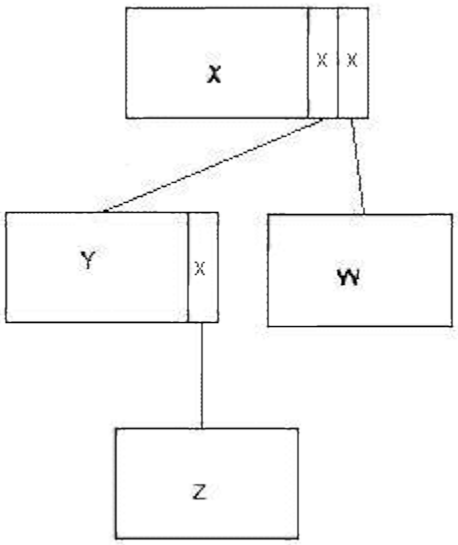



Figure 12.3

However, if Y and W could be conceptualized as a single list (rather than two different kinds of relationships) - say an employee's children, you could build "glue" IPs and use them to look after the linkage. The above diagram would then become:

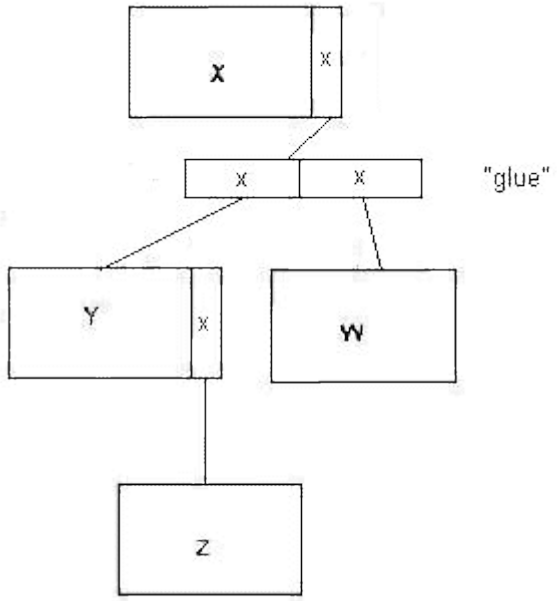



Figure 12.4

Along these lines, this simple mechanism lets you build a wide variety of structures by creating special IP types which contain various arrangements of pointers. For example, you could build a chain by just adding one pointer field to each IP in the chain, which can be used for the successor IP if there is one. You could build LISP-like lists by having "car" and "cdr" pointers on each IP, or even only on certain reserved IP types ("glue"). My colleague Charles Douglas came up with the neat analogy that, when you use a paper clip to attach two memos together, the paper clip takes up some room on the paper!

For the Japanese DFDM product, our group felt that you should not have to preplan all the list structures an IP might have attached to it. For instance, an employee might have lists attached to him or her showing children, courses attended, departments worked for, salary history, etc., and it would very nice if this information could be added incrementally, without having to change the descriptions of participating IPs. We therefore introduced the idea of named chains, any number of which could be added to an IP, without that IP requiring any changes to its description. An employee IP could for instance have the following chains attached to it: *CHILDREN, SALARY_HIST, COURSES*, etc. We then of course needed to provide traversal services, and we in fact built some fairly powerful services, e.g. add an IP to a named chain (it is created if it does not yet exist), get next chain (so you could walk the chains without knowing their names), get the first IP of a named chain, get next IP in a chain, detach an IP from a chain, and so forth. You

could attach a chain to an IP, or an IP to a chain, but not a chain to a chain, or an IP to an IP. These trees thus had a less uniform structure (chains alternating with data IPs), but we felt that this still provided a powerful paradigm, and a less "programmer-dependent" approach to tree manipulation and traversal. An employee IP with one child, Linda, and who has taken two courses might look like this:

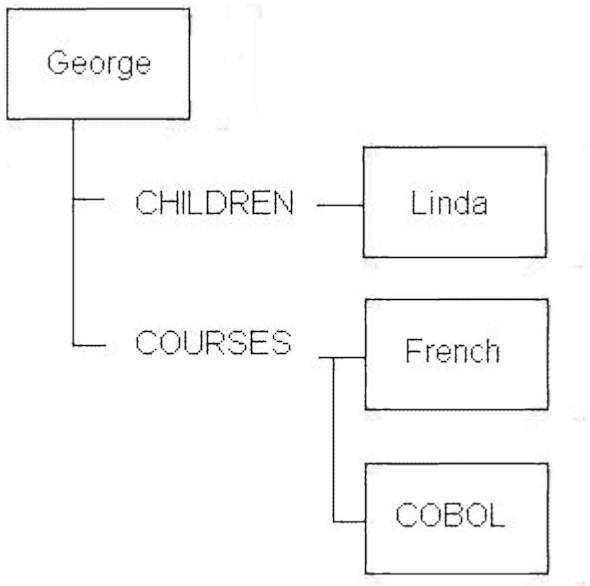



Figure 12.5

Along the same lines as having to dispose of IPs explicitly, we had to put certain constraints on how trees can be disassembled. This is hinted at above when we talked about "direct" and "indirect" descent. The root is owned directly by whichever process has just received it. IPs which are chained to that root are therefore owned indirectly by the same process (they are not owned directly by anyone, except possibly the root IP). You can only send or drop an IP you own directly - you have to detach a chained IP first, then send it or drop it. A process can however chain another IP (which it must own directly) onto an IP which it only owns indirectly, but that's the only service it can reference it with, apart from looking at it! We also added logic to check that a process did not try to chain a root IP onto one of its own descendants - that would have been allowed by the other rules, but would result in a closed loop!

If the above sounds complicated, try this analogy: suppose you have just cut down a tree, and you want to dispose of it. It seems reasonable that, before you burn a branch, you should cut it off first. We could arrange that burning a branch without cutting it off first would "cauterize" the point of attachment, but it seems that this would add unnecessary complexity. Conversely, we

will allow you to add leaves and branches to the tree, but only if they are not connected to another one! Also, you are not allowed to attach things to a tree in such a way that part of the tree becomes a closed loop!

One other point about trees is that hierarchic tree structures, no matter how they are implemented, can easily be converted into nested substreams like the ones described in Chapter 11. For instance, the tree shown in Figure 12.2 can be "linearized" as follows:

```
<X <Y Z> W>
Figure 12.6
```

where the convention is that the IP following a left bracket is the "mother" of the other IPs at the same level of bracket nesting. This kind of transformation will be familiar to LISP users. In fact we have just shown a LISP "list of lists".

In the "chain" implementation described above, we would have to capture chain identifiers, so we can just add chain information to the left brackets, as we did in Chapter 9. Thus the tree shown in Figure 12.5 might look as follows after linearization (as in earlier chapters, the "group name" is shown as the data part of open and close bracket IPs):

```
IP type           data
<                employees
employee         George
<                children
child            Linda
>                children
<                courses
course           French
course           COBOL
>                courses
>                employees
Figure 12.7
```

This can then easily be converted back into tree format if desired. It also of course corresponds quite well to various data base approaches: "children" and "courses" could be different segment types in an IBM DL/I data base. In IBM's DB2 we could make "employee", "children" and "courses" different tables, where "employee" is the primary key of the "employee" table, and a foreign key of the other two. [Of course this maps perfectly onto XML!]

One last point: although we have stated several times our belief that IPs should be disposed of explicitly, it turns out to be very useful to be able to discard a whole tree at a time. The tree therefore has to have enough internal "scaffolding" to allow the 'drop' service to find all the chains and attached IPs and discard them. The later versions of DFDM needed this anyway, so that they could provide services like 'locate next chain', but this ability to drop a whole tree turned out to be important even when we had no traversal services. Although at first it seemed

that applications would always know enough about the tree structure to do the job themselves, we developed more and more generic components which understood about trees generally, but not about specific tree structures. This facility perhaps most closely resembles the "garbage collection" facility of object-oriented and list-oriented languages.

# Chap. XIII: Scheduling Rules

So far, we have talked about processes running asynchronously, but have not discussed how FBP software manages this feat. This is a key concept in FBP and we need to understand it thoroughly if we are to design reliable structures, and debug them once they have been built! Although this subject may appear somewhat forbidding, you will need to grasp it thoroughly to understand how this kind of software works. After a bit it will recede into the background, and you will only need to work through it consciously when you are doing something complex or something unexpected happens. Of course you are at liberty to skip this chapter, but, if you do, you will probably find some of the later chapters a little obscure!

Let us start by looking at the following component pseudo-code (from a previous chapter):

```
receive from IN using a
do while receive has not reached end of data
    if c is true
        send a to OUT
    else
        drop a
    endif
    receive from IN using a
enddo
```

You will be able to see that the job of this component is to receive a stream of IPs and either send them on or destroy them, depending on some criterion. As we said above, this code must be run as a separate process in our application. We will use the term "component" when talking about the code; "process" when talking about a particular use of that code.

Before, we were looking at processes and components strictly from a functional point of view. Now instead let's look at a component as a piece of code. Clearly it is well-structured: it has a single entry point and a single exit (after the "enddo"). Once it gains control, it performs logic and calls subroutines until its input stream is exhausted ("end of data" on IN). It is therefore a

well-formed subroutine. Subroutines have to be called by another routine (or by the environment), and at a particular point in time. So.... what calls our Selector component, and when? The what is easy: the Selector component is called by the software which implements FBP, usually referred to as the "scheduler". The when is somewhat more complicated: the answer is that a component is called as soon as possible after a data IP arrives at one of its input ports, or just as soon as possible if there are no input ports.

"As soon as possible" means that we do not guarantee that a process will start as soon as there is data for it to work on - the CPU may be busy doing other things. Normally this isn't a problem - we want to be sure that our process has data to work on, not the reverse! If we really needed very high responsiveness, we would have to add a priority scheme to our FBP software. So far, the only implementation of the FBP concepts that I am aware of that implemented such a priority scheme was a system written in Japan to control railway electric substations (Suzuki et al. 1985). This software built on the concepts described in my Systems Journal article, and extended it to provide shared high-performance facilities. None of the implementations I have been involved with have needed this kind of facility.

The other possible start condition is what one would expect if the process has no input connections. In this case, one expects the process to be "self-starting". Another way of looking at this is that a process with at least one input connection is delayed until the first data IP arrives. If there are no input connections, the process is not delayed. Again, the process will start some time after program start time when the CPU is available.

Note that THREADS Initial Information Packets (IIPs) do not count as input connections for the purposes of process scheduling - when a process starts is determined by the presence or absence of connections with upstream processes. IIPs are purely passive, and are only "noticed" by the process when it does a receive on an IIP port element.

Now we have started our process - this is called "activation", and the process is said to be "active". When it gives up control, by executing its "end" statement, or by explicitly doing a RETURN, GOBACK, or the equivalent, it "deactivates", and its state becomes "inactive".

Now remember that our component kept control by looping until end of data. Now suppose our component doesn't loop back, but instead just deactivates once an IP has been processed. The resulting pseudo-code might look like this:

```
receive from IN using a
if c is true
    send a to OUT
else
    drop a
endif
```

We talked about this kind of component in Chapter 9. They are called "non-loopers". The logic

of non-loopers behaves a little differently from the preceding version - instead of going back to receive another IP, it ends (deactivates) after the "endif". A consequence of this is that the process's working storage only exists from activation to deactivation. This means that it cannot carry ongoing data values across multiple IPs, but, as we saw in Chapter 9, the stack can be used for this purpose. A non-looper becomes "inactive" after each IP.

What happens now if another data IP arrives at the process's input port? The process is activated again to process the incoming IP, and this will keep happening until the input data stream is exhausted. The process is activated as many times as there are input IPs. The decision as to when to deactivate is made within the logic of the component - it is quite possible to have a "partial looper" which decides to deactivate itself after every five IPs, for example, or on recognizing a particular type of IP. This "looping" characteristic of a component is referred to as its "periodicity".

Let us consider an inactive (or not yet initiated) process with two input ports: data arrives at one of the ports, so the process is activated. The process in question had better do a receive of the activating IP before deactivating - otherwise it will just be reactivated. As long as it does not consume the IP, this will keep happening! If, during testing, your program just hangs, this may be what is going on - of course, it's easy enough to detect once you switch tracing on, as you will see something like this:

```
Process_A Activated
.
.
Process_A Deactivated with retcode ...
Process_A Activated
.
.
Process_A Deactivated with retcode ...
Process_A Activated
.
.
Process_A Deactivated with retcode ...
Process_A Activated
.
```

and so on indefinitely!

Our group discussed the possibility of putting checks into the scheduling logic to detect this kind of thing, but we never reached a consensus on what to do about it, because there are situations where this may be desirable behaviour - as long as the activating IP does get consumed eventually. We did, however, coin a really horrible piece of jargon: such a component might be called a "pathological non-depleter" (you figure it out)! And besides, it's really not that hard to debug...

Returning to IIPs, we have said that in THREADS processes read in IIPs by doing a receive on their port. If they do a receive again from the same port within the lifetime of the process, they get an "end of data" indication. This means that a component can receive an IIP exactly once during the lifetime of the process (from invocation to termination). If a component needs to hold onto the IIP across multiple activations, it can use the stack (described in Concepts) to hold the IIP, either in the original form in which it was received or in a processed form.

So far, we have introduced two basic process states: "active" and "inactive". We now need terms for the very first activation and the very last deactivation of a process: these are called "initiation" and "termination", respectively. Before initiation, the process doesn't really exist yet for the FBP software, so initiation is important as the scheduler has to perform various kinds of initialization logic. Termination is important because the scheduler must know enough not to activate the process again. Termination of a process also affects all of its downstream processes, as this determines whether they in turn are terminated (a process only terminates when all of its upstream processes have terminated).

Processes may thus be thought of as simple machines which can be in one of a small number of different "run states". The four main run states are the ones we have just described:

- not yet initiated
- terminated
- active
- inactive

"Not yet initiated" is self-explanatory - it means that the process has never received control. All processes start off in this state at the beginning of a job step or transaction.

"Terminated" means that the process will never receive control again. This can only happen if all of a process's upstream connections have been closed - each of the input connections of a process can be closed explicitly by that process, or it will be closed automatically if all of the processes feeding it have terminated.

The underlying idea here is that a process only becomes terminated if it can never be activated again. It can never be activated again if there is nowhere for more data IPs to come from. Note that, while a component's logic decides when to deactivate, termination is controlled by factors outside of the process. There is one exception to this, however: a component can decide not to be reactivated again, e.g. as a result of some error condition. It does this in both DFDM and THREADS by returning a particular return code value (or higher) to the scheduler. [This has not been implemented in JavaFBP or C#FBP.] Earlier versions of FBP allowed one component to bring down the whole network, but in recent years we felt that this was not conceptually sound (parts of the network could be on different computers), so now a process can only terminate itself

- no process can terminate another. There is a different way that a component can decide to terminate itself, and that is by closing its input ports. Some processes don't have input ports, but, for those that do, this has the same effect as terminating with a high return code.

A simple example will show why this needed: suppose you have a Reader process which is reading a file of a few million records, and a downstream process crashes: under normal FBP rules, the Reader keeps reading all the records, and sending them to its output port. As each send finds the output port closed, the Reader has to drop the undeliverable IP. So it has to read all the records, requiring a few million each of "create", "send" and "drop". Instead, it is much better to bring the Reader down as quickly as possible, so it can stop tying up time and resources. Readers are therefore normally coded so that, the first time they find the output port closed, they just deactivate with a high return code. This takes care of any necessary housekeeping, and eventually the whole network can close down. By the way, this practice makes sense for all components, not just long-running ones - it is good programming style for components always to test for unsuccessful sends. If this condition is detected, they must decide whether to continue executing, or whether to just close down - this usually depends on whether the output port is related to the main function of the component, or whether it is optional.

When all the processes in a network have terminated, the network itself terminates. Now, it is possible for one process to block another process so that the network as a whole cannot come down gracefully. This is called a "deadlock" in FBP and is described in some detail in a later chapter. This is however a design problem, and can always be prevented by proper design. If the network is properly designed, it will terminate normally when all of its processes have terminated, and all resources will then be freed up.

In Chapter 7, we talked about various kinds of composite components. DFDM's dynamic subnets and the composite components of FPE had the ability to revive terminated processes. In that chapter we explained why this ability is necessary, and we shall run into it again in Chapter 20, when we talk about Checkpointing. Revived processes essentially go from the "terminated" state back to the "not yet initiated" state.

If we had a separate CPU for each process, the above-mentioned four states would be enough, although a component waiting to send to a full connection, waiting to receive from an empty one, or waiting on an external event, would have to spin waiting for the desired condition. To avoid this, we have introduced a suspended state, so we can split the active state into "normal" and "suspended", resulting in five states:

- not yet initiated
- terminated
- active normal
- active suspended

- inactive

At this point I would like to stress the point that a given process can only be in one of these states at a time, and, when suspended, a process can only be waiting for a single event. While you will occasionally feel that this is too much of a restriction, we deliberately made this decision in order to make an FBP system easier to visualize and work with. At various times in the development of FBP systems, we were tempted to allow a process to wait on more than one event at a time, but we always found a way round it, and never needed to add this ability to our model. Suppose you want to have a process, *P*, which will be triggered by a timer click or by an IP arriving at an input port, whichever comes first: the rule about processes having a single state suggests that you might need two processes, one waiting on each event type. One possible solution, however, is to have one process send out an IP on each timer click, and then merge its output stream with the IPs arriving from another source, resulting in a single stream which is then fed to P. The overhead of the extra processes is outweighed in our experience by the reduction in complexity of the mental model and the consequent reduction in software complexity and improved performance. Here is a picture of the resulting network:

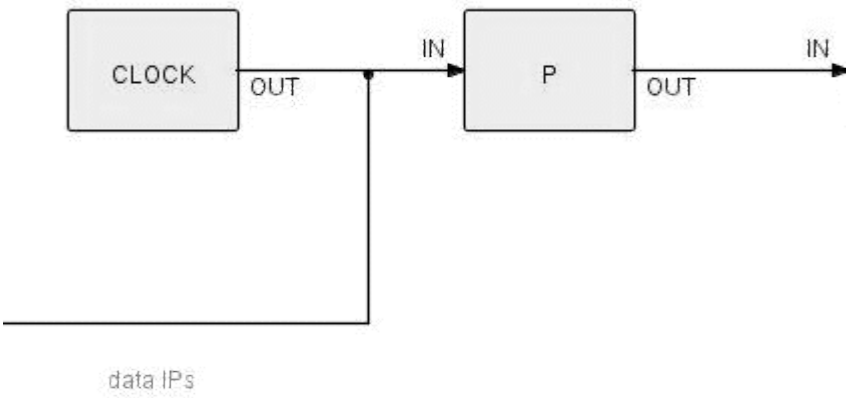



Figure 13.1

Up to now, we have assumed that all ports are named. Not all ports need to be known to the components they are attached to: sometimes it is desirable to be able to specify connections in the network which the processes themselves don't know about. These are especially useful for introducing timing constraints into an application without having to add logic to the components involved. In DFDM we used port 0 for various functions of this kind, but this idea has been generalized into what we call automatic ports, to reflect the idea that their functioning is not under the component's control.

Consider two processes, one writing a file and one reading it. You, the network designer, want to interlock the two components so that the reader cannot start until the writer finishes. To do this, you figure that if you connect an input port to the reader, the reader will be prevented from starting until an IP arrives on that port (by the above scheduling rules). On the other hand, readers don't usually have input ports, and if you add one, the reader will have to have some additional code to dispose of incoming IPs, looking something like this:

```
if DELAY port connected
    receive from DELAY port
    discard received IP
endif
```

Similarly, you would also have to add code to the Writer at termination time to send an "I'm finished" IP to an appropriate output port.

Now, to avoid having to add seldom used code (to put out and receive these special signals) to every component in the entire system, the software should provide two optional ports for each process which the implementing component doesn't know about: an automatic input port and an automatic output port. If the automatic output port is connected, the FBP scheduler closes it at termination time.

The automatic input works like this: if there is an automatic input port connected, process activation is delayed until an IP is received on that port, or until the port is closed. This assumes that no data has arrived at another input port.

Here is a picture of the Writer/Reader situation:

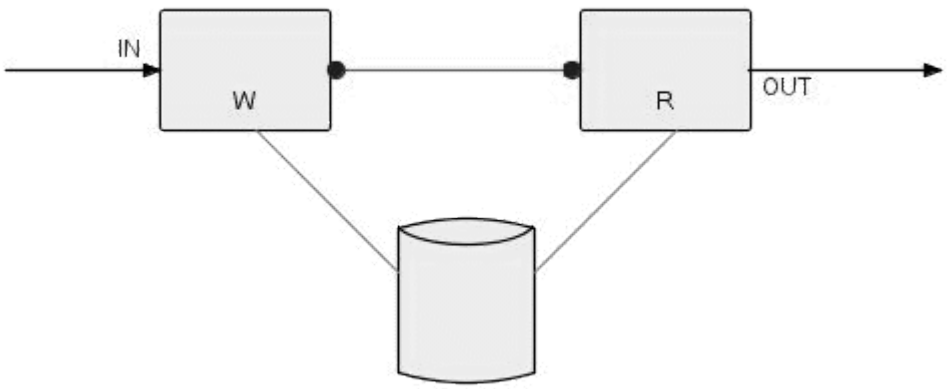


Figure 13.2

where the solid line indicates a connection between W's automatic output port and R's automatic input port. The solid circles at each end of the line indicate automatic ports. Since W only terminates when it has written the entire file, we can use the automatic output signal to prevent the Reader from starting too early. An automatic port need not only be connected to another automatic port - it can always be connected to a regular port, or vice versa. Any IP that has been received (by the scheduler) at an automatic input port is automatically discarded (better not use any important data for this job, unless you have taken a copy)!

Here is a network where the automatic output signal gates an IP from another process. Assuming that C receives from I1 before it receives from I2, then C will not process the input at I2 until A has terminated. If we made I1 automatic, we would have essentially the same effect, except that C would not have to do a receive, but conversely, it would not have the option of processing the input at I2 first.

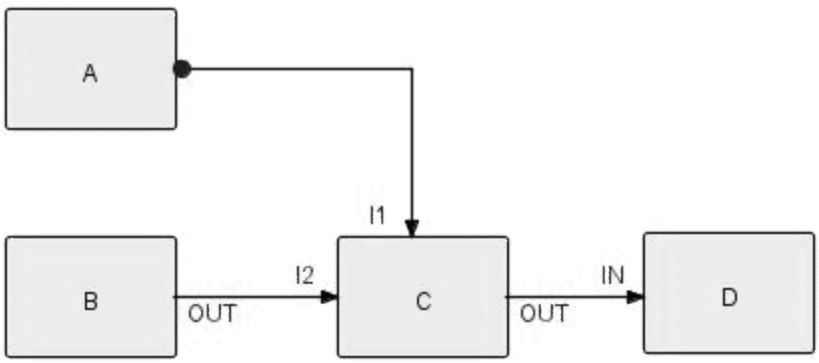



Figure 13.3

There is another situation which is similar to the automatic output port: all output ports of a component, whether the component "knows" about them or not, will present end of data when that component terminates. This can also be used to defer events until one or more processes have terminated (see Chapter 20). [This was true in THREADS - this situation is not allowed in JavaFBP or C#FBP.]

One last topic we should mention is the problem of the "null" stream: in DFDM a component receiving a null stream (a stream with no data IPs) was invoked anyway. This logic, while perhaps consistent with the regular scheduling rules, tended to increase run-time costs. In one interactive application, we found that 2/3 of all the processes were error handlers, and so should really never get fired up if no errors occurred. If they were fired up, as in the normal DFDM case, the added overhead became quite expensive - especially since most of the processes were written in HLLs, so the run-time environment of every process had to be initialized separately.

We therefore introduced [in the next implementation] the ability to change this behaviour for a whole network. But then we had the problem of Writers and Counters. Consider a Writer component writing a disk file: when it receives a null stream, you want the Writer to at least open and close its output file, resulting in an empty data set. If it doesn't do this, nothing gets changed on the disk, and another job reading that file would see the data from the previous run! Counter components which generate a count IP at end of data have a similar problem - how can they generate a count of zero if they are never invoked? So then we had to add the concept of components which are "end of stream sensitive" (components which behave differently in these two modes - most do not need to), which could in turn be overridden by a "suppress end of stream processing" indication in the network (in case you really didn't want this special logic for a particular run).... Old systems which have been in use for a number of years tend to develop layers upon layers of incrustations, rather like barnacles on a boat....

THREADS defaults the other way from DFDM, but for the cases where you want a process receiving a null stream to be activated, THREADS uses the concept of "must run at least once". This is an attribute of the component, not of the process, and means simply that the component must be activated at least once during each run of the network. So Writer and Counter components can do their thing, even when receiving a null data stream. [This seems much simpler! The Java and C# implementations also work this way.]

# Chap. XIV: Loop-Type Networks

All of the network shapes we have encountered so far have been of the kind we call "batch", and have generally had a left-to-right flow, with IPs being created on the lefthand side and disposed of on the right of the network. Sometimes we need to use a different kind of topology, which is a **loop-type** network. Several of the later chapters contain examples of this topology, so it is worthwhile spending some time talking about this type of network at a general level. Many networks will in fact be a mixture of the two types, but, once you understand the underlying principles, they won't present any problems.

Here is a very simple example of a loop-type network:

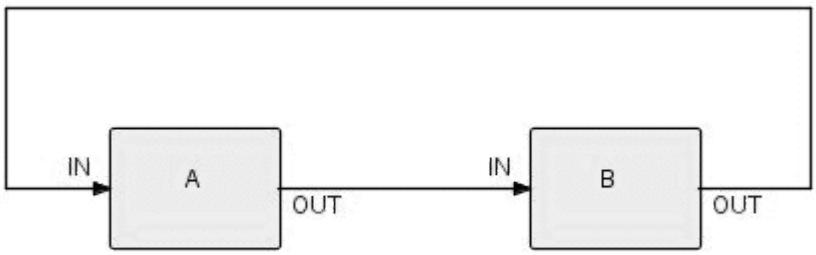



Figure 14.1

The first question we need to answer about this type of network is: how does it get started? You may remember that the only processes which get started automatically are those with no input connections (IIPs don't count). If you look at Figure 14.1, you will see that there are no processes which have no input connections [double negative intended!]! *B* has an input connection coming from *A*, but *A* has an input connection coming from *B*! Although on occasion we have been

tempted to relax the restriction about which processes can start, the simplest thing to do is just to add an extra process which has no input connections and then use it to start *A* or *B*. So the picture now looks like this:

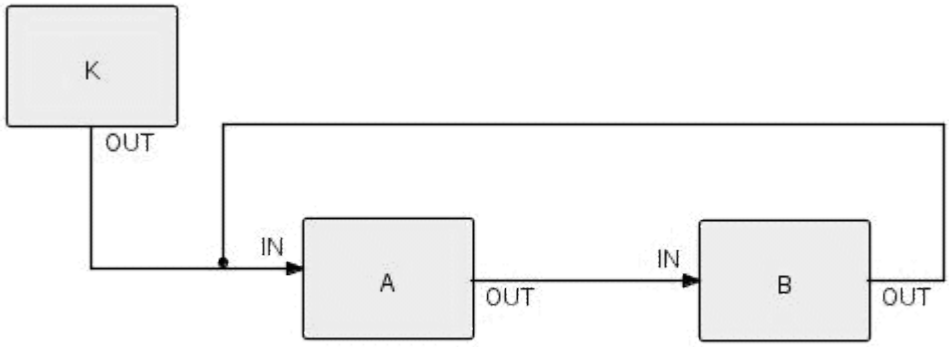



Figure 14.2

where *K* is the starter ("kicker") process, that emits a single packet containing a blank. *K* can be connected to either *A* or *B* as the logic demands.

Now we have started our loop-type networks, not surprisingly, there is another problem: how do they close down? The problem here is in the definition of close-down of a process - a process closes down on the next deactivation after all of its upstream processes have closed down. In the above diagram, since *A* is upstream of *B*, but *B* is also upstream of *A*, we get a "catch-22" situation: *A* cannot close down because *B* cannot close down until *A* closes down, and so on. The solution is to provide a special service which makes a process look to its neighbours as if it has closed down. One of the processes involved must then decide to close down and will use this service to notify the other processes. In the batch situation, closedown of the network as a whole was typically initiated by readers closing down (because they had finished reading their files or had run into problems). In loop-type networks, one of the processes - usually one which is interacting with a user - has to decide that no more data is going to arrive, so it closes down.

The service which tells a process's neighbours that it has closed down is one we have mentioned casually before: "close port". FBP lets a component close an input port or close an output port. The function of this service is to close ports before they would normally be closed (this would normally happen automatically at process close-down time, but there are cases, like this one, where we just can't wait that long). A process closing an output port has the same effect on its downstream processes as if the process had terminated. A process closing an input port has the

same effect on its upstream processes; also if all of its input ports are now closed, it automatically terminates.

So, to close down the network, *A* or *B* in Figure 14.2 simply closes its input or output port - it doesn't matter which one. Suppose *B* closes its input port and ends execution: it will now terminate [because no more input data can now arrive]. *B* is in fact *A*'s upstream process, so *A* will also be able to close down, thus bringing down the whole network (the "kicker" process will have closed down long ago).

Now that we know how to make loop-type networks start and stop, why would we want to use them? This usually has to do with synchronization, which we will also be talking about in a later chapter. In a regular left-to-right network, the left side of the network will be processing the last IPs, while earlier IPs are being processed further to the right in the network. This asynchronism gives this technique a lot of its power, but there are situations where you have to coordinate some processing with a specific external event, or make sure that two functions cannot overlap in time. One such example is that of an interactive application supporting one user. Here *A* in the above example might be an interactive I/O component and *B* might be a component to handle the input and generate the appropriate output, e.g.:

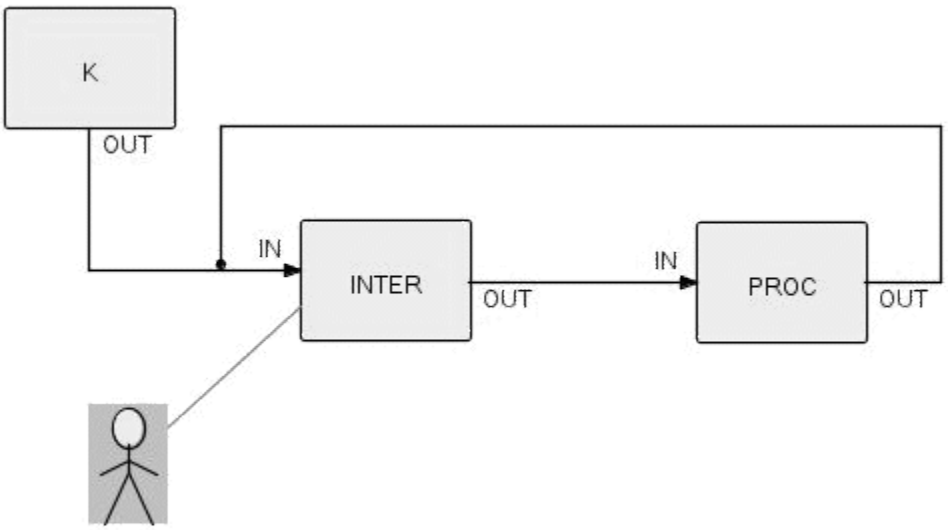



Figure 14.3

where *INTER* controls a screen. In this figure, *INTER* receives some data from *PROC*, displays it on the screen, waits for some action on the part of the end user, and then sends information back

to *PROC*. If the software infrastructure allows it, waiting for input need only suspend *INTER*, and other processes could be working on their input while *INTER* is suspended.

If, on the other hand, this were a left-to-right flow, and *PROC* were preceded by an input process and followed by an output process (without the "back flow"), input and output to and from the same screen would no longer be synchronized. You therefore have to synchronize at least one component to the pace of the user, so that he or she can act on the data presented on a screen before getting the next screen.

Since the IPs from which a screen is built must fit into the queues of the loop or/and the working storage of the processes, we have to make sure there is enough capacity in these queues. One way to make sure we don't have to worry is to use the tree structures described in an earlier chapter. A tree of IPs can be used to represent the screen data and can be sent around the loop as a unit. Alternatively, the screen data can be represented as one or more substreams, and then we just have to make sure the total queue capacity is set high enough.

As we shall see in Chapter 19, where we describe IBM's IMS on-line applications, you also get a loop structure there, but with a different purpose. IMS is a queue-driven on-line environment, and its Message-Processing Programs (MPPs) keep obtaining transactions from the IMS message queue until there are no more for that MPP, or until certain other conditions are met which cause the MPP to close down. Each time a transaction is obtained from the queue, IMS takes a syncpoint, so any positioning information from the previous transaction is lost, data bases are updated, etc. In FBP, we therefore build MPPs as loops where the next transaction is not read from the message queue until the previous one has been fully processed. The diagram would be the same as the previous one, except that *INTER* is replaced by a "transaction getter", as follows:

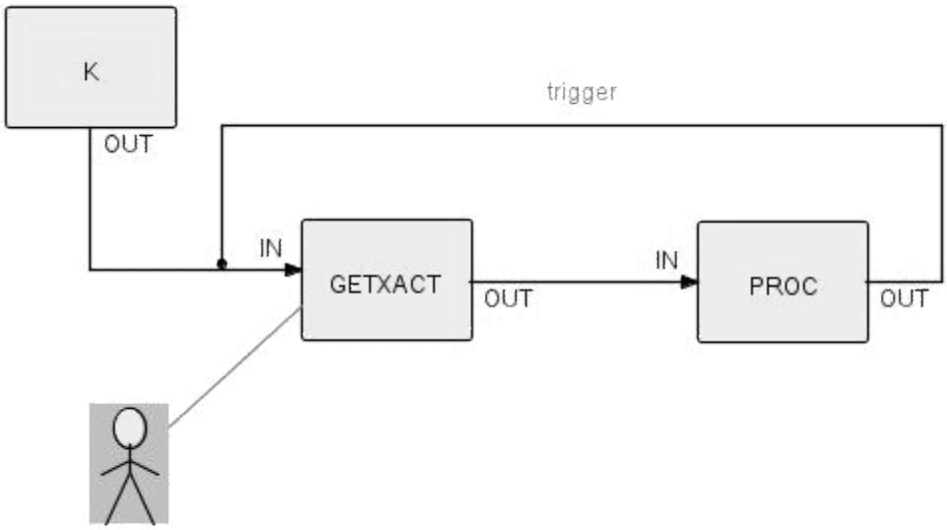



Figure 14.4

If you are familiar with IMS, you will also realize that each time around this loop the program will normally be dealing with different users, so, unlike the previous example, you cannot use the working storage of the processes to save information relating to one user.

Obviously in both of the above cases, *PROC* is shorthand for a group of processes which collectively process one screenful or transaction. This group of processes will accept data from the screen, and send out some kind of signal when they have finished with it. In both cases, it doesn't really matter whether the screen output data is routed back to *INTER* or *GETXACT*, or put out by a different process, as long as no more input is accepted until the output has been presented to the user.

[Some paragraphs on subnets and substream sensitivity - including Figure 14.5 - have been dropped as they are described in more detail in Chapters 7 and 20.]

Another use for loop networks is for "explosion" applications, of which the classical example is the Bill of Materials explosion, where components of some complex assembly will be "exploded" into subcomponents progressively until they reach ones which cannot be broken down any further. If you know that the largest possible explosion would not fill up storage, you could use a loop of two or more processes with very high capacity queues connecting them (it is dangerous to use a loop with only one process as you could land up getting deadlocked). Of course, the IPs for composite parts must be removed from the data stream when their subcomponents are added to it,

or when they are found not to be further reducible (like nuts and screws), so eventually the looping data stream will go empty.

This type of logic can also be useful when parsing other kinds of recursive structures, e.g. lists of lists or expressions in a language. A colleague, Charles Douglas, used it very effectively in a text processing application, where the user needed to be able to name lists of data bases and in turn use those names in other lists. He implemented this very similarly to a Bill of Materials explosion. His application went through all the lists, progressively exploding them until it got down to the actual data base level. Thus suppose, we have the following lists:

```
A:  B, C, D, E
B:  D, E, G
D:  E, F
```
Figure 14.6

Then, if you feed in A, the successive stages of explosion are as follows:

```
A
B, C, D, E
D, E, G, C, E, F, E
E, F, E, G, C, E, F, E
```
Figure 14.7

If the goal is to figure out how many of each atomic object you have, the totals are:

```
1 C, 4 E, 2 F, 1 G
```
Figure 14.8

If you simply want to list the different atomic objects involved, then you get:

```
C, E, F, G
```
Figure 14.9

Either way, the simplest technique is to follow the explosion with a Sort, and then either count items, or eliminate duplicates.

# Chap. XV: Implementation, Network Splitting and Client/Server

> "A complex system which works is invariably found to have evolved from a simple system which works", John Gall (1978)
>
> "Systems run better when designed to run downhill. Corollary: Systems aligned with human motivational vectors will sometimes work. Systems opposing such vectors work poorly or not at all", John Gall (1978)
>
> "Loose systems last longer and work better", John Gall (1978)

A colleague of mine, Dave Olson, has been studying the application development process in particular, and large organizations in general, and has realized that the budding science of chaos has a lot to tell us about what is going on in this kind of complex process and what we can do about it. In the study of chaos one frequently runs into feedback loops, and, in our business, just as one example, we have all experienced what happens when a project starts to run late. Fred Brooks described this kind of thing in his celebrated book, "The Mythical Man-Month" (1975). Chaotic systems also are characterized by areas of order and areas of apparent disorder. Dave's recent book describes these concepts and also describes how they can be applied to explain and handle many of the problems we run into in our business (Olson 1993). In a section on techniques for reducing disorder, he talks about DFDM and its relationship to the Jackson Inversion methodology (mentioned elsewhere in this book). He later gives more detail on DFDM, and describes how you would approach designing an application with it as follows:

To use DFDM, you first define the data requirements of the application, defining how the data flows and is transformed during the application process. Then you create transform modules for the places in the flow where data must be merged, split, transformed or reported. The full application definition consists of the data definitions and data flows, defined to DFDM, and the

transform modules that are invoked by the platform.

In a personal communication to me, he expands this into the following set of recommendations:

- View the application in terms of the information that needs to move from place to place.
- Create transform modules for the places where data must be merged, split, transformed, or reported.
- Separate data definitions from transform code so that transforms can be reusable code. Transforms thus care more about what has to be done, not so much about how individual pieces of information are represented.
- Provide certain building block transforms that will be needed by most applications, including file storage, file retrieval, file copy, printing, user interaction, etc.
- Build a systems framework so that an application can be defined in terms of its data flows and transforms; the framework handles scheduling, concurrency, and data transfers between transforms.

Doesn't sound much like conventional programming, does it? However, viewing an application in terms of data and the transforms which apply to it underlies many design methodologies. The problem has always been how to get from such a design to a working program. This is the "gap" I have referred to above. With FBP, you can move down from the highest level design all the way to running code, without having to change your viewpoint.

The first stage of designing an application is to lay out our processes and the flows between them. This is pretty standard Structured Analysis, and has been written about extensively by Wayne Stevens and others. Of course the scope of your application is sometimes not obvious - we can draw our network of processes as large or as small as we want. A system is whatever you draw a dotted line around. By this I mean that the boundary of any system is determined by an arbitrary decision that, for practical purposes, part of the world is going to be considered and part is going to be ignored. In the astronomy of our Solar system, we can treat influences from outside the Solar system as effectively negligible, and this works fine most of the time, even though theoretically every object in the universe affects every other. You may think that your skin separates a well-defined "you" from the rest of the universe, but biochemistry teaches us that all sorts of molecules are constantly passing in and out through this apparent barrier. Indeed all the molecules in our bodies are totally replaced over a period of a few years. This is rather a Zen idea - taken to the extreme it says that objects don't have an objective existence (pun intended), but are just the way we "chunk" the universe. And we should also be aware of the way the words we use affect the way we view the universe. As a linguist, I find a number of B.L. Whorf's examples quite striking: in one example (Whorf 1956), he gives a word in one of the Amerindian languages that describes people travelling in a canoe, in which there is no identifiable root meaning "canoe". The canoe is so much an assumed part of their experience that it doesn't have to be

named explicitly.

Thus, to design systems, we have to delineate the system we are going to design. This first decision is critical, and may not be all that obvious. So consider the context of the system you are designing: have well-defined (and not too many) interfaces with the systems outside your chosen system.

You can now lay out the processes and flows of your system. Explode the processes progressively until you get down to the level where you start to make use of preexisting components. In FBP, you will find these starting to affect your design as you get closer to them. This is one of the points where FBP design differs from conventional programming, but it should not really surprise us as we do not design bridges or houses purely top-down or purely bottom-up. As I said before, how can you "decompose" the design for a house so that it results in the invention of dry-wall? But conversely, you have to know where you're going in order to be able to determine whether you're getting closer to it. Ends and means must converge. Design is a self-conscious process of using existing materials and ideas or inventing new ones, to achieve a desired goal.

On the way to your goal, like any hiker, you have to be willing and able to change your plans as you go. The nice thing about programming is that you can move between a "real" program and its simulation very easily, so you can try something and modify it until you're happy with it. In fact, with FBP we can realize an old programming dream of being able to grow an application from its simulation. You can replace processes by components which just use up 'x' seconds of time, and vice versa.

In programming I believe there should be no strong distinction between a prototype and a real program until the real program actually goes into production. If something in the environment or in your understanding changes, so that an old assumption is no longer true, you have to be able to change your design. While I am very aware that the cost of change goes up the further back you go in the development process, you have to know when to cut your losses, and go back and do some redesign. If it helps, consider how much harder it is for, say, a surgeon to do this!

If you combine FBP with iterative development, I believe that you have a very powerful and responsive application development technology. You can try different approaches, it's cheap to make changes, and when it is time to dot the i's and cross the t's, it is essentially a linear process. By way of comparison, I believe that the chief failing of the "waterfall" methodology was that it was so awkward to go back that development teams would press forward phase after phase, getting deeper and deeper into the swamp as they went, or they would redesign, and pretend they were doing development (or even testing!). Here is Dave Olson again: "Programmers know that highly detailed linear development processes don't match the way real programming is done, but plans and schedules are laid out using the idea, anyway." He goes on to stress that, while there are some projects for which the "waterfall" process will work fine, you cannot and should not

shoehorn every project into it.

Apart from the iterative explosion process for networks we have just described (vertical), in FBP I have found there is another process which is orthogonal to it. This is the process of cutting up networks horizontally, to decide how and where they are going to run. In this process, we assign parts of our network to different "engines" or environments. Since FBP systems are designed in terms of communicating processes, these processes can run on all sorts of different machines or platforms - in fact anywhere that has a communication facility.

When you think about the ways a flow can be chopped up, remember you can use anything that communicates by means of data, which in data processing means practically any piece of hardware or software. I call this process "cleaving". Networks can be cleft (or cloven) in any of the following ways (in no particular order and not at all mutually exclusive):

- multiple computers (hosts, mid-range, PCs or mixture)
- multiple processors within one computer
- multiple tasks
- different geographical sites
- multiple job steps within a job
- secondary transactions
- client/server
- multiple virtual machines
- etc.

All of these have their design considerations, but that's just like the real world, where there are many different processes and they all communicate in different ways. Today I can call up a friend by phone, mail him or her a hand-written letter, fax it, communicate by E-mail, send a telegram, or stick a note in a bottle, throw it into the Atlantic and hope it gets delivered eventually! Each one of these has areas where it is stronger or weaker.

Let us take as our starting point one of the most common network cleaving techniques: multiple job steps in a job. Suppose you have decided that your network is going to run as a batch job under MVS. Here is a picture of a simple network:

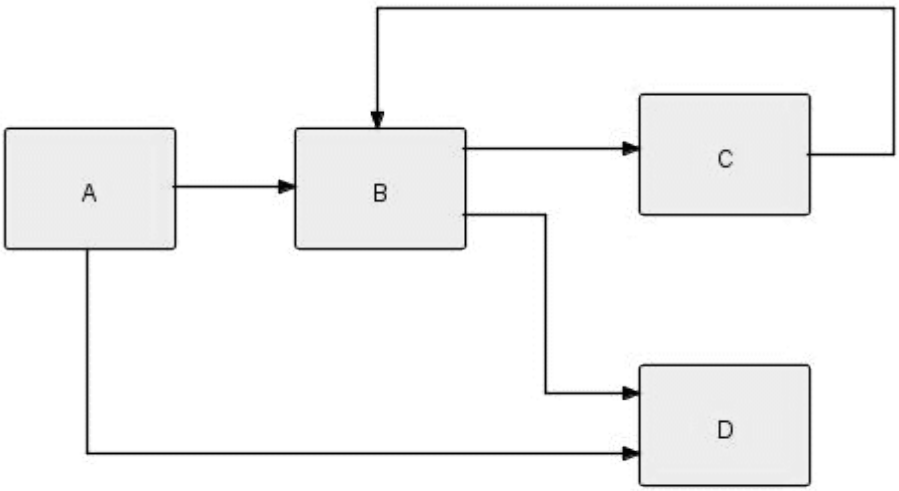



Figure 15.1

Let us suppose that, for whatever reason, we want to run this as multiple job steps of a single batch job. Now we know that job steps can only run sequentially. While they can be skipped, they cannot be repeated. This means that B and C must be in the same job step. However A and D can be split off into different job steps if desired. Suppose we decide to make both A and D separate job steps - this will result in 3 job steps, in total.

We also know how job steps communicate - by data files (yet another kind of data flow). In FBP we are free to replace any connection with a sequential file, so we can change the connections leading out of A and those leading into D to files. Once we separate two processes into different job steps, of course all connections between them must be replaced by files. Each one of these will in turn require a Writer and Reader. Using dotted lines to show step boundaries and [X]'s to indicate files, our diagram now looks like this:

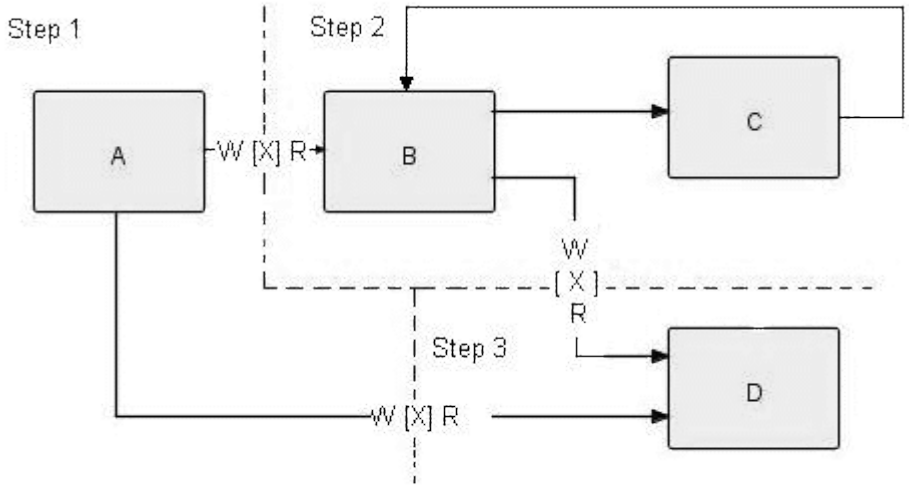



Figure 15.2

I have shown the Writers and Readers around the files as W's and R's to keep the similarity between the two previous figures. These are proper processes even though they are not shown as boxes. Steps 1 and 3 thus have 3 processes each; step 2 has 4.

If we show this picture at the job step level, we see that we could have designed it this way earlier in the process, and then exploded it to get the previous figure.

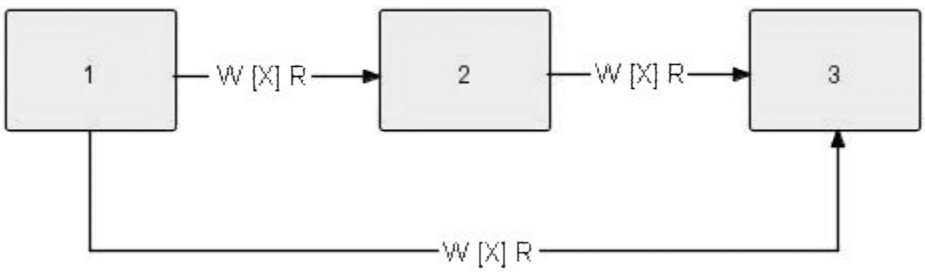



Figure 15.3

However, it is much better to design a complete application and then cleave it later, as other

considerations may affect our decisions on how the network should be split up. (For instance, the problem I mentioned in an earlier chapter of not being able to overlap sorts forced many of them into separate job steps!) In fact, since it is so easy to move processes from one job step to another, there is no particular point in splitting up our network early - you can just as well leave it to the last minute. In conventional programming, moving logic from one job step to another is very difficult, due to the drastic change of viewpoint when you move from data flow design to control flow implementation, so the decision about what are going to be the job steps has to be made very early, and tends to affect the whole implementation process from that point on.

I'd like to make another point about this rather simple example: in Figure 15.2, you notice that communication between process A and process B is managed by two processes and a file. In FBP, this is a common pattern whenever any two processes communicate other than via FBP connections. Thus a communication line might be "managed" by a VTAM Send process at one end, and a VTAM Receive process at the other. I will give a few examples later in this chapter of other kinds of communication between networks, and you will see this pattern in all of them.

A more subtle point is that the transition from Figure 15.1 to Figure 15.2 was made entirely by manipulating the network, without having to add any complexity to the underlying software. This point is related to the principle of open architectures I talked about in an earlier chapter: our experience is that it is much better to implement subsystems like file handling and communications as visible processes under the developer's control than to bury them in the software infrastructure. Since the latter is common practice among software developers, all I can say is that this preference is based on solid experience. But consider the advantages: Write-file-Read becomes just one mechanism supporting a particular way of cleaving networks - we could have used many others, and in fact could change from one to another during project development.

To try to show what happens if we go the other way, consider what would have happened if we had decided to have two kinds of connections: an intra-step connection and an inter-step connection. Let's take Figure 15.1 and assign each connection to one of these two categories depending on whether it crosses a step boundary or not (we'll assume you do this at the same time you divide the network up into job steps).

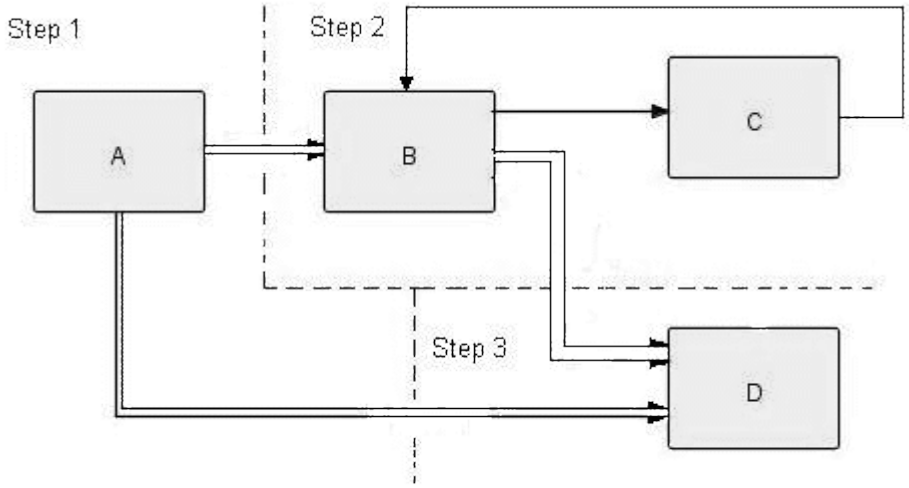



Figure 15.4

Now, I claim that we have made both the implementation and the mental model more complex, without providing any more function to the developer. Not only will the added complexity make the system harder to visualize and therefore harder to trouble-shoot, but we may have made some useful operating system features inaccessible or only accessible through still more complexity in the interface. And if a programmer wants to replace the inter-step connections with, say, a DB2 data base, s/he is going to have to add extra processes anyway, and change the inter-step connections back to intra-step connections....

Another way of presenting this discussion is that it is very important, when designing software, to decide on its scope. The scope of a given software system obviously depends on the particular environment it is running in, but it is sometimes not obvious which of the various constructs of that environment should be the basic "unit" of your software. In the case of DFDM, it will probably be obvious from all the foregoing descriptions and examples that a DFDM network corresponds to one of the following:

- an MVS job step,
- an IMS transaction,
- BMP,
- batch job,
- a CMS "program"

However, in the early stages of DFDM it was not at all obvious that this was the right decision. Even the CMS statement above is somewhat vague because the word "program" is one of the most confusing in the whole science of "programming"! Consider Figure 15.4 above - if we had implemented DFDM that way, the scope of a DFDM application would have been an MVS job, not a job step. There is no counterpart of this in IMS (except perhaps a "conversation"), or in CMS, so it would have been harder for programmers to move from one of these environments to another. I have always considered our choice of the scope of a network as being one of the keys to the usability of DFDM. In THREADS, a network is implemented as an .EXE file, which I think is also usually referred to as a "program".

So a system should have components of well-defined scope, with visible, controllable interfaces between them - in fact very much like FBP itself! Such systems (and, I believe, only such systems) can then be combined easily in open architectures.

Let's take another look at Figure 15.2. We could draw a box around each of the patterns shown as

```
----W [X] R----
```

to make it a process in its own right. This process can run perfectly well within a job step, and in fact we have met this pattern before in Chapter 13 (Scheduling Rules), where we drew it like this:

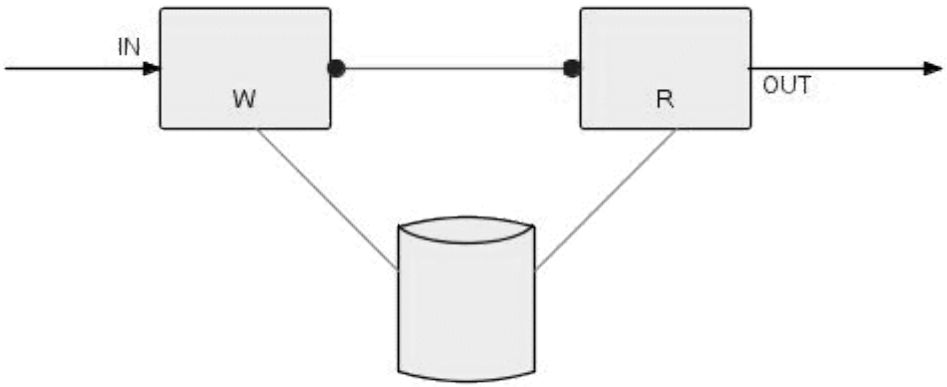



Figure 15.5

As you will recollect, we used automatic ports to ensure that the reader did not start until the writer had finished writing the file. Taken as a whole, this process is rather like a sort without the sorting function! It writes a whole data stream out to disk, then reads it back and sends it on. We can also state this a bit differently: it keeps accepting IPs, storing them in arrival sequence, until

it detects end of data; it then retrieves them and sends them on. This suggests the other name for this structure - we shall run into it again in the chapter on Deadlocks, where it is called, slightly fancifully, an "infinite queue". You will find that it is one of the most useful techniques for preventing deadlocks.

Why did we call it an "infinite queue"? a) In the old days we used to call "connections" "queues". b) Using a file for this type of structure gives us an effectively infinite capacity because, while a connection has whatever capacity the developer specified (or the default if none was specified), the pattern shown in Figure 15.6 can keep on soaking up IPs until it hits the maximum number of records allowed by the operating system or storage device. Actually, we could increase its capacity still further by using smart code inside a special-purpose component.

Given that processes are connected by queues, how can a particular type of process also be called a "queue"? We suggested at the beginning of the chapter that processes can be as large or small as you like. For pragmatic purposes, we have chosen to pick a certain "grain" of process as the one we will implement, and connect them with a different mechanism, the connection. There are higher-level processes, which we call subnets, and the highest subnets are job steps. Above this are still higher processes called jobs, applications, and so on. Different levels of this hierarchy will tend to be implemented by different mechanisms, but, if we need to, we can always move processes up or down the hierarchy. Treating a connection as a special type of process is like using a magnifying glass to zoom in on a part of our network. Any time we want a connection to behave differently from normal connections, we simply replace it by a process (plus the connections on either end). We could also use this technique if we wanted to have another type of connection between processes, say, a connection supporting "priority mail". However this will require additional port names - a number of writers have described processes implementing FBP's connections (also known as "bounded buffers"), but we have found it more convenient to put them in the infrastructure.

As I read the ever-growing body of literature on parallel processes, I have been struck by the variation in granularity across the various implementations, from fine-grained to very coarse-grained. And in fact you will have noticed that FBP processes vary from very simple to quite complex. However, they do seem to agree on a certain "grain size". In the chapter on performance, we will talk about how granularity affects performance, and how this fact can be used to obtain trade-offs between maintainability and performance.

In addition, we do not claim that the level of granularity described in this book is the best or the final one for all purposes - only that it is a level which we have found to provide a good balance between performance and expressive power. It is productive of useful components and developers seem to be able to become comfortable with it! Here is a network which implements what used to be called a "half adder", using IPs to represent bits and FBP processes to model Boolean operations - while this worked fine as a simulation, I do not suggest that we could build a real machine on it (although we might be able to build a slow simulation of a machine)!

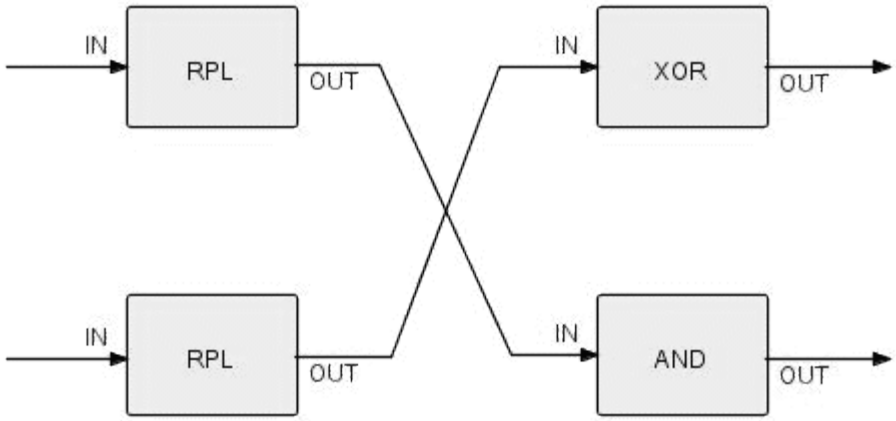



Figure 15.6

where RPL means Replicate, AND is the Boolean AND operator, and XOR is the Boolean exclusive OR.

We alluded above to "managing" a communication line by means of matching VTAM processes. Here we could distribute a network of processes between multiple computers, which may or may not be in the same room, building, city or country. This is "distributed" processing, which is gaining more and more attention. Instead of each computer labouring away by itself in splendid isolation, people today are accustomed to having computers be able to talk to each other around the world, using a wide variety of different communication techniques. The Net (Cyberspace) of today was foreshadowed within the IBM Corporation by the internal network called VNET, which is an incredibly powerful medium for sharing information and ideas. I once impressed the heck out of a visitor by showing him how, from a terminal in Toronto, by using a one-line command, one could display the VNET file traffic between Australia and Japan!

While I am not an expert in distributed systems, it seems logical to me that if we have two computers, each with multiple processes running on it, communicating asynchronously, then the two computers should be able to exchange data by simply using two dedicated processes at each end of the link. Schematically:

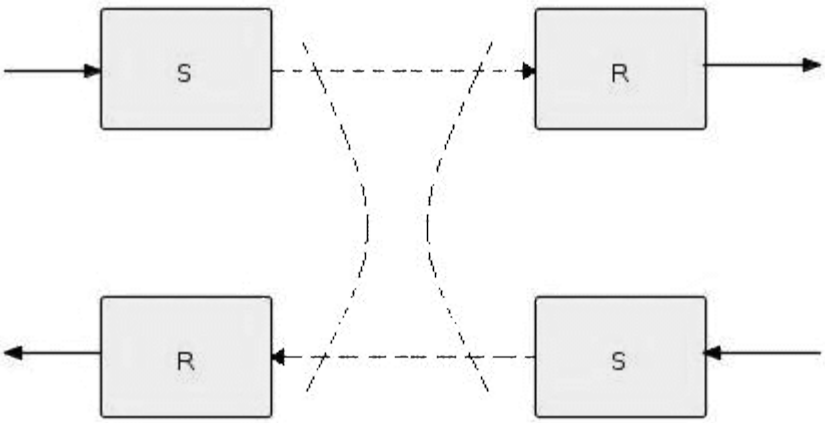



Figure 15.7

This diagram shows two systems, each with a Sender and a Receiver process. The Sender of one system sends to the Receiver of the other, and vice versa. At the far left and far right of the diagram there will be more processes, which are implementing the applications on each computer. You could view this whole structure as a single loop-type network which has been distributed over two systems [the fixed capacity of FBP connections can be simulated, we believe, by having R and S send and wait for acknowledgment packets, respectively, after every 'n' packets].

Clearly, in this example, we have coupled our two systems together, so that the speeds of the two systems are effectively matched. However, if the traffic between the systems is low enough, this may not be a bad design for some scenarios. For example, suppose a bank decides to spread its processing over two or more machines in different regions, linked by communication lines. Banking transactions are usually characterized by high "locality". The largest proportion of banking transactions will be on the customer's branch, a smaller number will be between branches on the same regional computer (assuming the bank splits up its branches that way), and only what's left over will actually need to travel between different regional machines, and most of the time this traffic should be balanced in the two directions. So we can visualize each machine as a "cloud" of processes, and every so often a transaction gets sent across to the other machine. If response time is adequate and/or the traffic is low enough, the requesting process can actually wait until a response comes back. If this is not adequate, the requesting process must have some way of suspending the logic which needs the data, and then resuming it later. This is exactly the problem of suspending/resuming FBP processes, but at a higher level. Also, in the case of links,

there is the possibility that the link has gone down (when there is a physical medium like wire, it can be cut or struck by lightning), so some thought has to go into what kind of degraded performance the application should provide if one of the systems cannot communicate with another.

One topic which comes up a lot with distributed systems is the vexed question of how to synchronize events across multiple systems. If all of the processes in an IMS application are running in the same IMS region, then issuing one Rollback command will undo all data base changes since the last checkpoint, so it is easy enough to undo some data base updates if it turns out that the larger action they are part of has failed. Now, however, imagine that we want to transfer funds between two different hardware nodes - you have to withdraw funds from an account on one node and deposit them into an account on the other one. Either of these operations can go wrong, so you need some hand-shaking to either delay both until you know it's safe, or be prepared to undo an earlier operation if a later one goes awry. The latter approach seems more natural, even if less general: you do the withdrawal first; if that works, you send an IP representing a sum of money to the other node, specifying both source and destination accounts. If the receiving end for some reason cannot accept the transfer, it gets sent back to be redeposited. You might call this the "optimistic" stategy, while only committing when you know both ends are OK is the "pessimistic" strategy. The point is that qualitative changes occur when you split your network between systems, but this is an area of expertise in its own right, and the argument for distributing systems is so compelling that the new problems that it raises will definitely be solved! After all, systems of communicating humans have been working this way since time immemorial! IBM has recently announced its Message and Queuing Interface, so it is betting that asynchronous communication between systems is the way to go, and I am sure there will be solutions to these problems appearing over the next few years.

There is a lot of interest these days in what are called client-server relationships. You may in fact have realized by now that, in FBP, every process is a server to its upstream processes. In the banking example above, where there are more than two systems involved, every system is potentially a server to all the others. So the client-server paradigm is a very general one, from the level of FBP processes as high up as you care to go. However, it is important to stress also that FBP processes are peers, and peer-to-peer, which is more general than client-server, is a very natural match with the FBP approach. The FBP model is cooperative rather than hierarchical.

One major addition we have to make to the above schema is to allow for the possibility of asynchronous flows from multiple sources. If the Toronto machine can have data flowing in from Montreal and/or Edmonton, it has to be able to accept data on a first-come, first-served basis (subject, perhaps, to a priority scheme if there is a need to handle express messages). We have already seen in FBP a mechanism which provides this - namely, multiple output ports feeding one input port. If you have this situation, you must have some way of routing the response back to the right source, so the data must be tagged in some way. In FBP, one way is to arrange for the

incoming data to contain a "source index", which we could use as an output port element number. Showing just two clients, schematically:

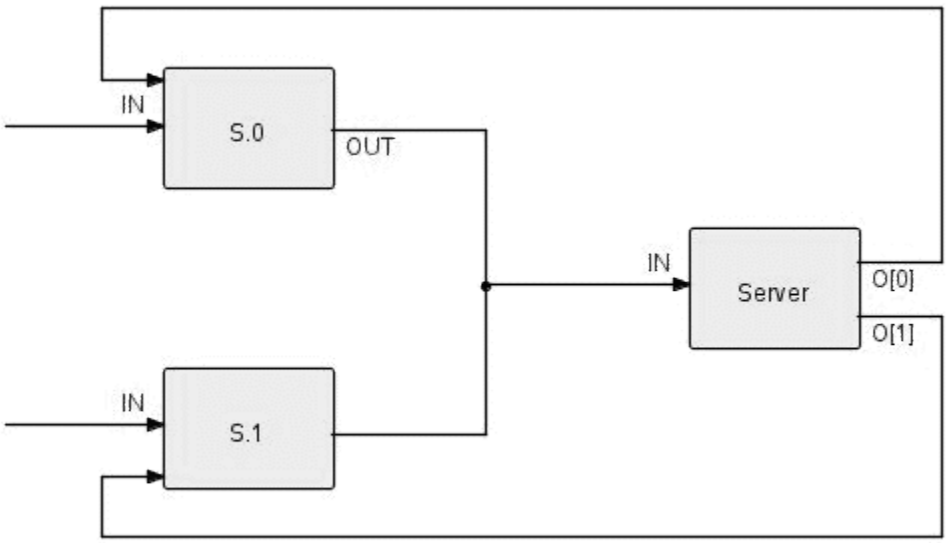



Figure 15.8

In this configuration "S.0" sends an IP to the server, and then waits for the response to arrive back, after which it can send another request. Meanwhile, "S.1" is going through the same cycle. You can see from the diagram that IPs from "S.0" and "S.1" will arrive at the server in "first come, first served" sequence. Of course, the server can only handle one request at a time, so it has to be able to process requests as fast as possible, relative to the arrival rate. Otherwise, a queue builds up which will in turn slow down all the processes requesting service. As in the above discussion of "cleaving" networks, any line in the above diagram can be replaced by other forms of communication, including the above-mentioned "Send-line-Receive" pattern, resulting in various kinds of distributed network. The "n to one" connection in the middle of Figure 15.9 would be implemented very naturally as multiple Senders sending to a single Receiver process. In FBP, the reverse situation, a "one to n" connection, is not implemented directly in network notation, but can be implemented simply by a Replicate process. In a distributed network, this would correspond naturally to a "broadcast" communication function, where one process sends the same message to multiple receivers. This is the kind of arrangement used by many taxi dispatching systems, where the dispatcher sends in broadcast mode, but individual taxis reply on their own wavelength.

We mentioned that the server in Figure 15.9 can only handle one request at a time. This would be the simplest way of implementing its logic. Another wrinkle might be to add an "express" port, like the express lane at some banks. However, unless the express port has its own server, this scheme can still be degraded by a low-priority request which is tying up the server. So it might be better to have two processes running concurrently, one handling express requests, and the other low--priority requests. Of course, this adds logical complexity, especially if the two servers are competing for resources. One possible solution is to implement an enqueue/dequeue mechanism to resolve such conflicts - this actually fits quite well with FBP, and was implemented experimentally in DFDM. The above-mentioned article (Suzuki et al. 1985) also describes a similar feature its authors developed as an alternative communication mechanism between processes.

Generalizing still further, we may multiplex the server by having multiple instances of the same component. For instance, suppose the server is doing I/O which may be overlapped with processing - you could have many instances of the same component serving the client queue. This is like many bank branches today, where several tellers are servicing a single queue of customers. To allow them to overlap I/O, you have to be able to suspend only a single process on an event, without hanging the whole application. DFDM supported this for the MVS "basic" access methods (BSAM, BPAM, BDAM), VSAM and EXCP. THREADS does not support this. Of course, this is subject to the same resource contention we talked about above.

If you are going to multiplex your server processes, you will probably need a "load balancing" process to ensure that queues don't build up more in front of some servers than others. The resulting diagram might therefore look like this:

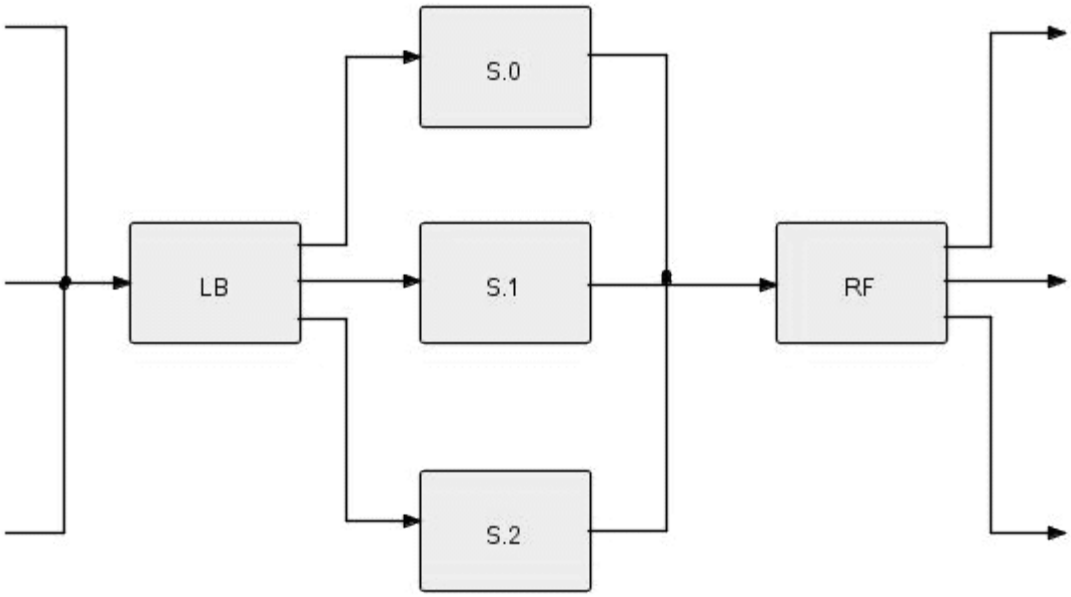



Figure 15.9

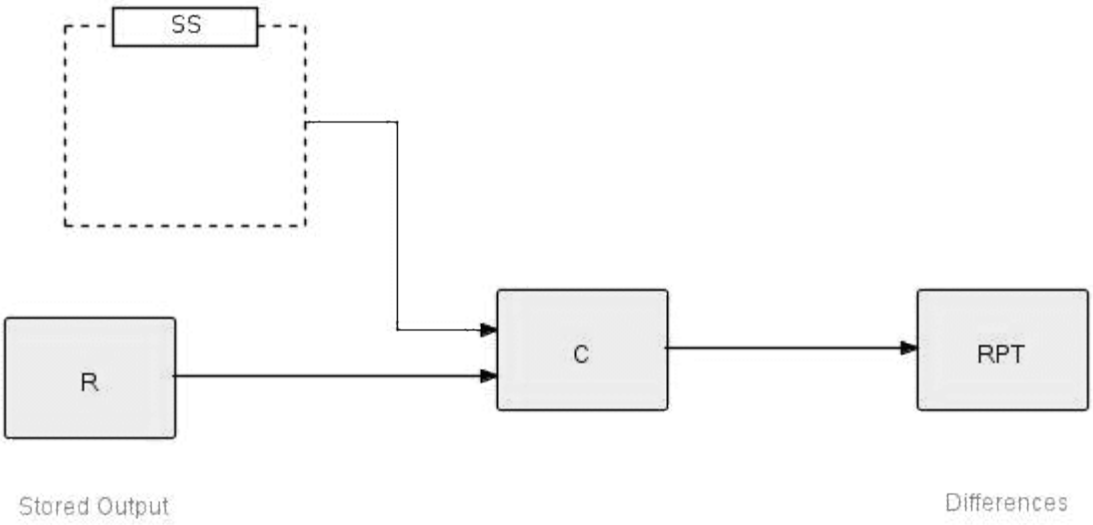
In the above diagram, requests are assumed to be coming in at the left side, tagged with an indication as to their source. They merge into LB which performs load balancing, sending the incoming requests to whichever server has the shortest input queue (number of IPs in its input connection). From the servers, the IPs then travel to RF, which is a redistribution facility. From there they eventually travel back to their original requesting processes. You will find a description in Chapter 22 of how this technique was used very effectively to reduce the run-time for a disk scanning program from 2 hours down to 20 minutes.

One topic which has been getting a lot of attention recently is that of portability of code. This is certainly one of the appeals of HLLs, but my personal feeling is that their disadvantages outweigh their advantages. But clearly you cannot port Assembler language from one type of hardware to another using a different instruction set. I believe the solution is to port at the function level. A Collator can be defined to do the same job on two or more input streams, independent of what language it was coded in. So we can move Collators from one machine to another and expect that our function works identically. Not only does this approach take advantage of the strength of the black box concept, but it removes the necessity for code to be white boxes written in a portable language (which usually results in lowest-common-denominator languages, or frustrating and time-consuming negotiation about standards).

We have now seen how to decompose designs vertically, cleave them horizontally, and some other concepts such as client-server relationships. But in what order should you develop your application? I don't think there is a fixed sequence, but I have found two sequences productive: along the lines of bottom-up and top-down, I call them "output-backwards" and "centre-out". "Output-backwards" means that you start with the outputs intended for human consumption, and work backwards deciding how this data is going to be generated. If it's an interactive application, output is the screen, and you will eventually work back to the screen, which closes the loop. "Centre-out" means that you start with the core processes, usually the ones which do business logic, together with any required Collators and Splitters, and get them working using the simplest Reader and Display components. Fancy output formatting, input editing, etc., can then be added incrementally working out from the centre. This approach is really a kind of prototyping, as you can develop the main logic and check it out first, before investing effort in making the output pretty, making the application more robust, etc. At selected points along the way, you can demonstrate the system to potential customers, prospective users, your manager, or whomever.

Last but not least, during this whole process of development you will want to test your developing product. You can test any component by just feeding it what it needs for input and watching what comes out as output. You generate its input using Readers and its output using some kind of Display component. In the early stages, the simpler these scaffolding components are, the better. The trick, as in prototyping, is to only introduce one unknown at a time. The more you can use precoded components, and the simpler they are, the less unknowns you will be dealing with. Any time you want to see the data that is travelling across a connection, just insert a Display process, like a probe in an electrical circuit. So testing is very simple.

As I have tried to emphasize in what went before, I feel that, while FBP is great for developing applications, it is perhaps even more effective in improving maintainability. After all, all attempts to make programs more maintainable after the fact have failed, so maintainability has to be designed in from the start. As some of the anecdotes above attest, FBP seems to have it designed into its very fabric. Given that your program will change constantly over the years, you will need a simple way to verify that the logic which you didn't change still works. Testing that a system still behaves the way it used to is often called "regression testing". Wayne Stevens came up with an FBP-based technique for doing this. You take the original output of the system to be tested and store it in a file; you then imbed your system in a network like this:




```
where
        SS represents the Subject System
        R is a Reader for the previously stored output
        C compares the stored output against current output
        RPT generates a report on the differences
```

Figure 15.10

This network is clear while also being generic. For instance, you might need to insert filters on C's two input streams to blank out things which will vary between the two runs, e.g. dates and times. You might want to display the differences in different orders, so you can insert a suitable Sort upstream of RPT.

A related (FBP-specific) problem comes up if C's input streams are not in a predictable sequence, say, because they are the result of a first-come-first-served merge, or if overlapped I/O has been occurring, so some IPs may have overtaken others. The simplest solution is probably to sort them before C sees them, but a better one, if it is feasible, is to mark the IPs at a point where they still have a fixed sequence, so that they can be split apart again before doing the compare. Of course, any processing you do to the differences (after the compare) will be cheaper than processing you do before the compare process (assuming that most of the data has compared equal).

I have given you a very high-speed description of some methodology and implementation techniques which are characteristic of FBP. You will doubtless come up with your own

procedures, guidelines and "rules of thumb", and I hope that eventually there will be a community of users of FBP all exchanging ideas and experience (and experiences).... and more books!

# Chap. XVI: Deadlocks - Their Causes and Prevention

Deadlocks are one of the few examples in FBP of problems which cannot involve only one process, but arise from the interactions between processes. They occur when two or more processes interfere with each other in such a way that the network as a whole eventually cannot proceed. I know some programmers feel that they have traded the known difficulties of conventional programming for a new and exotic problem which seems very daunting, and they wonder if we have not just moved the complexity around! Although deadlocks may seem frightening at first, I can assure you that you will gain experience at recognizing network shapes which are deadlock-prone, and will learn reliable ways to prevent deadlocks from occurring. I know of *no* case where a properly designed network resulted in a deadlock during production.

First, however, I think we should look a little more deeply at the question I mentioned above: if FBP is as good as I claim, why does it give rise to a new and exotic class of problem which the programmer would not encounter in normal programming? Well, actually, it's not new - it's just that a conventional program has only one process, so you don't have to worry about deadlocks. In FBP we have multiple processes, and multiple processes have always given rise to deadlocks of various kinds. If I have convinced you in the foregoing pages that the FBP approach is worthwhile, then you will be relieved to know that there is quite a large literature on multi-process deadlocks. Anyone who has designed on-line or distributed systems has had to struggle with this concept. For instance, suppose two people try to access the same account at a bank through different Automatic Teller Machines (ATMs). In itself this doesn't cause a problem, one just has to wait until the other one is finished. Now suppose that they both decide to transfer money between the same two accounts, but in different directions. Normally, this would be programmed by having both transactions get both accounts in update mode (Get Hold in IMS DL/I terms). Now consider the following sequence of events (call the accounts X and Y):

- trans A - get X with hold
- trans B - get Y with hold

- (1) trans B - try to get X with hold
- (2) trans A - try to get Y with hold

At point (1), transaction B gets suspended because it wants X, which is held by A. At point (2), transaction A gets suspended because it wants Y, which is held by B. Now neither one can release the resource the other one wants. If A had not needed Y, it would have eventually finished what it was doing, and released X, so B could have proceeded. However, now that it has tried to grab Y, the result is an unbreakable deadlock. This is often referred to as a "deadly embrace", and in fact it has similarities to one kind of FBP deadlock. Usually on-line systems have some kind of timeout which will cancel one or both of the transactions involved.

Normally, the chance of this kind of thing happening is infinitesimal, so a lot of systems simply don't worry about it. It can be prevented completely by always getting accounts in a fixed sequence, but this may not possible in all situations, so it is probably true to say that some proportion (even if a tiny one) of transactions will deadlock in an on-line system, so you have to be able to take some kind of remedial action.

In FBP, we can also get deadlocks between the processes of a network. Of course, in an FBP on-line system, each transaction would be a network, so you can get deadly embraces between separate networks requesting the same resources, but we will assume these are handled in the normal way for the underlying resource management software. However, because a network has multiple processes, we can get deadlocks within a network. In all these intra-network cases, we have discovered that it is *always* a design issue.

This type of deadlock can always be detected at design time, and there are tried and true techniques for preventing them, which we will be talking about in this chapter. Wayne Stevens also has a very complete analysis of deadlocks and how to avoid them in Appendix B of his last book (Stevens 1991).

The general term for deadlocks like the deadly embrace is a "resource deadlock". There is a classical example of this in the literature called the "dining philosophers problem", which has been addressed by many of the writers on multiple processes. Imagine there is a table in a room, on which are are 5 chopsticks, spaced equally around the table, and a bowl of rice in the centre. When a philosopher gets hungry, he enters the room, picks up the two chopsticks on each side of his place, eats until satisfied and then leaves, replacing the chopsticks on the table (after cleaning them off, I hope). If all the philosophers go in at the same time, and each happens to pick up the chopstick to his right (or left), you get a deadlock, as nobody can eat, therefore nobody can free up a chopstick, therefore nobody can eat! Notice that the dining philosophers exhibit the same loop topology which I have been warning you about in earlier chapters. The dining philosophers can also suffer from the reverse syndrome, "livelock", where, even though there is no deadlock, paradoxically one of the philosophers is in danger of starving to death because there is no guarantee that he will ever get the use of two adjacent chopsticks. Kuse et al. (1986) proved that,

although a network with fixed capacity connections (like the ones in FBP) can suffer from deadlock, it can never suffer from livelock.

We will use Figure 14.2 from the chapter on Loop-Type Networks to introduce an FBP version of a resource deadlock. I'll show it again here for ease of reference:

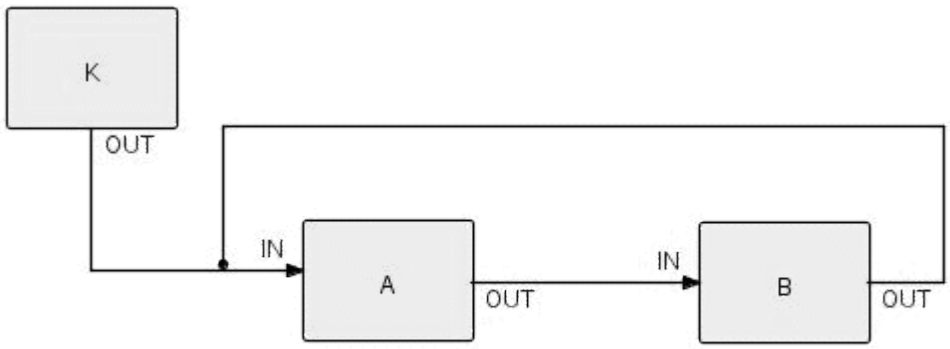



Figure 16.1

Suppose that A's logic and B's logic are both the same, and look like this:

```
/* Copy IPs from IN to OUT */
        receive from IN using a
        do while receive has not reached end of data
                send a to OUT
                receive from IN using a
        enddo
```

Figure16.2

In other words, they are both simple filters. However, suppose that under some circumstances B fails to send the IP it has just received to its OUT port, so its logic is effectively the following:

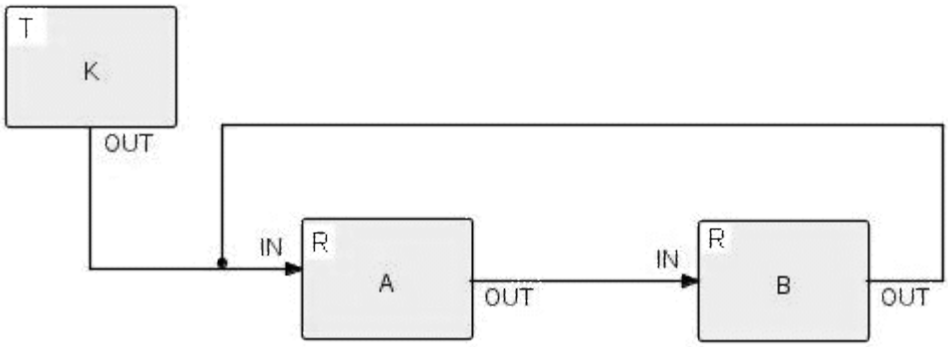
```
/* Copy c-type IPs from IN to OUT */
        receive from IN using a
        do while receive has not reached end of data
                if condition c is true
                        send a to OUT
                endif
                receive from IN using a
        enddo
```

Figure16.3

We'll begin by starting A. A sends an IP out of its OUT port. It next does a receive on IN, but there is nothing there, so it suspends. The connection between A and B now has something in it, so B starts up. Let's say condition "c" is false, so B receives the IP, but fails to send it out. B then goes on to its receive, and suspends. A is also suspended on a receive, but it is going to wait indefinitely as B hasn't sent the IP that A is waiting on. B is suspended indefinitely since A cannot send what B is waiting on. This is very like the deadly embrace: two processes, each suspended waiting for the other to do something. Since A and B are suspended on receive, there is nothing active in our network (K has terminated). So no process can proceed. The program as a whole has certainly not finished normally, but nothing is happening! This is in fact how the driver recognizes that a deadlock has occurred.

AMPS and DFDM add a wrinkle to this: it is possible to have one or more processes suspended waiting for an external event (such as I/O completion) to occur. In this case, the network as a whole goes into a "system wait", waiting on one or more external events. When one of these happens, the driver regains control, and gives control to the process in question. If it weren't for the fact that someone is waiting on an external event, this would look just like a deadlock. Instead, the event will eventually happen, allowing the process to continue, thus allowing all the other interlocked processes to start up as needed. This feature is one of the important performance advantages of FBP: unlike conventional programming, if a process needs to wait for an event to occur, it is usually not necessary with FBP to put the whole network into wait state (underlying software permitting).

Let's redraw the diagram with the states of each process shown in the top left corner of the process. You'll find this pattern of states is quite common, and is quite characteristic of a certain kind of deadlock. The drivers of all FBP implementations always list the process states as part of their diagnostic output in the event of a deadlock.




```
where T means terminated,
and   R means suspended on receive
```

Figure16.4

Clearly, this kind of deadlock will occur any time a process outputs fewer IPs than the downstream process requires. Since the only kind of resource "native" to FBP is the IP, the only problem can be non-arrival of IPs. So thorough testing will detect it. We could also have "infrastructure" or "hybrid" (FBP plus infrastructure) deadlocks, where, say, one process is waiting on an external event which never happens, and another on an IP that the former is to send out. There is nothing to prevent a DFDM process from issuing an MVS ENQ, thus allowing two processes to interlock each other by both issuing two ENQs for the same resource, but in different orders. Old-style deadlock considerations apply here, but, as I've said, this is a well-understood area of computing.

The other class of deadlock familiar to systems programmers is the "storage deadlock". In this kind of deadlock, the deadlock occurs because one process does not have enough storage to store information which is going to be needed later. This type of deadlock, unlike the resource deadlock, can usually be resolved by providing more storage (not always, because some variants of this continually demand more storage...).

Here is an example of a storage deadlock: suppose you are counting a stream of IPs, and you want to print out all the IPs, followed by the count. The components for doing this should be quite familiar to you by now: the IPs to be counted go into the COUNT process at the IN port and emerge via the OUT port, while the count IP is generated and sent out via the COUNT port at close-down time. So all we have to do is concatenate the count IP after the ones coming from OUT. This is certainly very straightforward - here's the network:

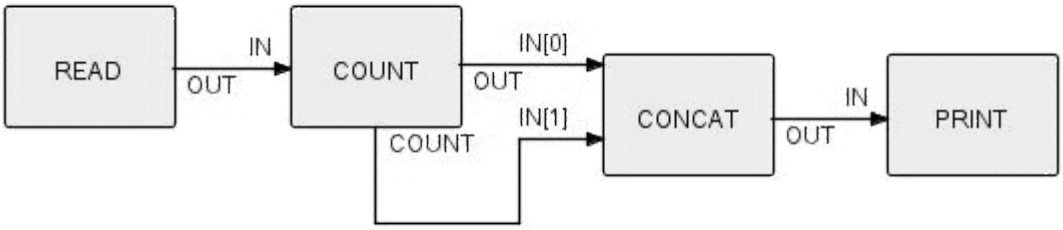



Figure16.5

We can code this up in THREADS, using the general purpose components available in THREADS. We will be describing the THREADS syntax more fully in Chapter 23 on "Notation", so we will not show the full network definition here, but the essential point is that we will connect the OUT port of Count to IN[0] of Concatenate, and the COUNT port of Count to IN[1] of Concatenate, as follows:

```
    Read(THFILERD) OUT ->
      IN Count(THCOUNT) OUT ->
        IN [0] Concatenate(THMERGE) OUT ->
          IN Print(THVIEW),
    Count COUNT -> IN[1] Concatenate,
   'data.fil' -> OPT Read;
```

Figure16.6

Running this network gives the following result, which is just what we would expect:

```
RUNNING NETWORK deadlk
 Process: Read (THFILERD)
 Connection: Read OUT[0] -> IN[0] Count
 Process: Count (THCOUNT)
 Connection: Count OUT[0] -> IN[0] Concatenate
 Process: Concatenate (THMERGE)
 Connection: Concatenate OUT[0] -> IN[0] Print
 Process: Print (THVIEW)
 Connection: Count COUNT[0] -> IN[1] Concatenate
 IIP: -> OPT[0] Read
    'data.fil'
Scan finished
Length: 3, Type: A, Data: 111
Length: 3, Type: A, Data: 123
Length: 3, Type: A, Data: 133
```

```
Length: 3, Type: A, Data: 134
Length: 3, Type: A, Data: 201
Length: 3, Type: A, Data: 211
Length: 3, Type: A, Data: 222
Length: 3, Type: A, Data: 234
Length: 3, Type: A, Data: 300
Length: 3, Type: A, Data: 450
Length: 3, Type: A, Data: 700
Length: 3, Type: A, Data: 999
Length: 10, Type: COUNT, Data: 12******70
Done
```

Figure16.7

In this example, the extra asterisks and 70 in the count IP are due to the fact that this particular Count component (THCOUNT) allocates a 10-character IP, but uses the C function "ltoa" to display the count, in this case resulting in 2 characters followed by a binary zero byte. The rest of the IP is unchanged, so it contains "garbage" (official programming term).

But now let's suppose that, for some perverse reason, you want to see the count IP ahead of the ones being counted. Since we're using a Concatenate function, you might think all we have to do is concatenate the count ahead of the IPs coming out of OUT. So we change the picture as follows:

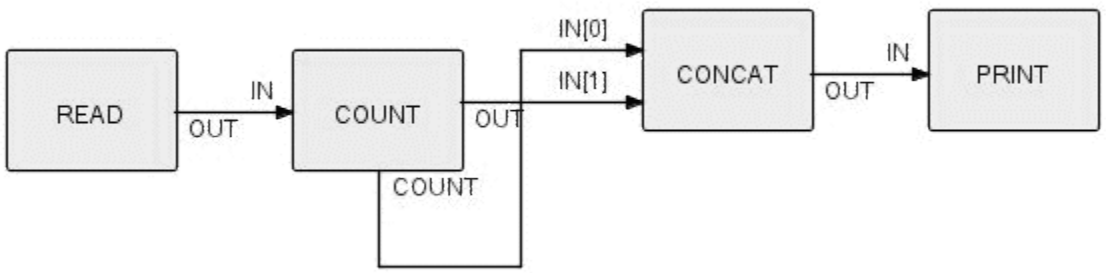



Figure16.8

Let's change the previous network to do this and try it out:

```
Read(THFILERD) OUT ->
    IN Count(THCOUNT) OUT ->
      IN [1] Concatenate(THMERGE) OUT ->
        IN Print(THVIEW),
  Count COUNT -> IN[0] Concatenate,
```

```
   'data.fil' -> OPT Read;
```

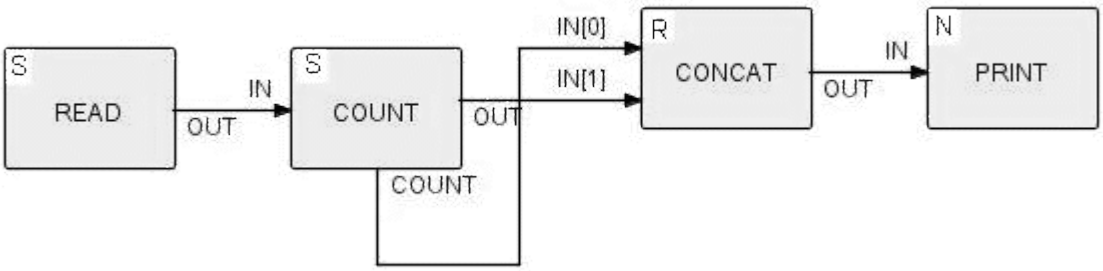
Figure16.9

To our surprise, the result of this is a little different:

```
RUNNING NETWORK deadlk
 Process: Read (THFILERD)
 Connection: Read OUT[0] -> IN[0] Count
 Process: Count (THCOUNT)
 Connection: Count OUT[0] -> IN[1] Concatenate
 Process: Concatenate (THMERGE)
 Connection: Concatenate OUT[0] -> IN[0] Print
 Process: Print (THVIEW)
 Connection: Count COUNT[0] -> IN[0] Concatenate
 IIP: -> OPT[0] Read
    'data.fil'
Scan finished
Deadlock detected
 Process Print Not Initiated
 Process Concatenate Suspended on Receive
 Process Count Suspended on Send
 Process Read Suspended on Send
```

Figure16.10

This is definitely not the result we wanted! Now, the message said "Deadlock", so, as before, let's try taking the process states and inserting them back into the diagram:




```
where S means suspended on send,
      R means suspended on receive,
and   N means not initiated
```

Figure16.11

You will find that in this kind of deadlock there is always a "bottleneck", or point where the upstream processes tend to be suspended on send, and the downstream processes are suspended on receive. As we saw before, some processes may have terminated - these can be excluded from this discussion. Processes which are inactive or not yet initiated can be treated as if they are suspended on receive.

If COUNT is suspended on send and CONCAT is suspended on receive, but they are adjacent processes, then clearly COUNT is sending data to a connection different from the one CONCAT is waiting on! Although there is data on the connection labelled OUT, CONCAT insists on waiting on a connection where there is no data, and, at this rate, there never will be! The problem is that the COUNT IP is not generated until all of the input IPs have been processed, and the connection labelled OUT has a limited capacity, so there is nowhere to store the IPs after they have been counted.

Although this kind of deadlock resembles a resource deadlock (because each process is waiting on the other to do something it can't do), it can be resolved by giving one or more connections more storage, so it is of the class of storage deadlocks. You will remember that we don't make all our queues "infinite" because then you can't guarantee that everything will get processed in a timely manner (or ever, if you want to allow 24-hour systems). But if you visualize the OUT connection bulging like a balloon as IPs are pumped into it, you can see that, eventually, COUNT is going to have processed all its input and will generate the count IP, and we can start to let the balloon deflate back to normal!

What's the best way of doing this? Unfortunately, the question depends on how many IPs are being counted. If you knew absolutely how many IPs there were, you could just make the capacity of the connection big enough. We can do this in THREADS by adding a capacity number in brackets after the arrow. Since "data.fil" has only 12 IPs, let's make the capacity 20:

```
    Read(THFILERD) OUT ->
      IN Count(THCOUNT) OUT -> (20)
        IN [1] Concatenate(THMERGE) OUT ->
          IN Print(THVIEW),
    Count COUNT -> IN[0] Concatenate,
   'data.fil' -> OPT Read;
```

Figure16.12

The result is as follows:

```
RUNNING NETWORK deadlk
 Process: Read (THFILERD)
 Connection: Read OUT[0] -> IN[0] Count
 Process: Count (THCOUNT)
 Connection: Count OUT[0] -> IN[1] Concatenate
 Process: Concatenate (THMERGE)
```

```
 Connection: Concatenate OUT[0] -> IN[0] Print
 Process: Print (THVIEW)
 Connection: Count COUNT[0] -> IN[0] Concatenate
 IIP: -> OPT[0] Read
    'data.fil'
Scan finished
Length: 10, Type: COUNT, Data: 12********
Length: 3, Type: A, Data: 111
Length: 3, Type: A, Data: 123
Length: 3, Type: A, Data: 133
Length: 3, Type: A, Data: 134
Length: 3, Type: A, Data: 201
Length: 3, Type: A, Data: 211
Length: 3, Type: A, Data: 222
Length: 3, Type: A, Data: 234
Length: 3, Type: A, Data: 300
Length: 3, Type: A, Data: 450
Length: 3, Type: A, Data: 700
Length: 3, Type: A, Data: 999
Done
```

Figure16.13

Worked like a charm!

Hold on, though - how do you know the file will never exceed 20 IPs? Remember the provinces of Canada - the process has already started which should eventually lead to the creation of a new one (or several). It's a fair bet that there will always be 7 days in a week, or 365 1/4 days in a year, but most other "constants" are subject to change, not to mention really variable numbers like the number of departments in a company.

Now suppose we follow the balloon analogy a bit further - what we could do is allow connections to bulge only if you would otherwise get a deadlock (i.e. no process can proceed), and let them bulge until we run out of storage. This initially seems attractive, but it makes error determination more complex. Remember that we allocate storage for IPs when they are created, not when we put them into a connection, so you will eventually run out of storage somewhere, and you won't know who is the culprit... And anyway, it's not a good idea to let the FBP Driver grab all of storage, because the storage some other subsystem needs to let it come down gracefully may not be available.

This whole area is tricky. If you know there will never be more than 50 provinces in Canada, and that you will never need more than 1000 bytes per province, you could hold them all in storage for a total of 50,000 bytes, which is pretty small peas these days. But what if each province is a tree, and you really don't know how many IPs are hung off each province, or how big they are?

So far, the best general solution we have come up with is to use a sequential file. You will

remember from Chapter 15 that we can replace any connection with a file. Let's change the network as follows, using the notation we used in Chapter 15:

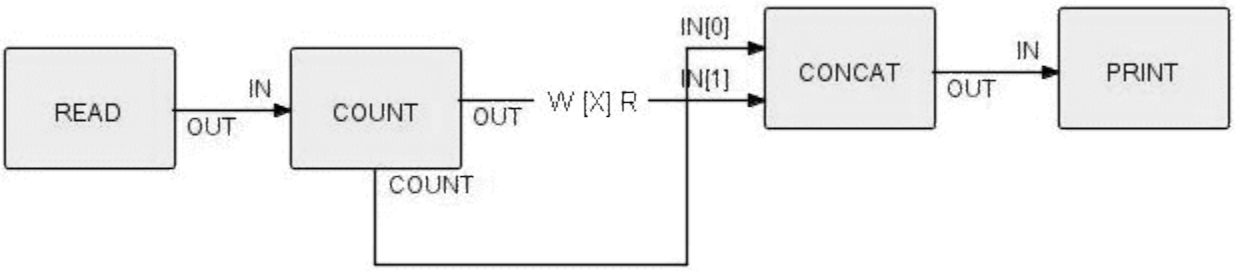



Figure16.14

In that chapter we used ---W [X] R--- to indicate a "sandwich" with the file being the meat, and a Writer and Reader being the bread on each side. In Chapter 15, we used automatic ports to ensure that the Reader does not start until the Writer is finished.

Now, if you step this through in your head, or try it out on the machine, you will find that the file soaks up IPs until it gets end of data. Meanwhile CONCAT is waiting for data to arrive at IN[0]. The "meat" file's Reader can start at some point after this, but will only run until its output connection is full. At this point, the only thing that can happen is for CONCAT to receive the count IP, then end of data, then switch to receiving from IN[1]. From this point on, the data IPs will just flow through IN[1] to CONCAT's output port.

We have seen that you can set capacities on all connections. What would happen if we gave all connections a standard capacity of 10? Then examples like the above [the situation before we added the sequential file "sandwich"] would work fine as long as there were only 13 Canadian provinces and territories. Add another province one day, and your nice production program goes boom! Since amounts of test data tend to be smaller than production amounts, we would never be sure that programs which worked fine in test would not blow up in production. We have therefore adopted the strategy during testing of making connection capacities as small as possible, to prevent this kind of thing from happening. Zero-capacity connections would be very nice, but they are qualitatively different from non-zero-capacity connections (there are some situations where they really don't behave in the same way), so we use 1-capacity connections. Once we go into production, however, we can safely increase connection capacities to reduce context-switching overhead. This strategy has been followed in both DFDM and THREADS, and has been very successful at helping to reduce the occurrence of deadlocks. A propos, my son,

Joseph Morrison, an experienced programmer, has suggested the following aphorism: "During testing, better to crash than to crawl; in production, better to crawl than crash!" The second half of this doesn't mean, of course, that you should put poorly performing systems into production - it just means that in production, when problems arise, it is better to try to keep going, even if in a degraded mode, but of course you should make sure someone knows what's going on! There is a story about one of the early computers (either SAGE or STRETCH) which did such a good job of correcting storage errors, that for some time nobody noticed that it was running slower and slower!

In Chapter 15, we talked about how the Writer/Reader sandwich is sometimes referred to as an "infinite queue". This is a general solution for storage deadlocks, and the only slightly tricky thing about such deadlocks is figuring out which connection has to be expanded. If you bear in mind that it must be a connection which is full (i.e. its upstream process is suspended on send) while its downstream process is suspended trying to receive from a different port, you'll find it pretty easily.

This dependency on number of IPs occurs in conventional programming, and you are probably familiar with the problem of having to make design decisions based on number of items. Michael Jackson [the writer on programming methodologies, not the singer] gives an example in his book (1975) of a file which consists of two types of groups of records (we would call them substreams), where the substream type is determined by the type of the last record, rather than the first. He gives as an example a file consisting of batches, each with a control record: if the control record agrees, you have a "good" batch; if it doesn't, it is a "bad" batch. The problem of course is: will the batch fit in storage? If it will fit safely, there's no problem. If not, you either use a file, or provide some form of backtracking (going ahead with your logic, but being prepared to undo some of it if it turns out to be a bad batch). In the foregoing I have tried to stress that it's not good enough if the batch fits into storage now - you have got to be sure it will fit in the future!

There is one final type of deadlock which again involves a loop-type network, but where most of the processes are suspended on send. This is more like a resource deadlock than a storage deadlock, and usually arises because one or more processes are consistently generating more IPs than they receive. In this kind of deadlock, giving them more storage usually doesn't help - the network just takes longer to crash. However, in the case of a loop structure like a Bill of Materials explosion, some processes are consistently generating more IPs until you reach the elementary items, so you have got to think pretty carefully about how many IPs may be in storage at the same time. When in doubt, use a file!

If all this seems a trifle worrying, in most cases you will find that you can recognize deadlock-prone network topologies just by their shape. In fact, the very shapes of the above networks constitute a clear warning sign: any loop shape in your network diagram, whether

circular

Figure16.15

or divergent-convergent,

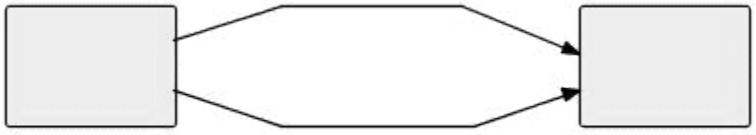


Figure16.16

should be treated with caution, as they are possible sources of deadlocks.

In an earlier chapter (Chapter 8), we talked about how, after you have split a stream into multiple streams, the output streams will be "decoupled" in time. If you decide you want to recombine them, you will usually land up with a variant of the above diagram. Unless you are willing to use the "first come, first served" type of merge (by connecting multiple output ports to one input port), you are creating a very fertile environment for deadlocks. Generally speaking the split and merge functions have to be complementary, but that may not be enough. Let's set up an example. Suppose we want to spread a stream of IPs across 3 servers, and then combine them afterwards. One approach might be to use a "cycling" splitter, which sends its incoming IPs to its output port elements 0 to n-1 in rotation. We will also need a merge which takes one IP from each port element in rotation and outputs them to a single output port. The picture looks like this (setting 'n' to 3):

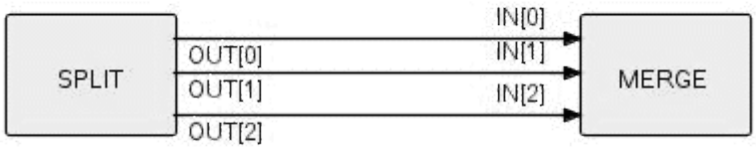



Figure16.17

Now we can add any processing we like at the spots marked X, Y and Z. Well, not quite, as a number of people have discovered. For one thing, MERGE must see 0's, 1's and 2's in strict rotation. If you want to drop an IP, you had better make sure there is a "place-holder" IP in its place, unless you can arrange to drop one or more entire sets of 0, 1 and 2. If you don't do this, at the least your IPs will be out of order, and you may eventually get a deadlock (actually that might be better from a debugging point of view!). The same thing applies if you want to add extra IPs, except that the place-holders will have to be in the other streams. We have found that this impulse to split up streams into multiple ones is a common one among programmers, and it can be very useful. I give this example as a warning of some of the pitfalls which await you when you try to recombine them. In fact, it may not even be necessary! You may find that the assumption that data has to be kept in strict sequence is just a hold-over from the old synchronous style of thinking! [Humans usually like to see data in sort sequence, but computers often don't care!]

I am going to show one more deadlock example, because it illustrates the use of the "close port" service, which has been required by all FBP dialects. Consider the following diagram:

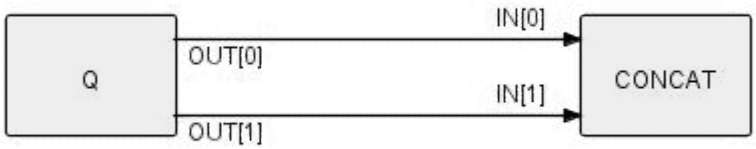



Figure16.18

You know that CONCAT receives all IPs from [0] until end of data, and then starts receiving from [1], and so on. What can you deduce about the upstream process Q? Well, it should somehow be able to generate two output streams which overlap as little as possible in time. We also notice that this (partial) network has a divergent-convergent topology, which suggests it is deadlock-prone. What might cause it to deadlock? One possibility is if Q starts to send IPs to [1] while CONCAT is still expecting data at element [0]. What causes CONCAT to switch from [0] to [1]? CONCAT switches every time it detects end of data on an input port element. But... end of data is normally caused by the upstream process terminating, and Q has not terminated. So Q has to have some way to signal end of data on [0] to CONCAT, so CONCAT can start processing the data arriving at [1] - otherwise CONCAT will still be waiting on [0] while data build up on [1]. What Q has to do is close each of its output ports before it switches to the next one. This will send a signal downstream enabling CONCAT to switch to its next input port.

I feel the example shown in Figure 16.18 is interesting as it illustrates a useful technique for subdividing a problem: what Q is really doing is subdividing its input stream into "time domains", which CONCAT can then safely recombine into a single stream. A lot of sequential files in the business world have this "time domain" kind of structure - e.g. a file might have a header portion, perhaps a list of cities followed by a list of sales staff, followed by a trailer portion. This kind of structure can be handled nicely by splitting processing into separate "time domains".

# Chap. XVII: Problem-Oriented Mini-Languages

> "One of the greatest advantages of little languages is that one processor's input can be another processor's output" - Jon Bentley (1988)
>
> "Another lesson we should have learned from the recent past is that the development of 'richer' or 'more powerful' programming languages was a mistake in the sense that these baroque monstrosities, these conglomerations of idiosyncrasies, are really unmanageable, both mechanically and mentally. I see a great future for very systematic and very modest programming languages" - E.W. Dijkstra (1972)

Problem-oriented mini-languages can be thought of as essentially an extension of the types of parametrization we talked about in Chapter 5. Bentley, like many other writers, has pointed out the importance of finding the right language for a programmer to express his or her requirements. This language should match as closely as possible the language used in the problem domain. If you view the programmer's job as being a process of mapping the user's language to a language understood by the machine, then clearly this job becomes easier the closer the two languages are to each other.

You may have gathered, correctly, that I am not overly excited about the HLLs (Higher-Level Languages) of today. They appear to me to occupy an uncomfortable middle ground between more elegant options at each end of the spectrum. At the low end: machine code - Assembler is fine; at the high end: reusable "black box" modules. Some jobs can really only be done in Assembler (although C is moving in on its turf) - the problem is portability. Black box modules solve the portability problem much more effectively than HLLs - you just port at the function level, rather than at the code level. Historically, HLLs evolved out of "small" languages like FORTRAN by the gradual accretion of features as people tried to extend the languages to new application areas. Most of them are on firmest ground when dealing with arithmetic because here they had traditional notations and experience to build on.

In my opinion, there are three major problem areas common to almost all HLLs:

- the data types of variables,
- constants
- the basically synchronous nature of these languages.

We have already talked about variables in the context of the usual computer concept of storage. Data types we have talked about in the chapter on Descriptors. Now let's talk about constants.

Constants are usually not (constant, that is)! Constants are vastly overused, mostly because it is so easy to hard-code a constant into a program. To take an extreme position, I would like to see constants only used when they represent the structure of the universe, such as Planck's constant or $\pi$ , or conversion rates between, say, kilometres and miles - they're not likely to change. A case can also be made for allowing metadata (data about data) although much of this can be avoided by judicious use of descriptors and generalized conversion routines. I also vote for zero and one - they're too useful (for things like clearing and incrementing counters) to give up! Also to one and zero let's add their Boolean cousins, "true" and "false". But that's the lot!

An ex-colleague of mine *claims* that he knew someone who wrote a program for a company which had exactly 365 employees. You guessed it - an employee left the company, and all the date calculations were off! While this story is probably apocryphal, there is an interesting point here: the only way it could have happened is if the program used the same constant for both numbers. Many shops ban literals, in an attempt to reduce this kind of thing, whereas, of course, it only helps if people are also encouraged to use meaningful symbol names. If, as many programmers do, the constant in question was labelled F365, it would actually make this kind of error *more* likely.

Literals would actually have prevented this kind of thing from happening, whereas calling a constant F365 works the other way! One of the constants should have been NUMBER_OF_DAYS_IN_YEAR (that's non-leap, of course), and the other one shouldn't have been a constant at all! Remember: *most constants aren't!*

So far we have mostly talked about numeric values. There is another class of constants that one runs into quite often in programs: strings which identify entities or objects. Consider a test like

```
IF PROVINCE = 'ONT'.....
```

I would argue for two reasons that this shouldn't be used: the first point is that in HLLs we are forced to compare two character strings - whereas what we would like to ask is (in English):

```
If this province is Ontario,....
```

This may not look very different, but in the first case we are dealing with how affiliation with

Ontario is encoded in a particular field; in the second, we are asking if the entity being referenced is the entity Ontario (with all its connotations). Smalltalk is better in this regard: PROVINCE would be an attribute of one object which contains the handle of another object (ontario) belonging to the class Province, and you can ask if two objects are the same object (==). Another programmer, or even a different field of the same object which needs to reference Ontario, might specify it using the value 2, so you would spend all sorts of machine resources converting back and forth between 2 and 'ONT'. Nan Shu has identified conversions between codes as one of the most common functions performed by business application code (Shu 1985).

The second point is more subtle: even if you can refer to unique objects as a whole, rather than by an indirect encoding, should you? Consider the following sort of code, which we often find in business applications:

```
IF PROVINCE = 'NB' OR PROVINCE = 'PEI' OR PROVINCE = 'NS'...
```

If Canada adds a new province, how do you find all the lists like the above, and how do you decide if the new province should be added or not? What is the concept that makes this set of provinces different from all the others? This is an example of a very common problem with code which is probably responsible for significant maintenance costs in shops around the world. At this point you will probably realize that we haven't done anything yet in FBP to prevent people doing this. And as long as we are stuck with today's computers as the underlying engine, we probably can't, unless we ban constants entirely! What we can do is provide tools to help with this kind of situation (i.e. support the logic which people are trying to implement), and raise programmers' consciousnesses by means of walk-throughs, inspections, apprenticeship or buddy systems, or whatever.

The general concept which I would like to see known and used more widely is what IBM's Bucky Pope calls "class codes". He suggests you first ask what is the underlying concept behind the list; you then build a table or data base, and implement the concept as an attribute of the entities (in this case, provinces). So the above test becomes something like:

```
FIND PROVINCE IN PROVINCE_TABLE
IF MARITIME, .....
```

The overhead goes up a bit, but maintenance costs go down drastically, and since it is now generally accepted that human time is a lot more expensive than machine time (Kendall 1977), it seems short-sighted to keep on perpetuating this type of code, and incurring the resultant costs. By the way, this principle applies equally if only one province happens to be mentioned in a particular list: how do you know that a new province won't show up which shares attributes with the one you've picked?

If we are going to (almost) ban constants from code, put entity attributes and what have been called "variable constants" on disk, why not put logic on disk as well? This gets us into the

domain of rules-driven systems. Remember we said above that FBP gets rid of a lot of the non-business logic, so most of the remaining logic in code should be either general-purpose, e.g. the logic to Collate two data streams, or it should be business-related, e.g. IF INCOME > $50,000, CALCULATE SURTAX USING FORMULA ..... Now, both the criterion and the formula are certainly going to change, and on a regular basis, so why put them in code which requires programmers to change it, after which the changed programs have to be recompiled, approved, promoted, the old ones archived, and so on and so on? It seems much better to put the whole thing somewhere where it can be maintained more easily, and perhaps even by non-programmers. I believe that much business logic, perhaps almost all, can be put on disk and interpreted using simple interpreters.

One important question remains: whether to put our attribute tables, rules, etc., in load modules (created with a compiler and linkage editor) or in data files. The former is really a half-way house, as you still need programmers to maintain them, but at least the information is separate from code, so it can be shared, and is much easier to manage and control than when it is buried in many different code modules. However, I believe putting this kind of information into data bases is a still better solution, as it can be updated by non-programmers, e.g. clerical staff, and it can have separate and specific authorization.

In rules-driven systems, straight sequential logic may not be the best way to express the rules. PROLOG and its derivatives provide a very interesting approach to expressing logic. It should be stressed that logic programming is not incompatible with FBP - I once wrote an experimental component which drove a PROLOG set of rules to perform some logic tests on incoming IPs. The effect was like having a friendly data base, because you could ask questions like "list all the mothers older than 50" even when you had not stored the attribute "mother" explicitly in the data base (just tell PROLOG that a "mother" is anyone who is female and has offspring). The logic programming people have independently explored the possibilities of combining logic programming with parallelism, e.g. such languages as PARLOG.

Now, if we can put our rules out on disk, avoiding such perilous traps as variables and constants, and restrict the code proper to non-business-related logic, then it seems that there really isn't much role for today's HLLs!

So far, we have talked about the basics of most programming languages as being arithmetic and logic, but there exist languages today which address quite different domains. Over the years, we have seen a number of other kinds of specialized languages, such as SNOBOL (pattern-matching), or IPL V (list processing) appear and sometimes disappear. Since in conventional programming it is hard to make languages talk to each other, it is generally the richer ones which have survived. FBP, on the other hand, makes it much easier for languages to communicate, which suggests that what we may see is a larger number of more specialized languages communicating by means of data streams. Based on our experience with human languages, what would be very nice is if they could all share the same syntax, but work with different classes of

objects (semantics). It has been found that humans have a lot of difficulty switching from one syntax to another, whereas we can hold many sets of words independently without getting them confused. This is supported by recent work with bi- and multi-lingual communities - people tend to use one syntax for both languages, or a hybrid, but they can keep the vocabularies quite well separated.

A colleague of mine makes the interesting point about the above scenario that it ties in with the new science of Chaos. If you consider the total application space as chaotic, then you will get areas of order and areas of disorder (Olson 1993). Each mini-language can handle its own area of order, and can communicate using standard interfaces (IPs) with other areas of order. Each mini-language defines its own paradigm - while no paradigm should be expected to do the whole job, judicious combination of many paradigms is often highly effective.

I would argue that, any time a set of parameters reaches a certain level of complexity, you are approaching a mini-language. The parameters to the IBM Sort utility almost constitute a language about sorting, and in fact IBM has added a free-form, HLL-like syntax to the older-style pointer list which it used before. Since decoding the sort parameters is relatively fast compared with the sort itself, it is reasonable to make the sort parameters as human-convenient as possible. The semantics of a set of sort control statements are quite simple, just referring to objects of interest to the Sort (and its user).

Earlier in this book I mentioned a prototyping system which I built, in which you could describe screen layouts in WYSIWYG form. For fun I added a graphics subsystem to it, which let you specify simple pictures. The problem was: what mental model would its users find convenient? I decided to use the idea of colour-filled polygons, which had been used successfully by the Canadian videotext system, Telidon. I later added the ability to have curved edges, as well as straight ones, plus various fill patterns, such as cross-hatching. While this choice of mental model may seem fairly obvious, I could have used bit maps, straight lines, or some other primitive, but I found the polygon idea seemed to be an especially good bridge between the artist and the computer. Not only was it a good match with the software I was using (GDDM), but also I was pleased to discover how fast I could develop a drawing or modify an existing one. The artist I was working with found that he was able to adapt to this medium, and I was impressed with the quality of what we turned out, working together! I use this perhaps rather simple example to make the point that it is the mental model which is important, and which makes the most difference in how usable people find a tool. Humans are visualizing creatures, and, if they have trouble developing a mental model of your tool or system, it will never become real to them, and they will have endless trouble with it.

If you have spent the last few decades labouring with a conventional HLL, by now you may be wondering how one can do anything without using variables or constants! Well, quite a bit, as it turns out. In a later chapter (Streams and Recursive Functions), we will talk about a style of programming which is called generically applicative or functional. When you combine the idea

of functions with recursive definition, it turns out that you can express quite complex computations without ever using a variable (Burge 1975)! As for constants, as long as we put most of them on disk, I'll be quite happy! Of course, I am not suggesting that tax specialists have to learn recursive programming in order to be able to describe tax calculations to the computer, but, based on various experiences, I have a strong intuition that, by judiciously combining a number of the ideas described so far in this book and some that are about to be described, we could develop user-friendly languages which would be decades ahead of the rather user-hostile tools we are forced to use today.

In the next chapter I am going to describe some work I did on a suggested approach to a mini-language for describing business processes running in an FBP environment, taking advantage of the fact that there are other precoded FBP components (such as Collate) to do a lot of the hard work. It is applicative, in that it doesn't use variables but defines its outputs purely in terms of operations on its inputs. This is not supposed to be a definition of a complete language, but more of a sketch of how a language might look which breaks with many time-honoured but rather shop-worn traditional solutions.

# Chap. XVIII: A Business-Oriented Very High Level Language

The following is a sketch for a different kind of language - one that describes business logic descriptively, rather than procedurally. I was reading an article by M. Hammer et al. (1977), describing a system which they called the *Business Definition Language* (BDL), and I started thinking that some of BDL's basic ideas were really complementary to FBP - FBP doesn't have a native way to express business logic, while FBP provides simple solutions to some of the awkwardnesses that I spotted in BDL. BDL might be described as a tabular/functional approach to expressing business applications, which embodies a number of desirable program attributes listed by B. Leavenworth in his 1977 paper:

- elimination of arbitrary sequencing (sequencing not dictated by the data dependencies of the application)
- pattern-directed structures (non-procedural specification)
- aggregate operations
- associative referencing

Of course, these are embodied in BDL, which the following language is modelled on, but it seems to me that FBP can make BDL even more natural, so these attributes should carry over to the resulting language.

I want to expand on these four points a bit. The following are my interpretations, but I hope that I don't depart too far from the spirit in which he intended them!

Leavenworth sees the first point, "arbitrary sequencing", as one of the serious problems with procedural languages, and of course it has come up frequently in various forms in this book. As we have said before, the von Neumann machine forces you to fix the sequence of every statement, *whether it matters to the logic or not*. Consider the two statements

```
MOVE A TO B
MOVE C TO D
```

Clearly it makes no difference to the logic what order they are executed in, but change the first statement to refer to C or D, and you reverse them at your peril! Thus,

```
MOVE A TO C
MOVE C TO D
```

is totally different from

```
MOVE C TO D
MOVE A TO C
```

It is also worth noting that much of the optimizing logic in compilers involves trying to determine which statements can be moved and which ones cannot.

It should also be clear in the light of what has gone before why "pattern-directed structures" are desirable. Humans are much better at working with descriptions (especially visual ones) than lists of instructions.

"Aggregate operations" - this is closely related with trying to avoid procedural solutions. We have found that the higher level the constructs the language deals with, the more powerful it is, and the less of a barrier there is between the programmer's concept and its expression in the language. Of course, higher-level constructs require higher-level operations.

An example I have always found interesting is the APL language. It is highly expressive and a lot of its power derives from the foundation work that Ken Iverson did investigating the basic theory of matrices. While most of us were taught to treat matrix multiplication as an atomic operation, Iverson found a simple yet powerful way to make visible the fact that it is really a particular relationship between two binary operators, which he represented using the "dot" symbol. This is a second-order operator, as it works with other operators, rather than variables. Not only did this allow other binary operators to be combined in the same way (like "or" and "and", or "min" and "max"), but it freed up the simple "multiply" operator so it could be generalized smoothly from scalars to vectors to matrices.

As an example of aggregate operations, let's compare the way PL/I and APL handle matrices. In PL/I you can write the following statement:

```
A = A + A(2,3);
```

This is implemented in a very procedural manner. PL/I executes this statement one element at a time, and does it in such a way that the rightmost dimension is the one which cycles most rapidly. So the sequence will be `A(1,1), A(1,2), ..., A(1,n), A(2,1), A(2,2), ...`, and so on. When these additions hit `A(2,3)`, all "later" elements (those following `A(2,3)`) will have the new (doubled) value added to them, rather than the original one. In APL, on the other hand, matrices are treated as "aggregates", which are conceptually handled all at the same moment of time. Thus, you can write

```
A <- A + A[2;3]
```

and it behaves much like the PL/I example, but `A[2;3]` does not change halfway through execution. This of course means that APL has to be able to manage matrices as more or less opaque objects, while the PL/I model is more that a matrix is just another part of the machine's memory, with a particular structure superimposed on it. In FBP, although for performance reasons we have to process a stream one IP at a time, you very often can *think* of it as a single object. If you really have to process a whole stream as a single object, we have seen in the earlier chapters that you can turn it into a tree, and then turn it back later. In the chapter on Streams and Recursive Functions we will see how one can use recursive function definitions to effectively treat a whole stream as a single aggregate object.

"Associative referencing" means the ability to locate an object by content, rather than by location. On most modern machines, this is done by means of a look-up of some sort, although there have been machines designed which did it by hardware. A few years ago there was a lot of interest in associative or "content-addressable" memory. While it is certainly nice to be able to find things that have moved, my feeling is that "handles" are fine, provided you have a way to find new things which you have never searched for before. Once they are found, they shouldn't have to move again, and their handles can be passed from process to process. An additonal

wrinkle is to leave a trail in the rare event that things do move (of course in both cases we are talking about a single shared memory). It is interesting that the main difference between Carriero and Gelernter's Linda (1989) and FBP is that the former is associative, while FBP, as we have seen, uses connections and handles. We will be looking at Linda in more detail in a later chapter, but it seems to me that Linda's design should increase its overhead considerably over FBP, without a corresponding advantage in expressiveness. An example in one of the articles about Linda describes how to simulate an FBP connection, and it would be trivial to write matching FBP components which store and retrieve Linda tuples! Of Leavenworth's four points, this last seems to me the least exciting, but it should certainly be borne in mind as a design technique.

To give you a flavour of BDL, here is a sample definition for the "extended details" file (or stream) in Leavenworth's paper. This is the same TR1 that was used in Chapter 10. In what follows "'s" is the BDL notation denoting a collection or group.

| Specification | | Glossary | |
|---|---|---|---|
| Name | Derivation | Name | Definition |
| 1 Ext detail's | ONE PER Detail | Detail's | INPUT |
| 2 Ext price | Unit price x Quantity | Unit price | IN Product Master |
| | | Product Master | Master WITH Master Product number = Detail Product number |
| | | Master product number | IN Master |
| | | Detail product number | IN Detail |
| | | Detail | CAUSE OF Ext detail |
| | | Master's | INPUT |
| | | Quantity | IN Detail |
| 2 Salesman | Old Salesman | Old Salesman | IN Detail |
| | | Detail | CAUSE OF Ext detail |
| 3 District | Old District | Old District | IN Detail |
| | | Detail | CAUSE OF Ext detail |

[I shall continue to use their term "extended price", although a better term would be something like "extended monetary value".]

I have labelled the columns as shown to give an indication of their purpose and interrelationships (BDL proper does not do this). The first two columns express the hierarchical structure of the file in question and the derivations of each of the entries, while the last two columns give definitions

of all names appearing in the second column, and of any new names introduced in these definitions. The fourth column uses some reserved words to describe relationships between words in the first column and their sources. Thus, the derivation of "Ext price" is shown in the second column, and uses "Unit price" and "Quantity", whose definitions are in turn shown in the third and fourth columns. The whole approach is based on *definitions*, rather than *procedures* , which I found very appealing.

The structure shown in the first column is considered to repeat indefinitely through the "Ext detail's" file.

If a job step produces more than one output file, a BDL "program" similar to that shown above must be given for each one separately. Clearly, since each one is "driven" by the same input file, there is unnecessary duplication of logic here. In fact, in an attempt to avoid uncontrolled "global" definitions, the BDL language is considerably more redundant than one would like - every duplication increases the risk of one or more of the copies getting out of step.

A more subtle problem arises from the lack of an explicit stream concept: in column three of the referenced diagram, "Product Master" is defined as that Master record (hopefully unique) whose "Master Product number" field is equal to the "Detail Product number" in the Detail record. WITH is an example of what Leavenworth calls "associative referencing" - it describes an implicit matching process. However, there is no indication of *how* the appropriate master record is located. While this is to some extent deliberate, when we actually come to build an application, we will eventually have to choose between various techniques for relating details and their corresponding masters, e.g. direct access, sort and merge, and perhaps others, which will have a profound effect on the basic structure of our design. It is true that the software could be allowed to select the technique automatically, but in our work with FBP we tend to feel that it is better to leave developers free to choose whichever technique seems most appropriate. Remember there will be prewritten, "black-box" code available to support whatever approach they select.

In what follows, I will assume that sorted streams of data have been merged using Collate, so the master records will precede their associated details. I will now sketch out a possible mini-language which holds to the spirit of BDL, while taking advantage of the fact that we are only describing a single FBP process.

Three types of non-procedural information together describe what you want this kind of process to do:

- a description of the input and output streams, down to the IP level, and the creation and output criteria for the output IPs (let's call it an IP Relationship Diagram)
- descriptions of the various IPs involved (IP Descriptions)
- description of the calculation basis for "derived" fields - fields not present in the input IPs, or whose values are changed by the component (Derivation Descriptions).

A free-form notation is used in the accompanying figures, using English-like names for fields, and upper-case strings for "reserved words".

In what follows, square brackets will indicate that a clause seems slightly redundant and could easily be eliminated by adding a new rule to the set of evaluation rules. These are usually reasonable defaults, which can still be overridden if the developer needs to.

I will now show the three portions of a Component Description listed above, using the component TR1 from the above example.

Here is the IP Relationship Diagram (IRD) for TR1 first:

```
INPUT STREAM
1. Merged Input
 2. Product Substream: REPEATING
  3. Product Master
  3. Sales Detail: REPEATING
OUTPUT STREAMS
1. New Masters
 2. Product Master: ONE PER Product Substream
1. Sort Input
 2. Sales Detail: [ONE PER Sales Detail]
1. Report-1
 2. Product Summary: ONE PER Product Substream [,NEW]
```

This table describes relationships among IPs, so the lowest level items at any point always describe IPs - all higher level items are substreams, while all level 1 items are streams.

Only one input stream may be specified in the IRD, but there may be any number of output streams.

All streams consist of repeating patterns of IPs, and a COBOL-like "level" notation is used to show how they are nested to form a complete data stream.

The repeating pattern may be trivial, e.g. the stream may consist of only one IP, or it may be as complex as desired, consisting of a variable number of substreams, each of which consists of a variable number of substreams, and so on. In what follows, unless explicitly stated otherwise, the word "substream" will be taken to allow a single IP. This way I don't have to keep on saying "IP or substream". Similarly, "substream" can also refer to the whole stream, except where explicitly stated otherwise.

A substream in an IRD always has a "quantifier", which may, for an input stream, be one of the phrases REPEATING, OPTIONAL or ONE-OR-MORE, or absent; for an output stream, it may be either ONE PER or absent. These quantifiers describe the number of occurrences of the substream in question *within the next higher level substream*. If no quantifier is specified, then there is exactly one such substream in the next higher level substream. REPEATING,

OPTIONAL and ONE-OR-MORE mean respectively: zero or more, zero or one, and at least one occurrence. ONE PER 'x' indicates that the substream in question is to be generated once for each occurrence of substream 'x' (let's follow BDL in calling 'x' the "cause" of the substream being described).

Examining the IRD for TR1, we note that the input stream consists of zero or more Product Substreams, each of which consists of one Product Master, followed by zero or more Details. You will remember that substreams are delimited by bracket IPs, so a Product Substream is additionally marked by the presence of a Product Master. We have run into this kind of redundancy before, but it doesn't hurt, and, if the Product Master had been OPTIONAL, the brackets would be the only way to separate Product Substreams.

In the output streams, two of the IP types (Product Masters and Sales Details) actually came from the input stream, and are referred to as OLD, while the Product Summary IPs are created by TR1, and are called NEW.

The PER clause on Extended Details is shown in square brackets because a reasonable additional evaluation rule might be: if the PER clause is omitted on an OLD IP, the IP is its own "cause".

Now we will need some kind of description of the IPs. This has been discussed at length in the chapter on Descriptors, so I will not go into much detail here. The designers of BDL also saw the need to allow a more varied set of field types than conventional HLLs. A Sales Detail IP could also be described using COBOL-style level numbers to show relationships between fields, but the positioning of the fields is only important when importing or exporting files. The other main function of level numbers - to show groupings - either goes away when one has more powerful field types, or can be handled in other ways. We might therefore show the fields of an Extended Sales Detail IP as a simple list, e.g.

```
Extended Detail:
{
  Product Number: IDENT,
  Salesman Number: IDENT,
  District Number: IDENT,
  Quantity: QUANTITY,
  Unit Price: $CDN,
  Extended Price $CDN;
}
```

The last thing we have to specify is the derivations of any derived fields. The following diagram shows the algorithms for those fields which are changed or created by the action of the component in question. Fields which are unchanged from the values they had on entering the component are not shown. Indentation is used to relate fields to the IPs they are part of.

```
     Name              |       Derivation
-------------------------------------------------
                       |
 New Master            |
   Year-to-date Sales  |   Year-to-date Sales
                       |      [IN Product Master]
                       |         + Product Total
                       |
 Extended Detail       |
   Extended Price      |   Unit Price * Quantity
                       |
                       |
 Product Summary       |
   Product Total       |   SUM of Extended Price
                       |       [OVER Product Substream]
                       |
   Year-to-date Sales  |   SAME as IN New Master
                       |
```

When a field in a "new" IP is not mentioned, it is assumed to have the same value as the field with the same name in the "cause" IP.

The function "SUM of A OVER B" sums all occurrences of field A within higher-level substream B (which is called the "range" of the OVER). Thus, Product Total is defined as the sum of Extended Price OVER Product Substream. The OVER-clause is shown in square brackets in this example, as it becomes redundant if the following rule is added: if OVER is omitted, the "range" is taken to be the "cause" for the field being computed. Although this may seem arbitrary, in fact it covers off all the SUMs so far encountered in sample problems.

The IN in Year-to-date Sales is similar to qualification in COBOL or PL/I. It can be omitted if it is understood that it is the field of the same name in the "cause" that is meant.

"SAME as" is self-explanatory.

The Product Summary report poses a different kind of problem as it has to do with printing a human-readable report. Of course, screens are also human-readable, so many of the same solutions should apply to both. There are many possible approaches to parameterizing a report, and, under FBP, there is no need to decide on a single unique solution, as many printer components can all coexist in your library of components, just as a carpenter can have many different kinds of glue for different jobs. However it is not hard to imagine that a useful component would be a "report interpreter" similar to the interpreter which interprets the derivation rules. I am not going to show how reports could or should be defined, but am rather going to mention a few ideas that might be considered in designing such general components.

We have already seen in Chapter 10 what a useful Report generator component might look like, so we will concentrate on the components which generate the report lines which are fed into it. This of course brings in the question of "representations" that we talked about in Chapter 11 (Descriptors). There we talked about how representation standards normally come in layers: in a given application you may have an international standard, a national standard, and a company standard, and a particular report may even use one or more such representations for the same domain, e.g. amounts with and without separators. Since reports and screens have to be human-readable, you also have the problem of multilingual support, and two very noticeable differences between languages are: 1) that phrases in different languages are frequently of different lengths, and 2) that values may have to be imbedded in text in different orders.

The above-mentioned problems have been solved quite well in the area of message-handling. A common solution is to simply provide "skeletons" which contain the fixed information and have "insertion points" for variable information. This assumes that most of the variable information is numeric, but it covers most common reporting requirements. So a report or a screen would probably be like a set of messages with fixed columns (and maybe rows) added.

Reports (and screens which reflect reports) also will have subtotals and total lines, which, as we have seen, can be driven by special IPs generated upstream at close bracket time. There are a number of report generation tools, of which RPG is one of the oldest and best known, which can act as sources of mental models. The report design files or screens should resemble the final document in layout (WYSIWYG), but with the addition of IP type information (e.g. heading, detail, total for such and such level), and codes for common variable information, such as date, time, page number, etc. Such a design can then be encapsulated in a component, or maybe more than one - remember, you are not restricted to only one report generator! Similar specifications are showing up on in popular PC applications - for instance, the Report Generator function of the Database subsystem of Microsoft's Works™.

The foregoing is just a sketch for a mini-language that might do some of what people use HLLs for today. Because FBP allows us to mix languages freely, no one language has to be able to do everything, so we can design languages to specialize in different subject areas. By the same token, the same mental model is not valid for all possible uses - if you want to do some logic by pattern-matching, you do not have to do it all by pattern-matching! Similarly, while mathematics is a great foundation for some types of programming, that does not mean that all programmers must become mathematicians. Lastly, a language can be expressive and still be rigorous. If the mental model is one that people find easy to grasp, it should enhance productivity, rather than just becoming an extra burden on the developer. It is absolutely crucial that we shift a certain amount of the work of developing applications to people who are not necessarily professional programmers, both to reduce the DP bottleneck, but also to take advantage of their expertise.

The various diagrams and lists that we have seen above provide the advantages of applicative programming notation without the complexity of nested functions which are normally required in

such a notation: FBP does this by making explicit the timing relationships of the different processing events, and the derivational relationships of the IPs involved in that processing. The righthand side of a derivation specification is in fact a functional expression of any desired complexity (it can even include conditional parts analogous to the *cond* function of applicative programming), and such a specification conforms to the "single assignment" characteristic of applicative programs.

Because of the open-endedness of the FBP architecture, a component of this type does not have to cover all possible applications, and can be part of a total application comprising both custom and off-the-shelf components. One can easily visualize a whole range of such components relevant to different application areas, essentially acting as repositories of specialized knowledge about these areas. If one component is more powerful than another one or covers a wider application area, a sort of evolutionary "survival of the fittest" would be expected and is in fact desirable. The problem is that some dinosaurs have survived far beyond their "natural" span because of the enormous inertia of our environment, rather than because of any intrinsic superiority!

# Chap. XIX: On-Line Application Design

An increasing number of computer applications are interactive - that is, they have to communicate with an end-user, with the result that some (but not necessarily all) application processes must be geared to the pace of the end-user. An end-user will enter commands and data on a screen, select among options, etc., and results will be displayed on the screen. Usually these events alternate, but some displays occur unexpectedly, and the user must also be able to interrupt processes or switch to other activities. Hardware and software environments vary: the screens may range from "dumb" terminals to graphics terminals to programmable work-stations with powerful logic capabilities; terminals may be connected in various ways to the host; and finally the host may be running different software platforms (e.g. IBM's CP/CMS, TSO, IMS or CICS), or 4GLs based on them. While it is hard to generalize between all these variations, we can say with some assurance that there will always be a need for a program to be able to present text, numeric and alphameric information to a screen, and for the user to be able to send commands and data to the computer. There is also a need to format a screen, including specification of fixed or user-specifiable information on one or more screen and window layouts. The user and program must further be able to select among different such layouts.

In Chapter 14, I talked about "loop"-type networks supporting one interactive user (as in the IBM TSO and CMS environments), and how this changes when we move to multi-user environments such as IBM's IMS/DC (of course, TSO and CMS support multiple users, but in their case a whole program is dedicated to a single user, while an IMS/DC program supports a series of different users one at a time). In what follows, we shall go into more detail on IMS/DC, and also concentrate on one type of screen, the 3270 (without graphics capability). I believe these concepts generalize well to other environments, even to the new GUI-style interfaces with drag 'n' drop interaction (by assigning a different process to each window), but this chapter would get much too long if we tried to cover even just the IBM host environments!

First I want to describe at a high level how TSO (or CMS), CICS and IMS/DC differ. By the way, IMS/DC is now IMS/TM, but I will continue to use the older term. The basic problem they

all address in different ways is how to trigger program activity as a result of a user action on a screen or keyboard. In TSO (or CMS), the user has a whole program dedicated to his or her use, which multithreads with other such programs. During his or her "think-time", the program has to wait, but the other parallel programs (either supporting other users, or running batch) can use the available CPU time.

CICS and IMS/DC are both very popular for IBM hosts, and they are both delivering very respectable transaction rates. Traditionally, CICS has had lower overhead but provided less protection between users. These distinctions may be breaking down as these systems evolve. In the case of CICS, CICS runs a number of parallel tasks. When one of these tasks is triggered by a user's action, it executes the appropriate logic. After output has been generated, the task can be suspended ("conversational" processing), or can terminate after user-oriented information has been saved ("non-conversational"). If all code was conversational, a CICS program could only support as many users as there are tasks, so this mode is not recommended - especially since human think-times are long compared with the CPU time required by the machine. CICS, like all FBP implementations so far, can suspend a task at any API call. In fact, allowing tasks to be suspended elsewhere than at an API call would make FBP (and CICS) programming much harder, so it is unlikely that this will change. I have always felt that the internal logic of CICS is so similar to that of FBP that a hybrid system combining ideas from both might be extremely interesting.

IMS/DC also uses a number of parallel programs, but, unlike TSO, each program services multiple users, one after the other - each user "occupies" a program from the time a screen or keyboard action takes place until the response has been sent back to the screen, at which point the program is free to service a different user. In fact an IMS program cannot wait for the user to finish thinking - if there is no work waiting to be done, it just terminates. So IMS/DC cannot use CICS's "conversational" approach - while it also uses the term "conversational", it has a different meaning. Also IMS programs run in separate regions, so they can interrupt each other preemptively; whereas CICS tasks can only lose control at an API call.

IMS transactions are driven from a "message queue". Each message was added to the queue as a result of a user taking some action, such as hitting ENTER, a Program Attention key or a Program Function key at his or her terminal. IBM 327x terminals are "buffered", meaning that information is accumulated on the screen until one of the above keys is hit, at which time all modified data is sent to the host, together with an indication of what action was taken by the user, and where on the screen the cursor was. All this information is collected and put into an "input message", which is then placed on the message queue. In addition each message contains an 8-character code indicating which processing is to be applied to this data. Very often it is used to identify the screen which was being displayed when the interrupt occurred, allowing the program to select the processing. This code is called the "transaction ID".

The transaction ID is used by IMS to select, based on rules the installation has specified, which

of a set of programs, called "message-processing programs" (MPPs), is to service this transaction. The MPP may already be running: if not, it will be started in an available "message-processing region" (MPR).

When program logic decides that a new screen is to be displayed on the terminal, an "output message" is put on the message queue by the transaction, containing the data to be displayed and the position at which the cursor is to be positioned. The program usually specifies which screen layout is to be used for this.

An MPP processes serially the transactions it is supposed to handle. It continues picking its transactions off the message queue and putting output messages onto the queue until one of the following occurs:

- there are no more transactions waiting
- a higher priority transaction needs the region
- it reaches a predefined limit called the "program limit count" (PLC).

When one of these occurs, the MPP's command to get the next transaction fails with a specific return code ('QC').

Now note that the MPP serially processes transactions from different users, rather than being dedicated to one user. Also a given user's interaction with the host (main-frame computer) will jump about from one screen to another, and therefore from one MPP to another, and therefore from one message-processing region (MPR) to another. This means that any information which has to be carried from one screen of such a "conversation" to another has to be held on disk, or in a storage area which is reserved for that one user (IMS provides such an area, which is called the "Scratch-Pad Area", or SPA). Some applications also use the message itself for this kind of information, as not all the message information actually has to be displayed on the screen.

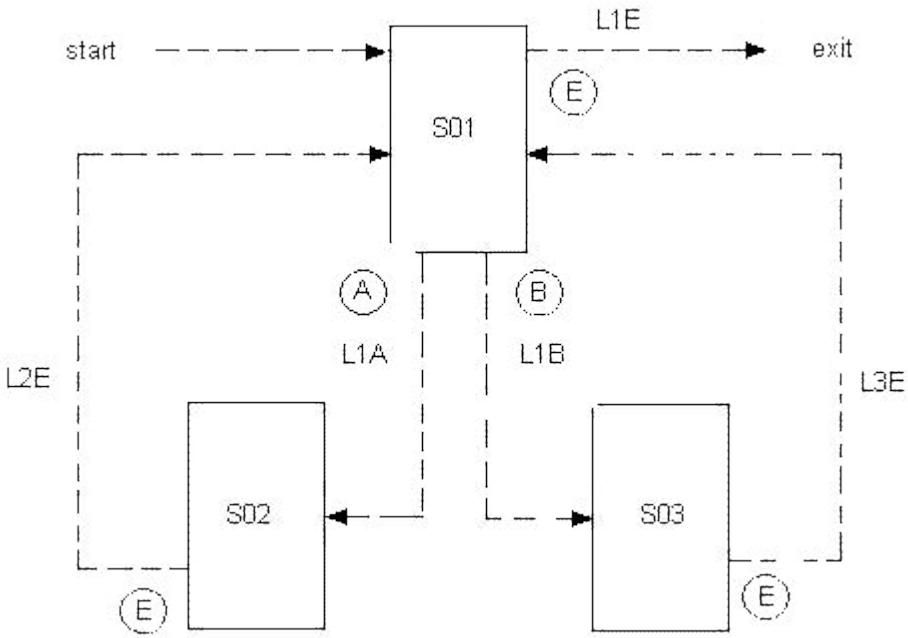
While there are a number of other types of IMS/DC application, such as WFI, pseudo-WFI, etc., the above sketch will suffice to give a background for building on-line applications using FBP.

Note that the term "transaction" in IMS is quite ambiguous - we will try to avoid using it since the terms "MPP" and "input message" cover most of its meanings. One other usage of the term "transaction" means the processing within an MPP which runs from the reading of an input message to the writing out of an output message. This processing is dedicated to a single user and hence it is important that no data belonging to another user be picked up inadvertently, even though a given MPP may service a number of users before it terminates.

An interactive application differs from a batch one primarily in that some (not necessarily all) of its logic is synchronized to the speed of the user. If you visualize an interactive application as an alternation of screens and processes, the screens can themselves be treated as processes whose job is to convert between internal and external data formats. Of course, these processes also

allow selections to be made from menus and lists, plus requests for certain general services, such as Help or Return to Previous Screen.

Let us use a very simple application consisting of 3 screens as an example. Diagramming the flow between screens, we get:

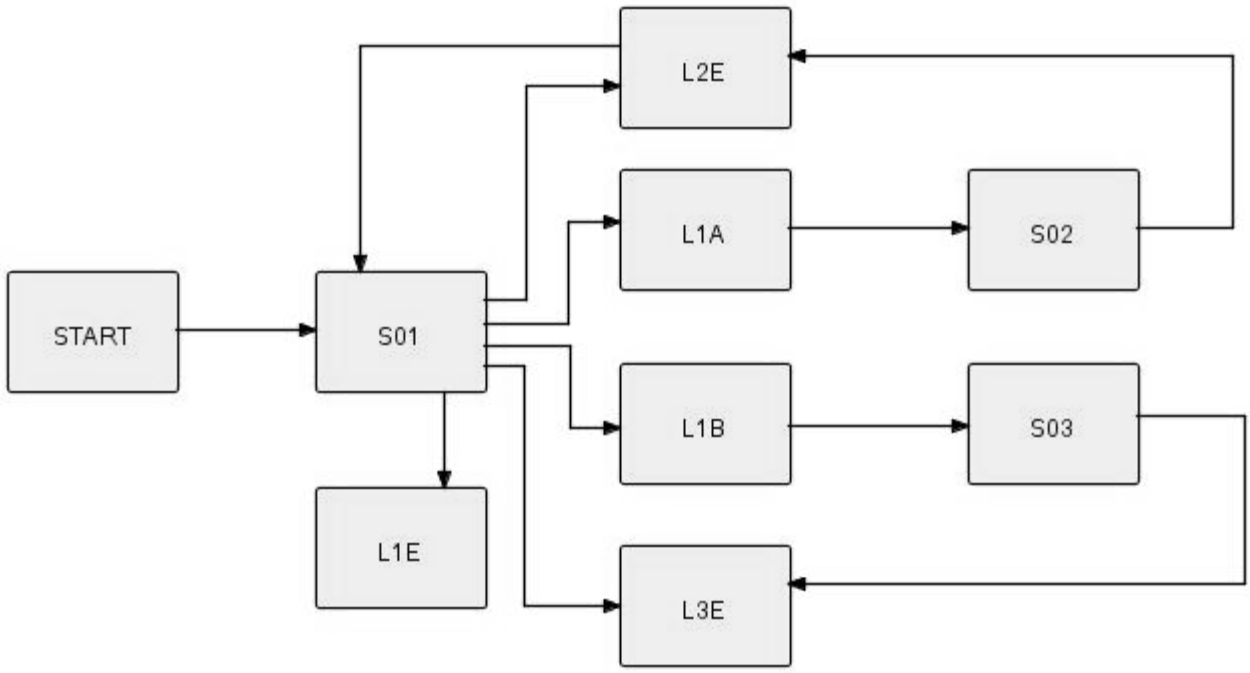



```
where S01, S02 and S03 are screens
      A, B and E represent user actions
      L1A, L1B, L1E, L2E and L3E represent logic
```

(user actions are commonly either PF keys being pressed or commands being entered)

Figure 19.1

The next stage is to convert all the screen blocks to processes and add in the logic processes (plus one to start the network). This gives us the following diagram:




```
where S01, S02 and S03 are screen processes
      A, B and E represent user actions,
      L1A, L1B, L1E, L2E and L3E represent logic processes
Figure 19.2
```

In the above diagram, the screen processes must output the IP streams which the logic processes expect, and logic processes must output the streams which the screen processes expect. In designing the application, the designer must coordinate the screen designs, IP streams and logic processes.

There is another consideration: a typical application may have quite a few different screens and expert users increasingly want to be able to jump from any screen to any other, without being required to go up and down through menus. In other words, the network becomes more and more thoroughly interconnected. Through all of this, the designer must make sure that each screen process gets the data it expects. Just as with batch applications, it is best for the designer to concentrate on the data flows, rather than on the processes.

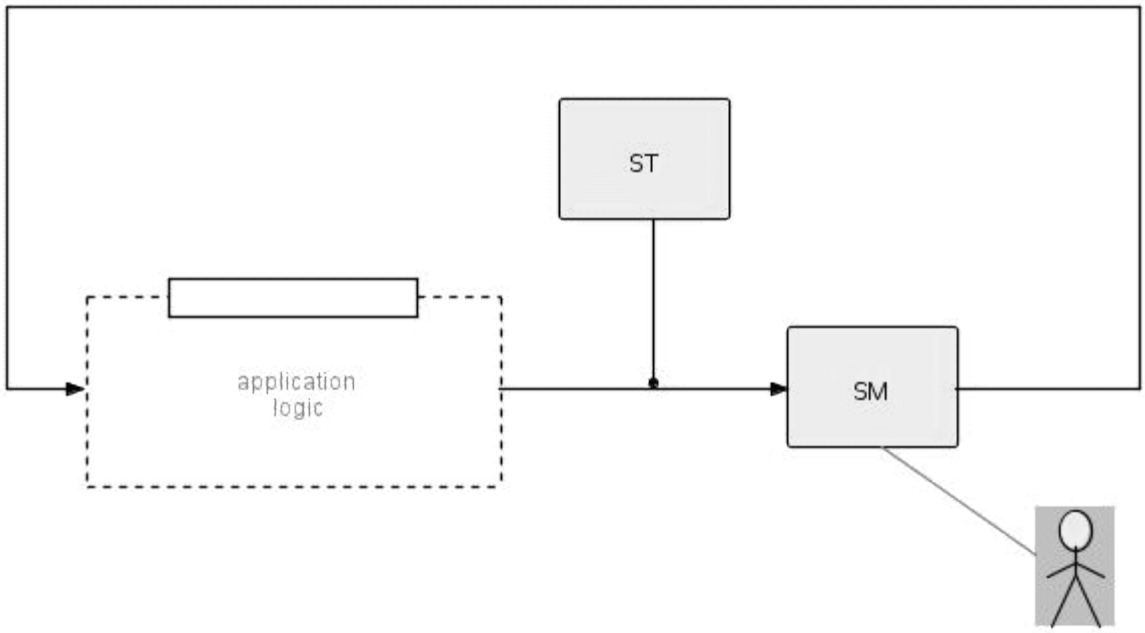
Now, if the network structure reflects the possible paths between the screens, it is clear that it will become more and more complicated as the connectedness between screens increases. You could eventually get a network where every process is connected to every other process, so that

the network no longer provides any assistance to the developer in understanding what is going on. Instead of using the network to define the flow between screens and logic, we have found that it is much better to do this with a table. This table specifies the screens and processes resulting from each possible combination of screen displayed and user action taken. This will allow the application to be "grown" in a natural way, without a corresponding increase in network complexity. The network shape arising from this approach actually becomes simpler, as we will have moved much of the complexity into a table. The network topology will now usually be a loop, comprising processes to display a screen, analyze the user action, trigger any processes, select another screen, and so on.

In conventional batch programming using data flow, the flow of data, and therefore the sequence of data transformations, is predominantly in one direction across the network. This kind of topology arises from the fact that there is no real-time interaction with human beings. The program is started by means of JCL or by an operator command and runs until all the data has been processed.

Increasingly, however, we require programs to interface to human beings, and therefore there will be at least one process in a network that is "paced" to the speed of the human interface. In a batch FBP network, as we have seen above, all of the processes run asynchronously. In an on-line application, a network topology that appears in a number of situations in one-user operating systems, such as CMS and TSO, is the "loop-type" network. We talked about loop-type networks in Chapter 14, so you are already familiar with how these work.

Here is a diagram of such a network:




```
where SM  is a Screen Manager
      ST  starts the network as a whole
      the dotted line represents logic processes
Figure 19.3
```

In this network, there is one process, the Screen Manager, which controls the user's screen. An IP (or group of IPs), conceptually similar to the token in a token-ring type of LAN, travels around the network triggering processes to execute. The bracket IPs provide a convenient way of grouping IPs for this kind of function - typically, we use the first IP of each substream for such information as screen name, name of key struck, position of cursor, etc., and the remaining IPs, if any, for the data. When the substream arrives at the Screen Manager process, it triggers the display of data on the screen, waits at that process until the user responds, then proceeds to the next process in the loop. The data IPs, if any, will hold the data to be displayed on the screen, and will receive any data that the user enters on the screen. The first, or "request", IP can also contain an indication of which key was pressed to tell the Screen Manager that the user has responded (e.g. function key, enter, etc.), where the cursor is and perhaps also which fields were modified (this information is often of interest to the host application).

The next process in the network will usually be a process which analyzes the response and takes appropriate actions, perhaps routing the request IP (with its accompanying data IPs) to a process

which will do the appropriate application processing.

Eventually the application logic will request that another screen (or the same one again) be displayed, and the request IP will be sent back to the Screen Manager to achieve this. This then is a very standard topology that you will often run into when building interactive applications in one-user environments.

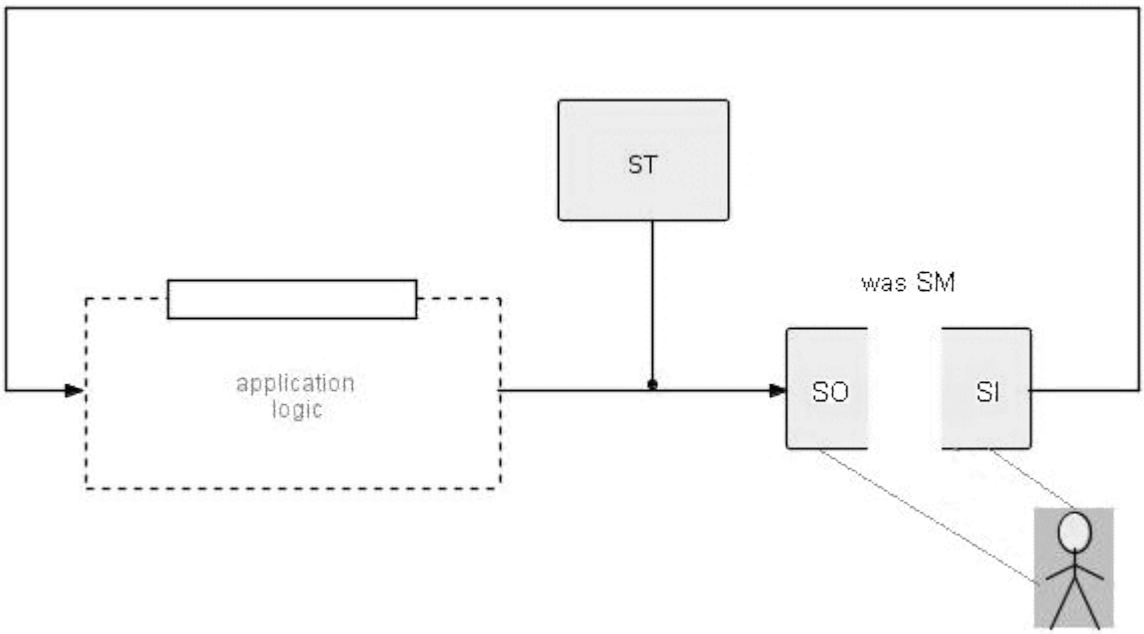
We now have to convert this type of logic into IMS transactions. We shall see that IMS transactions are also loops, but the function of the loop is slightly different.

We have seen above how a data IP can be associated with a request IP and then act as a carrier for the screen's variable data. If we temporarily ignore the request IP and simply picture the data IP triggering a screen display, we see that this is very much the way an IMS transaction asks for a screen to be displayed: an IMS "output message" containing the data to be displayed is generated by the program and is sent to the message queue. This signals IMS/DC to display a screen (conventional programs specify the screen name using a call). When the user responds, IMS/DC places an "input message" on the message queue. Soon afterwards, a transaction is triggered and the transaction program code, the MPP, gets the message from the message queue and processes it.

Let us take as an example the screen flow that we used above. Look at Figure 19.2, remembering that the boxes represent screens, not processes. Each screen in this diagram, plus its downstream logic, now potentially becomes a separate transaction. We have to convert the above screen diagram into IMS transactions by "cleaving" each box representing a screen into two pieces: an "output" piece and an "input" piece. This leaves us with a number of "batch-like" networks, with a screen input process on the left, and one or more screen output processes on the right. It is IMS/DC which provides the linkage between them. These screen handling processes appear to the transactions just the way File I/O appears to a batch program.

Another way of thinking about Figure 19.2 is that each "screen box" in this diagram is in fact a process which writes to the user's terminal, waits for a response and then sends the input from the terminal onwards (it is the "complement" of a logic process). Such a process can thus be split into two processes: one to put out to the terminal and one to get from the terminal. In between these two functions, it is too expensive in the IMS environment (unlike one-user environments like CMS and TSO) to have the whole region wait during the user's "think-time", so we essentially terminate the section of code processing that user, and restart another transaction when he or she finishes thinking and takes some action. (As we said above, ending a transaction does not necessarily mean ending the MPP). The result is a set of "batch-like" networks, with Screen Input at one end and Screen Output at the other.

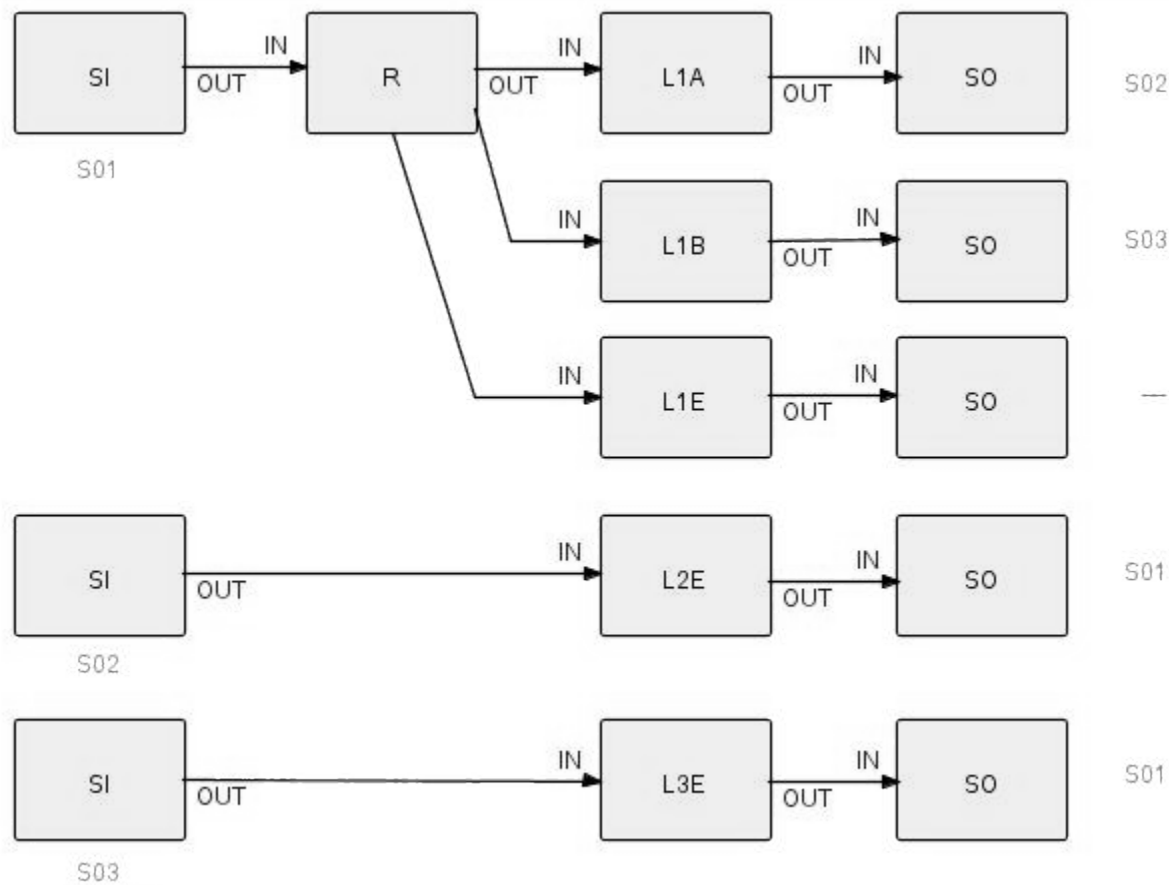
The following diagram shows a single "screen process" being split into separate output and input processes:




```
where SO  (Screen Out) puts information out to screen
      SI  (Screen In)  gets information back from screen
Figure 19.4
```

If you look at the "loop" in the above diagram, you see that it is topologically a straight line starting with SI and ending with SO. Each of these "opened up" loops becomes a separate message processing transaction.

Here is a diagram of a number of IMS transactions resulting from cutting up the screen flow shown in the above example:




```
where SI   (Screen In)  handles IMS input messages
      SO   (Screen Out) handles IMS output messages
      R    is a Selector which decides whether L1A, L1B or L1E
           is to be given control
      L1A, L1B, L1E, L2E and L3E are instances of the logic
           "legs" in the previous diagram
Figure 19.5
```

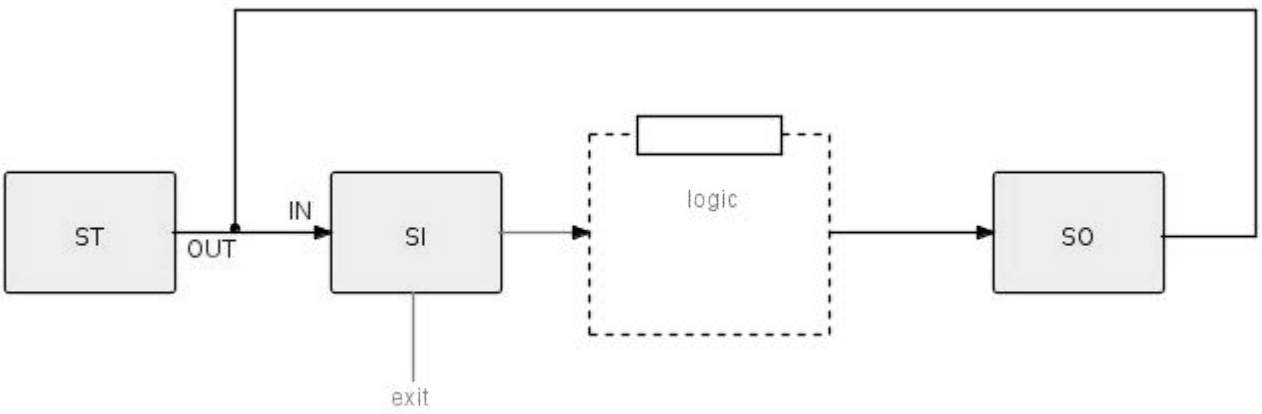
In this diagram I converted the application logic into a network structure. Clearly, this would not result in a very maintainable application - as I said above, it is much better to use a single network, and encode this kind of information in tables. Just as, in an earlier chapter, we did not want to encode the number of Canadian provinces into the network structure, we should avoid imbedding the screen flow into the network structure.

To keep the structure flexible, we will need to hold the output screen name or transaction ID in the "request IP", and allow this to trigger the display function. This allows us to have a single instance of *SO* which is completely general, and will dramatically simplify the transaction

networks. In the IMS/DC environment, *SO* will use this information to select the next screen to be displayed. Of course, the data being presented to *SO* must match the specified screen, but this is easy to control because the screen name will be in the same substream as, and followed by, the data that relates to it.

As I said above, the loop topology turns up again, but with a slight twist, when we actually try to build an MPP using FBP. The reason for this is that MPPs are normally coded so that they check (using an IMS "get" call) if there are any more messages waiting to be processed before they close down, and it is this check which provides the "sync point" which causes the screen to be displayed and files to be updated. It is not essential for the MPP network to "loop back" but in this case various expensive types of overhead would have to be repeated for every single message. Accordingly, MPPs are usually written so that they loop back to check for more input. Remember, each time through the loop, the MPP services a different user. On the face of it, it would seem that a straight left-to-right topology would be adequate for an MPP, but this will likely result in a second message being read before the previous message has been displayed, and the above-mentioned function of the "get" as a sync point absolutely requires that inputting a message not be allowed to happen before any data base records have been modified and the output message has been written - hence the reemergence of the familiar loop-type topology.

The FBP network for an MPP now looks like this:




```
where SI    is a process which handles IMS input messages
      SO    is a process which handles IMS output messages
      ST    starts the network
     'logic' represents application logic
     'exit'  indicates that SI may bring down the whole
          network if there are no more messages waiting
```

```
          or the PLC has been reached
Figure 19.6
```

Note that, unlike the CMS or TSO case, each iteration around the loop may involve a different user, so the discipline of reentrant coding is particularly important here.

In DFDM we developed components for the functions shown as *ST*, *SI* and *SO*, although these were not part of the DFDM system as it was marketed. These provided an environment which allowed the intervening logic to be coded as though it were simply processing records from a file. In addition, *SI* and *SO* collaborated to support a single message format only for a given screen, instead of two separate ones (input and output) as presently required in IMS. To explain this a little further: while an IMS output format requires all variable fields which may be displayed on the corresponding screen, the input message format only contains the fields which the user may change. People often get around this by telling IMS that even the protected fields have been modified (the screen has an attribute which lets you force the "modified" flag on for a field), but this results in more data traffic between host and terminal than is necessary. Instead, let's make SO save the output message in the SPA or in a user data base, and change SI to use that information both to create a "complete" message, and also to report on which fields have actually been modified. We now have a pair of collaborating components which make the job of handling IMS messages significantly easier!

Analogously to the paradigm shift we saw in batch, FBP also forces a shift from concentrating on screen layouts to concentrating on IP layouts. For a given screen in IMS, there will in general be three layouts: the "in-storage" format, the input message and the output message. However, it is possible to combine these down to two or even one layout, by using the MFS MFLD macros to act as a "bridge". Designers of IMS applications don't generally realize that IMS MFS allows the sequence of fields on the screen to be quite different from the sequence of fields within the area in storage containing the data. This will also allow the same data IP to drive a number of different screen layouts, which is a useful characteristic in on-line systems. One might also, for example, want to use a single data IP and show different parts of it, depending on the authorization level of the user.

One last topic that is relevant to the design of on-line applications in IMS/DC is that of storage of information within a "conversation" (the suite of transactions interacting with a single user to do a job of work). I mentioned above that we can use the message, Scratchpad Area (SPA) or disk storage. The problem with the SPA from a modularity point of view, is that information in it has to be accessed by offset - i.e. one declares a structure to describe the SPA, and all transactions participating in a conversation have to use the same layout. So you essentially have an entity, the "conversation" (not otherwise recognized by IMS), which is tied to the SPA layout. If you then want to share transactions between conversations, you constrain them all to share the same SPA layout. This is another form of the "global" problem. Since FBP forces modularity, we have to

find a way around globals, and we did in one of our applications this by storing data associatively in a storage area associated with the user - initially the SPA and later a special data base) - using data areas chained into a list and identified by 8-character names. We defined three components [reusable code modules], which could be used at will in the application networks. We could call these *Input from Area*, *Output to Area*, and *Free Area*. When we wanted to store a piece of data for later use, we simply sent it with an identifying name to an occurrence of an *Output to Area* component, which either replaced the data in an existing area of the same name, or created a new area. *Input from Area* was used to retrieve data, given an area name. The interesting thing was that (apart from *Free Area*) these behaved exactly like I/O components, and allowed us to maintain the "mini-batch" metaphor in the logic processing within a transaction.

There is (was) also another breakdown of modularity in IMS: namely the layout of the PCB List. Originally, IMS and transaction code had to agree on the sequence of the PCBs in the list, and the only way code could reference a PCB was by its position, so every subroutine in a transaction had to "know" the same PCB List layout. Again this made sharing subroutines between transactions very difficult. To solve this, DFDM provided a "locate PCB" function as a basic service, allowing PCBs to be located using the DBD name. IBM has since recognized this problem in IMS, and now provides the AIBTDLI interface, which allows PCBs to be referenced by name (you will remember that FBP ports went through a similar evolution from numbers to names).

One last topic I want to touch on is DB2 - it is appropriate to discuss it in a chapter on designing online systems. The relational paradigm is very powerful, and is very compatible with FBP's concepts - in FBP it is very straightforward to specify an SQL request in a component and then have it turn the rows of the resulting table into a stream of IPs. We have seen in Chapter 11 how we can attach information about "nullness" to IPs - this is a natural match with the "null" concept of DB2. We can even use the DESCRIBE facility of DB2 to generate IP descriptors automatically.

While in many ways DB2 is a marvellous system, it also suffers from what I have called a breakdown of modularity. In hindsight, it would have been better if its interfaces had been designed to be used in black boxes - unfortunately its designers did not foresee the need to use DB2 inside asynchronously executing black boxes, but we have found ways to live with this omission, so overall the two systems work pretty well together!

DB2 differs from most computation-oriented programming languages in that any components which contain SQL statements have to be precompiled as well as compiled, resulting in a separate type of output called a Data Base Request Module (DBRM). DBRMs are combined or "bound" (much as components are link edited) into what is called a "plan", which is required at run-time. One of the really nice things about using DB2 in an FBP environment is that the whole network only needs to be rebound when a coroutine issuing SQL statements has to be recompiled (re-precompiled, actually). If you therefore write all your Static SQL components and compile

and bind them into a "plan" early in the development cycle, you can add logic and other functions incrementally without ever having to rebind your application plan.

The major problem we ran into was that the DB2 "cursor" (the "pointer" which programs running under DB2 use to step through a table) is not a variable, so it cannot be moved around or passed to subroutines. So you can't do a SELECT in one component, and the related UPDATE in another one. In FBP we can get around this problem by using a single component to do both actions, either by using two separate input ports or by using a single input port with two different types of request IP.

The other problem is that there is only one DBRM for a component per task, so only one cursor, so a single component doing cursor-type SELECTs cannot run asynchronously with itself in the same task, unless you give it multiple cursors (we did that experimentally - if the component needs a cursor, and finds the current one busy, it just grabs the next one). Alternatively, we could eventually develop means to automatically generate the code for this kind of component as required. Dynamic SQL does not suffer from this limitation, but it provides weaker security, so most installations prefer to use Static SQL.

Last, but probably not least, when preparing a component to run under DB2, you have to specify at link-edit time whether the program is to run under TSO or IMS. This means that a component cannot be "ported" from TSO to IMS or vice versa in load module form unless it is relinked at its destination.

I'd like to close this chapter by addressing an argument which you may hear from time to time - namely that batch is dead, and that everything can now be done on-line and therefore synchronously. As you have been reading the foregoing pages, you may have been wondering what the relevance of data streams and components like Split and Collate is to today's interactive applications. For a while I also believed that FBP was less relevant to interactive systems than to batch, but as we built more and more on-line systems, we found that the benefits of reusability and configurability are just as relevant to on-line as they are to batch, if not more so. In fact, by removing many of the old distinctions between on-line and batch, not only do programmers move more easily between these different environments, but we have found that code can be shared by batch and on-line programs, allowing large parts of the logic to be tested in whatever mode the developer finds most convenient. We have even seen cases where data was validated in batch using the same edit routines which would eventually handle it in the on-line environment. Once you remove the rigid distinction between batch and on-line, you find that batch is just a way of managing the cost of certain overheads, just as it is in factory, and therefore it makes a lot of sense to have systems which combine both batch and on-line. If you take into account distributed and client-server systems, you will see that there are significant advantages to having a single paradigm which provides a consistent view across all these different environments.

# Chap. XX: Synchronization and Checkpoints

*[This chapter has been changed a bit, to reflect changes in the way automatic ports are handled in JavaFBP and C#FBP.]*

We will start off this chapter by talking about how to synchronize events in an FBP application. After having stressed the advantages of asynchronism so heavily, it may seem strange to have to talk about strategies to control it, but there are times when you simply have to control the timing of events precisely, so it must be possible to do this. It is just that we don't believe in forcing synchronization where it isn't required. We have already seen various kinds of synchronization, such as loop-type networks and composites, so let us look at synchronization a bit more generally.

Typically we synchronize something in a process to an event elsewhere in the processing logic, or to an event in the outside world. Only certain events in certain processes need this treatment - it would be against the philosophy of FBP to attempt to synchronize a whole network. That belongs to the old thinking - in fact, writers on distributed programming often show a certain nervousness about not knowing exactly when things will occur. It used to be considered necessary to synchronize commits on two different systems, but we are beginning to realize that it may not be possible to scale up this philosophy across an entire network in an enterprise. In fact, IBM has recently announced a set of Messaging and Queuing products [MQSeries] which provide asynchronous bridging between all sorts of different hardware and software. A basic assumption of this software is that applications cannot, and should not, try to enforce events to be synchronized across multiple systems.

The most basic kind of synchronization is synchronizing a point in the logic of a process to a point event in time. In IBM's MVS, this is implemented using POST/WAIT logic. This involves a thing called an "event", which may be in one of three states:

- inactive

- being waited on
- completed

Only one process can wait on an event at a time, but any process that knows about it may "post" it, thereby changing it to completed. If a process issues a "wait" when the event is inactive, it is suspended on that event, and the event is marked appropriately; if the event has already completed, the process that needed to know about the event just carries on executing. MVS uses this concept for its "basic" I/O: a process starts an asynchronous read or write channel program going and then continues executing. When, some time later, it needs to know whether the request has completed, it issues a "wait", and either is suspended or resumes execution, depending on whether the asynchronous I/O routine has posted the event complete or not. Thereafter the event would show complete until it is set back to one of the other states, presumably by the requesting process. All dialects of FBP except THREADS [this was written before JavaFBP and C#FBP - they do not have this facility either] have implemented an event-type wait service to suspend a single process. A nice feature of the FBP environment from a performance point of view is that one or more processes can be suspended on events without suspending the whole application. The application as a whole is only suspended if no process can proceed and at least one process is waiting on an event (if there are no processes waiting on events, you've got a deadlock). We found that applications with many I/O processes often ran faster than if they were coded conventionally using buffered I/O because, in control flow [non-FBP] coding, only one I/O is logically being executed at a time, so if it suspends, the whole application hangs.

Instead of a point event, we might instead have to synchronize an application to a time of day clock, e.g. "run this job at 5:00 p.m." All you need is a process which sends out an IP at 5:00 p.m., every day (or you could arrange for it to send out IPs every hour on the hour, or every five minutes, or every 20 seconds). Such a process can act as a clock, just as the clock in a computer sends out pulses on a regular schedule. These IPs can then be used to start or delay other processes.

Similarly to the "wait on event" service available to its components, DFDM also had the ability to suspend a process for a specified amount of time or until a particular time of day. This used the facility provided by MVS to post an event at a particular time. In the case of DFDM, this function was only provided as an off the shelf component: it could either delay incoming IPs by a certain amount of time, or generate a stream of IPs at given intervals. Where multiple time intervals were required, you could have as many of these processes as you liked in a network (DFDM kept track of which one was due to go off next).

Another kind of synchronization referred to already is the need to delay something until a process has completed. In DFDM, any unused output port would automatically present end of data to its downstream process when it closed down. In THREADS the same thing applies, or you can use automatic ports if you want to automatically generate a signal IP when a process deactivates. [In

JavaFBP and C#FBP, only the latter technique is available, with the closedown signal being generated at termination, not deactivation time.] Imagine that you want to delay a process until *two* others have completed. You have already run into Concatenate - this provides a simple way of doing this, as in the following diagram.

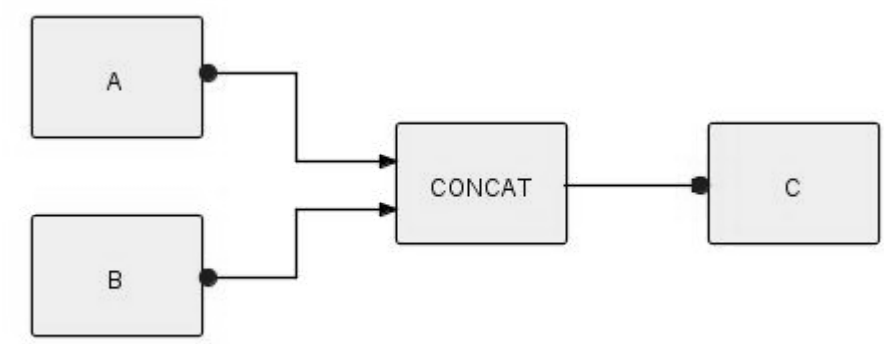



Figure 20.1

In this figure, processes *A* and *B* close their automatic ports when they terminate. *CONCAT* will not close down until both *A* and *B* have terminated, so the end of data output by *CONCAT* (when *it* closes down) can be used to delay process C.

Client/server relationships are a good way to solve a particular synchronization problem: suppose you have a stream of transactions that all access the same data base. If you allowed every transaction to run in parallel, and your programs were managing their own data, you would have to provide some kind of enqueue/dequeue mechanism to ensure that different threads were blocked from executing at the same time when this might cause problems. A simpler technique is to make one process a server and only allow that process to access the data base. This is in fact a common type of encapsulation because it allows the server to control what it will accept and when. We described this kind of approach in Chapter 15. The disadvantage of this arrangement, of course, is that you are serializing the data handling part of the transactions, so this may become a bottle-neck, but this is a trade-off that should be the decision of the application designer. It's even better if only a small proportion of the transactions need the server's services, or you have several data bases, each with its own server (like the different stations in a cafeteria, or the tellers in a bank). One other possibility is to use some kind of batch approach, especially in cases where the hit-rate (transactions per data record) is fairly high. After all, batching is just a technique for lowering the per-item cost at the cost of increased start-up and close-down costs. Since a server "batches up" its incoming transactions, you may be able to preprocess them to improve performance.

Another kind of synchronization is built into the "dynamic subnet" mechanism of DFDM. We said before that a composite component monitors the processes within it. In particular, if the composite is substream-sensitive, it handles exactly one substream from every input port on each activation. These input ports are therefore also synchronized, so you can visualize the composite component advancing, one substream at a time, in parallel, across all of its input streams.

Of course, most often such substream-sensitive composite components only have one input port, in which case they process one substream per activation. We have described in some detail how they work in the chapter on Composite Components (Chapter 7). Now we get to use them on something that seems to give programmers a lot of trouble.

Interactive systems and systems which share data bases have to wrestle with the problem of checkpointing. In the old days, checkpointing just meant saving everything about the state of a program, and restart meant loading it all back in and resuming execution. Well, for one thing, a program had to come back to the same location and this might not be available. It became even harder as systems were distributed across multiple tasks or even multiple systems. In FBP, the states of the processes aren't in lock-step any more, so it becomes harder still! In general, as the environment becomes more complex, checkpointing needs more information to be provided by the programmer. However, we would very much like to be able to write a general checkpoint component which we could use across a wide range of applications, and we feel it should be possible with FBP. In the following paragraphs I will describe an approach which seems to fit the requirements. Rather than trying to create an enormously intelligent and complex module, our approach is to provide a series of points in time where as many processes as possible are quiesced, so that they do not require much data to be saved about them.

Consider three scenarios:

a) IMS MPPs take a checkpoint every time they go back to the input queue for another transaction. This "commits" the updates, and unlocks them so other users can access them. If the system crashes before the checkpoint, the updates have logically not been done, and IMS has to ensure that is logically true (even if it has happened physically).

b) a long-running batch application should checkpoint about every half an hour, so that the amount of the job that has to be rerun is never more than half an hour's worth (this applies both to programs updating data bases and to batch jobs using ordinary sequential files).

c) an IMS BMP should checkpoint much more frequently - perhaps as often as every few seconds - as online users of the same data may become hung waiting for the BMP to release data which they need.

The common idea in all these cases is that the system saves the logical state of the system, so that it can be restored if required. The information needed to restore a process to an earlier state is often called its *state data*. On the other hand, the less data we can get away with saving, the less

time checkpoint will take, and the faster any restart can occur if it is needed.

Since checkpointing needs a stable base with as little going on as possible, we will have to quiesce as many of the processes in our application as possible, and have as few IPs in flight between processes as possible. The more we can do this, the less state data we have to save. Here's an analogy: a number of people are swimming in a pool, and a member of the staff decides it's time to put chlorine into it. Since this chemical would be highly irritating to the swimmers, the first thing to do is to get them all out of the water. So the staff member blows the whistle - s/he now has to wait until everyone is out of the water, which might take a little while as everyone has to finish what they are doing. She now puts the chlorine in, waits some amount of time and then blows the whistle again to indicate that it is safe to go back in.

Let's reuse a diagram from Chapter 7:

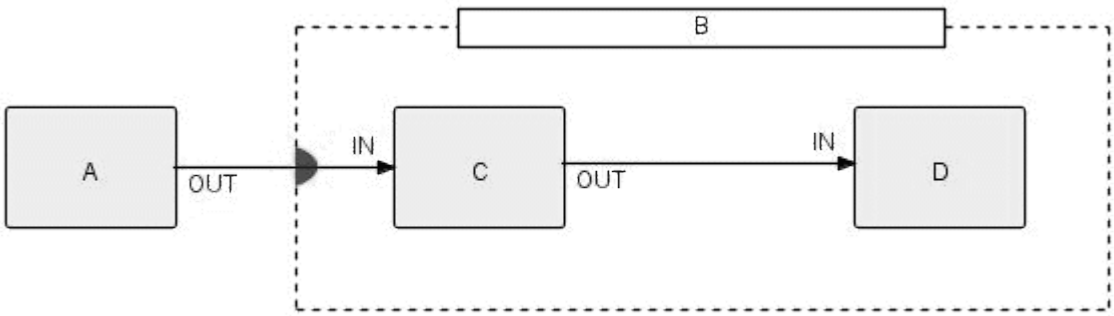



Figure 20.2

This shows a substream-sensitive composite *B*, containing two processes *C* and *D*. You will remember that, provided the data coming from *A* is grouped into substreams using bracket IPs, the inside of B will behave like a little batch job, starting up and closing down for every incoming substream. The composite deactivates each time its inside processes close down, and it restarts them when the next IP arrives from outside. During the times when *C* and *D* have closed down, there will be no IPs in flight, and *C* and *D* will not even have any internal storage allocated. The composite itself will be inactive. This then provides a rather neat mechanism for "getting everyone out of the water", because, remember, processes cannot be closed down until they themselves decide to allow it. Not only that, but we can ensure that the next substream isn't admitted until the chlorine has had time to dissolve! Just provide the composite with an automatic port, which will prevent it from inputting the next substream until a signal arrives. The diagram might now look something like this:

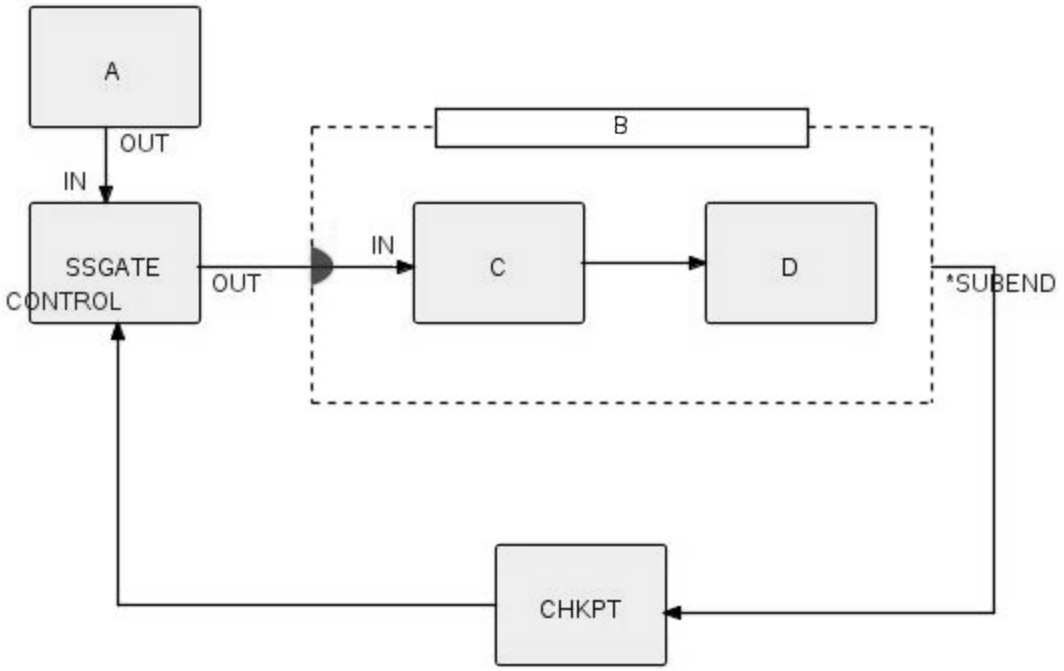



Figure 20.3

I am now going to suggest how we might modify this diagram, using a facility that has not yet been implemented, but should be very straight-forward. Let us modify the diagram to send a signal automatically every time all the components within B have terminated, using an "automatic" port called *SUBEND. When this happens, the subnet will send a null packet to the *SUBEND port, which signals *CHKPT* to run. *CHKPT* then sends the signal onwards to control a component called *SSGATE*, which lets the composite know it can accept another substream. *SSGATE* is a very simple component, and simply releases one substream at a time, each time it receives a signal at its CONTROL port.

There is another idea which is suggested by the swimming-pool analogy: a swimmer who will not get out of the water will hold up the whole process! Remember the term "periodicity", referring to whether a component is a "looper" or a "non-looper" - non-loopers are quiesced between every invocation, so the more often a component gives up control, the more flexible it will be from the point of view of fitting into the checkpoint process.

Since we are using the fact that *B*'s regular (non-automatic) input port is substream-sensitive, we now have to get delimiters into its input stream to make this whole thing work. It may seem

strange to use external markers to control what is going on inside the swimming pool, but this is really only a technique for dividing up the incoming stream into well-defined groups - and we want *all* the processes inside the composite to be able to close down. So what we do is insert delimiters into the incoming stream of IPs at the points where we want checkpoints to occur.

Now, there are two main criteria for when to take checkpoints: amount of I/O and time. Since in IMS a checkpoint will unlock changed records, we want to take checkpoints more frequently if there has been more update activity. Conversely, if the activity is low, we want to take checkpoints occasionally anyway to make sure that other programs are not hung for too long waiting for records to be unlocked. How can we drive checkpoints on both these criteria? Well, a close approximation to the amount of I/O is to count transactions, and do a checkpoint after every 'n' transactions, where 'n' is specifiable from outside. In addition, we want to trigger a checkpoint if 't' seconds or minutes have elapsed without a checkpoint.

Let's do the transaction counting first: we can just have a Count process which inserts a close bracket/open bracket pair every time the count of input IPs reaches a number 'n' (obtained from the option port). This Count process also has to send out an open bracket at the beginning and a close bracket at the end. Schematically:

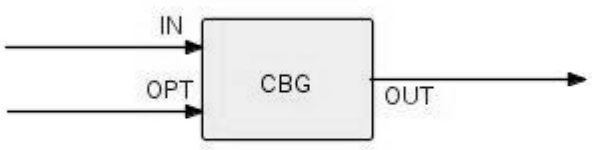



Figure 20.4

We'll call this component *CBG* for Count-Based Grouping. *OPT* can specify 'n'; *OUT* passes on the incoming IPs divided into substreams. So if the input to *CBG* is

```
a b c d e f g h i j k l ...
```

Figure 20.5

and 'n' is set to 5, the output looks like this:

```
(a b c d e) (f g h i j) (k l ...
```

Figure 20.6

This also works if the input consists of substreams, rather than just transactions. In general, the count should apply to the highest level substreams (we have seen before that we can treat

individual IPs as trivial substreams) - if we were to interrupt a substream to take a checkpoint, we would have a much harder time restarting from where we left off.

But now suppose that, during the "quiet" times, we decide that we also want to insert a bracket pair if 't' seconds or minutes have elapsed without a checkpoint. Let's take a "clock" process referred to at the beginning of the chapter, which generates an IP on every clock tick (specifying the interval via an options port), and merge its output with the input of *CBG*, as follows:

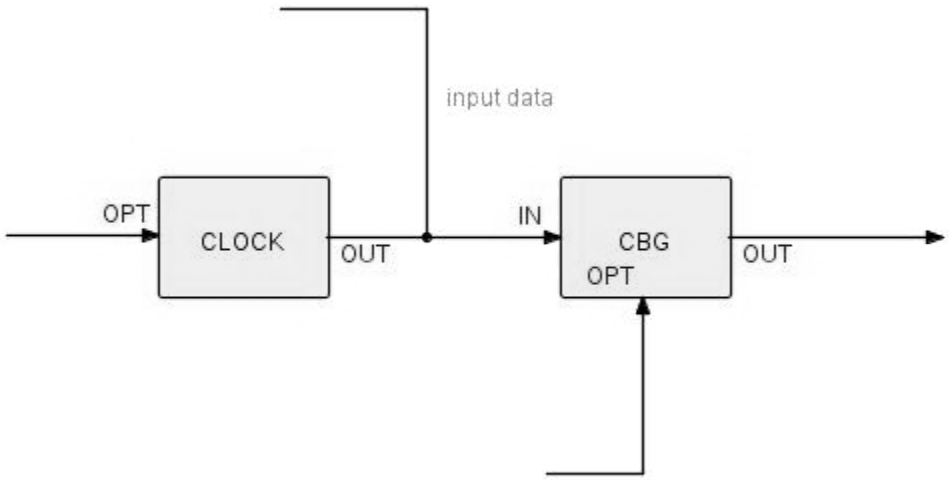



Figure 20.7

where *CLOCK* generates "clock tick" IPs at regular intervals, which are then merged with the original input IPs on a first come, first served basis. If the original data stream consisted of substreams, we would need a more sophisticated merge process.

The input stream to *CBG* now looks something like this:

```
a b c t0 d e f g h i j t1 t2 k l t3 m t4 n o p q r t5 s ...
where tn represents a clock tick
```

Figure 20.8

Now, the present FBP implementations do not *guarantee* that these clock ticks will ever get into this data stream unless there are simply no data IPs coming in. This is because we have always concentrated on making sure *that* all data is processed, but not *when*. And in fact this is probably adequate in this case, since we only care about the clock ticks when the frequency of incoming data IPs is low. To absolutely guarantee that the data IPs are inserted "in the right place", we

would need to implement something called "fair scheduling". I will not describe it here as it is well covered in the related literature.

Clearly when there are fewer data IPs between a pair of clock ticks, e.g. between *t1* and *t2*, there is less activity; when there are more IPs, there is more activity. So a simple algorithm might be to drive a checkpoint (insert back-to-back brackets) on every incoming clock tick, and also after 'n' IPs following the last clock tick IP. We might want to fancy this up a bit, by preventing checkpoints if a previous one occurred within some minimum interval, but the simple algorithm should do fine for most purposes. A lot of applications use a time interval only, especially in batch applications, where the problem is to reduce the cost of reruns, rather than releasing locked records.

Having identified our "bracket insertion" subnet (Figure 20.7), we can now insert it between *A* and *SSGATE* in Figure 20.2, as follows:

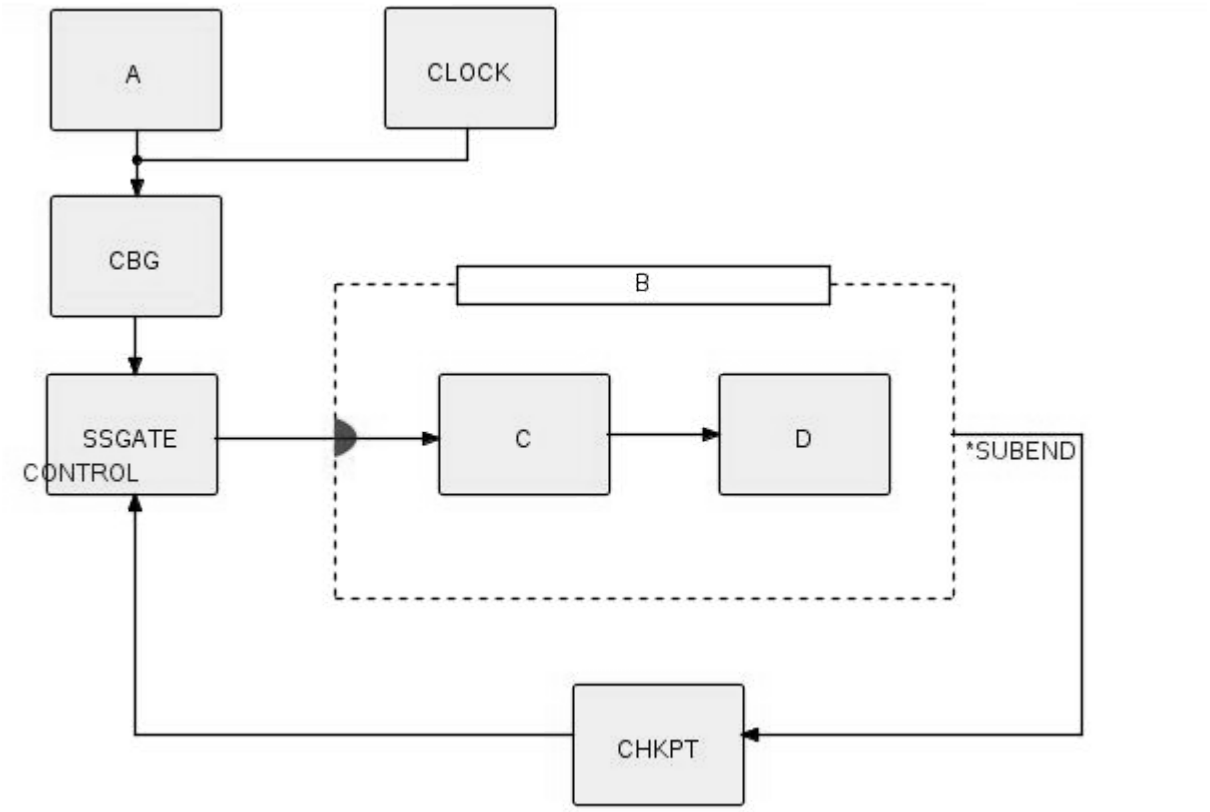



Figure 20.9

We have talked about *when* to take checkpoints - we now need to discuss *what* should be saved when we take a checkpoint. Not only does state data have to be saved in case a restart is needed, but in some systems when you *take* the checkpoint you lose your place - all your fingers get pulled out of the telephone book - so you have to be able to reposition them. So it makes sense to have as few processes active as possible at checkpoint time. In the example shown in Figure 20.3, process *A* is active at checkpoint time - since it has no input connection, it won't terminate until it has generated all its output IPs. Neither *CLOCK*, *CBG* nor *CHKPT* have any internal state information which needs to be saved if a crash occurs. So this means that *A* is the only one which needs to be restartable, and there must be a way to make sure that it only generates IPs which have not been processed completely. If *A* saves state data on a data base or file, the checkpoint mechanism itself will ensure that the state data gets saved when it should, and rolled back when it should. The exception to this is that data IPs which caused errors should probably not get reexecuted, so you may want to store information about them in a non-checkpointable store.

Apart from such oddball cases, we can generalize across the different environments and say that *A* and processes like it should save their state data on a checkpointable backing store, be able to be restarted using it, and that this whole process should be as automatic as possible. Let's say that the state data has some recognizable empty state - then the state information on the backing store should start off in that empty state, it should be updated for each incoming IP or substream, and the program as a whole should reset the state data to empty when it finishes. This way *A* can determine if a restart is required and, if so, at what point. Since it is a good idea to separate logic from I/O, we can split *A* as follows:

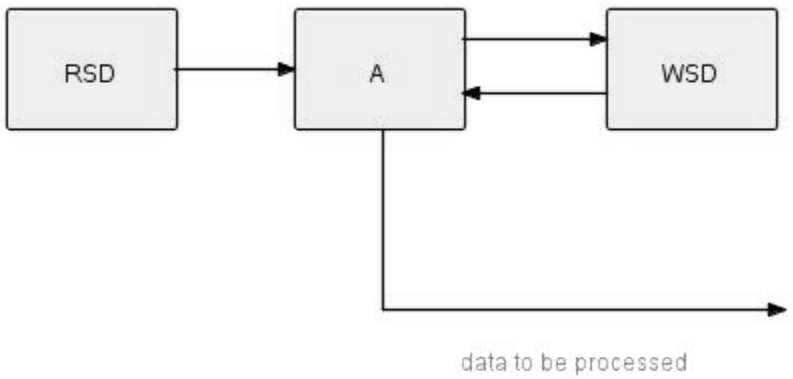



Figure 20.10

In this diagram, *A* only needs its state data at start time, so we can let *RSD* (Read State Data) start at beginning of program, and send the data to *A*. Every time *A* needs to store its state data, it

sends it to *WSD* (Write State Data). *A* very often needs to be notified that *WSD* has stashed it away safely, so we provide a return path for this information.

You may have noticed that *A* doesn't really need to save its state data until checkpoint time, but it doesn't know when *CBG* is going to decide a checkpoint is needed - this suggests that we might want to find a way to combine *CBG* or *CHKPT* with *A*'s writing to backing store. The other thing we might do is have a general repository of state data, and let *A* request it, say, by providing its process name as a key. A solution which combines all these ideas is to use a process like the List Manager (described in the next chapter) to hold up-to-date state data in high speed memory, and then expand the function of *CHKPT* so it writes out this information to disk before requesting the checkpoint. We therefore replace *WSD* by a State Data Repository (call it *SDR* ). *CHKPT* has to request the state data from *SDR*, so there will be connections in both directions between these two processes. The final diagram might therefore look something like the following figure:

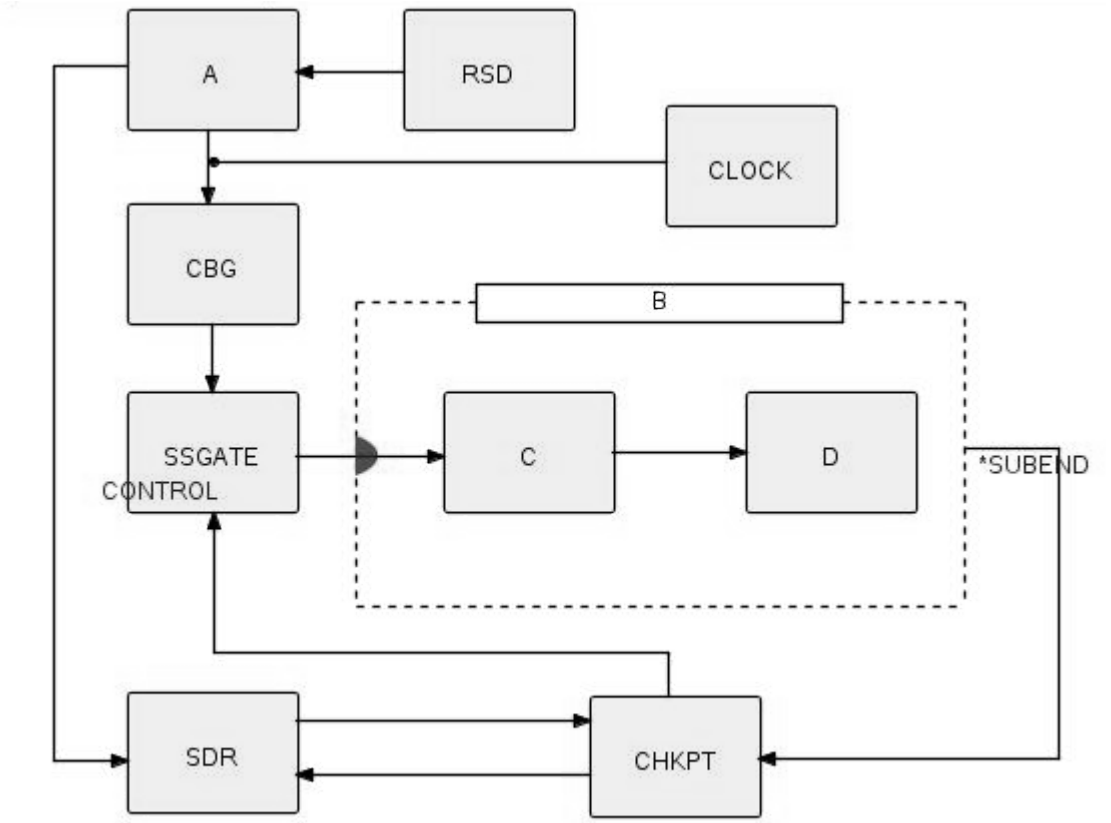



Figure 20.11

where there are two connections between *SDR* and *CHKPT*, one in each direction. The connection from *CHKPT* to *SDR* is used to allow *CHKPT* to request its stored data; the reverse connection is for *SDR* to send the requested data.

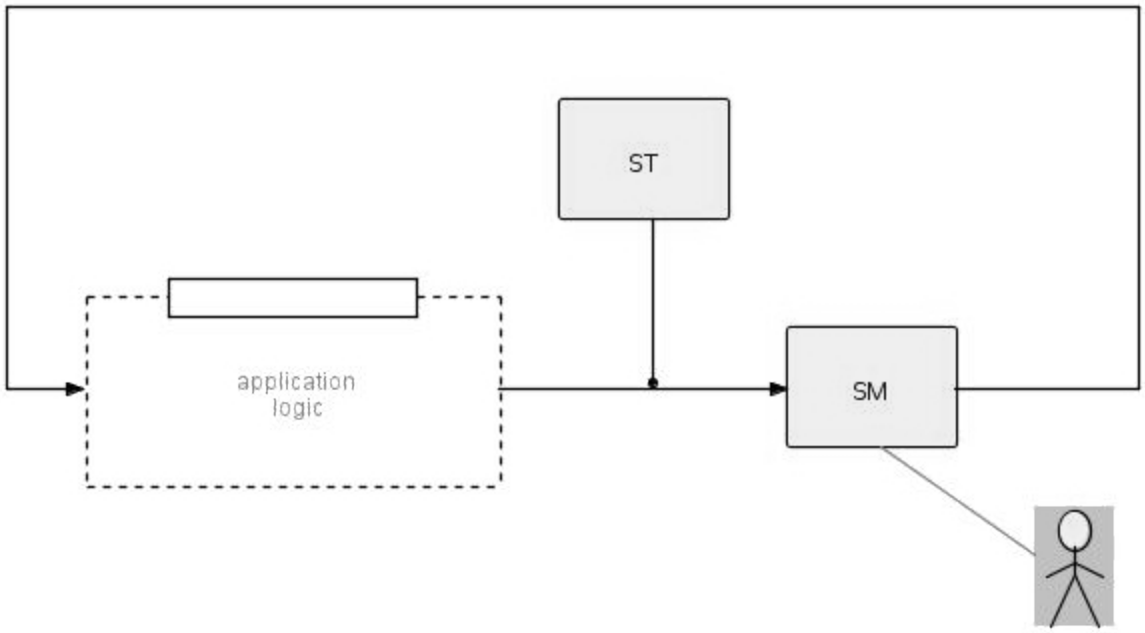
This is just a sketch, and even at that it's starting to look a bit complicated, but most of the components can be off the shelf, so they won't have to programmed from scratch for every application. At this point, I'm sure you can all come up with better solutions which draw on your own expertise - the point is to design generalized utility components which encapsulate expertise, but which still are easy for other, less expert programmers to use. When you consider the potential cost of reruns to your shop, I'm sure you can see that some standard, easy to use, checkpointing approaches and components will be well worth the effort that goes into developing them and I have tried to show that FBP's powerful modularization capabilities will make that job much easier.

# Chap. XXI: General Framework for Interactive Applications

*[This chapter talks mainly about our experience, pre-1994, using the FBP implementation called DFDM and IBM's ISPF. More recently, we built an e-business application using the Java implementation of FBP - now called JavaFBP. This was also essentially a loop-shaped network, but running across multiple servers (other chapters in my book talk about the ease of distributing FBP applications across physical networks). Communication between the user and the servers was handled by IBM's MQSeries transporting XML messages. The application also needed to communicate with multiple "back-ends", which it did using either MQSeries or CORBA.]*

In this chapter I am going to describe a general framework for interactive applications, showing a general structure and some component types which could help in the design of such applications.

We will start by reproducing Figure 19.3, which shows an IP substream travelling from application logic to a screen manager process and back again, and showing how it can be fleshed out to produce a very general design for interactive applications. You will remember the following diagram:




```
where SM  is a Screen Manager
      ST  starts the network as a whole
      the dotted line represents logic processes
Figure 21.1
```

You will remember that in IMS we had to split the process marked SM into SI and SO (Screen Input and Screen Output respectively), and change the function of the "return connection". However, for now, we will work with the above diagram, bearing in mind that it is very easy to convert it into one which will run in the IMS/DC environment.

In what follows I will describe a system we built using a single generalized Screen Manager component (which I will refer to as ISM1 - ISPF Screen Manager 1) which used IBM's ISPF both to write to and read from a terminal, but the concepts are extremely general and can be applied to other screen management software.

Although some systems allow a screen to be generated without using a program, it is simpler to assume that every application starts by putting up a "What do you want to do?" type screen. So assume that ST causes SM to output a menu screen. SM will have to have a place where the user's answer can be stored, so we can assume that ST sends out a substream consisting of at least three IPs: open bracket, "request" IP, zero or more data IPs and close bracket. The brackets are

needed so we can have a variable number of data IPs in the substream. The request IP will have, among other data, the name of the screen to be displayed. This substream then arrives at ISM1, which puts up a menu; the user enters a choice; the substream goes through the processing logic (which may change the contents of the IPs, or even add or remove IPs from the substream); and eventually we get back to ISM1 which puts up a new menu.

Now let's look a bit more closely at ISM1. This component accepted an input substream, put any variable data into position on the screen using the data descriptors associated with the data, and waited for action on the part of the user. When that occurred, the modified data was placed back into the right places in the data IPs, and the substream was then sent on to the next process downstream. ISPF identifies fields on the screen by name, and ISM1 used the field names from the descriptor to determine where to put each variable field.

In addition to this substream, referred to as the "fixed substream", ISM1 also accepted an additional, optional substream, called the "repeating substream". The mental image supported was that the screen has a fixed part, normally describing one or a small number of individual entities, and an optional list. Thus we could show a person's family on the screen: his or her personal information, the spouse's information (a separate IP), these providing fairly complete information, and zero or more children, showing just name, age and gender, say. If the user wanted more information on a child, he or she could select the child, and get a full screen devoted to that child, which might have further lists, e.g. education. One of the really neat things about being able to use IPs in this way is that both the list of children and the full screen describing a single child can be driven by the same IP - we just decide how much information we are going to show from that IP. By the way, since each screen was built using two substreams, we bracketed them together so that ISM1 would think of them as a unit - so ISM1 was using a substream of substreams.

Because field names are not unique in the repeating part of the screen, we could not use ISPF field names to control this part of the display, so we used a run-time table describing which fields from each IP went where in the repeating section. This had some interesting capabilities - ISM1 allowed you to specify more than 1 line per repeating IP, and the developer could also specify whether a "select" column (simulating the 1-byte column ISPF provides for selecting one or more items from a list) was required or not.

ISM1 also used the dynamic attributes which we talked about in Chapter 11 to keep track of which fields had been modified, and which were null. As I mentioned in that chapter, ISM1 also provided a special display for fields which had been "tagged" with error codes, and would let the user step through these errors using a reserved function key. ISM1 actually would not allow the user to go on to the next screen until all these "tags" had been removed one way or another! There has been lots of debate about whether this is a good idea or whether systems should be more forgiving! However, the important thing to remember is I am talking about the design of a single component - this in no way affects or is affected by the basic architecture of FBP.

So far, ISM1's abilities might seem about what you would need if you were "black boxing" a display function. However, it also provided another capability, which dramatically simplified the logic in the other components of the application: we have discussed this in Chapter 11 under the title of "representations". As I said in that chapter, representations mainly come into play when you need to present data to humans, or port it across systems.

In a prototype of an interactive application using straight ISPF I found three PL/I fields had to be defined for every numeric field on the screen:

- the field in a computational format
- a zoned decimal field (e.g. 000001234)
- a character field for input in case the user wanted to modify the field

When we converted this prototype to use ISM1, the number of fields we had to declare in the HLL portion of the application dropped by 2/3! We also discovered a number of additional bonuses:

- you could send an IP with an attached descriptor to ISM1 and it would automatically be displayed in the desired format
- the user could enter the data in free-form, but you could be sure that it wouldn't get into the system unless it was a valid representation
- you could implement a standard input convention for your whole shop - e.g. require that the field be clear apart from the incoming data (some screen managers allow you to leave junk at the end of a field following the data just entered)
- you could send an IP to ISM1 for interactive handling, or you could send it to a file writer and you didn't need to make any changes to your data IPs. The effect of this was enormously improved testing and regression testing, because you could test a lot of your logic in batch.

On one project in IBM Canada, this last technique was used very effectively by my colleague, Philip Ewing. Later in this chapter I will share with you what he has written about that project.

We have now sketched out a screen management component ISM1, which accepts one or two substreams as input, and outputs them again after the user has responded. If you are working in the IMS/DC environment, it wouldn't be all that hard to split these functions and link them together using persistent storage.

Now let's look at the first figure in this chapter. We need to fill in the logic between SM and the application logic. To do this, the first step is to interpret the user's action. Restricting ourselves for simplicity to ISPF and 3270-type terminals, the user may decide to:

- modify any data field, including Select fields as a special case

- enter a command in the command area
- hit a function key
- hit a Program Attention key (this will lose modified data)
- hit Attention
- position the screen cursor to a particular field

These will of course often be combined, e.g. putting an M [for "maximum"] in the command line and hitting PF8 ["down"] causes a jump to the bottom of the data in ISPF.

All these actions have to be encoded so that downstream processes can decide what is the appropriate response. If we add in more modern devices and interfaces, obviously there are still more variations, e.g. monitoring key-strokes and mouse movements in real time, but it seems that we will still have the cycle (or maybe many concurrent ones) of display - user action - interpret user response - program action - display.

In the ISPF world, and also IMS/DC, function keys are usually treated as commands, so one of the standard outputs of our Screen Manager will be a "command". These may be the very frequent ones like UP, DOWN, END and HELP, which are almost universal, or more application-specific ones. It turns out that these commands are convenient bases for the decision about what to do next. Always remember that each of the components described here can be used independently of any other. Now, in Chapter 7 we described DFDM's dynamic subnets - subnets which were linked as separate load modules and were loaded in dynamically and given control by a special component called the Subnet Manager. This will provide a convenient way of subdividing and managing our application. The Subnet Manager is driven by IPs containing dynamic subnet names, so we need a component which will take the output of the Screen Manager and generate the subnet names for the Subnet Manager. Let's call this the User Response Analyzer (URA).

The URA component's job is to look up in a table patterns consisting of screen + action, screen only or action only, and decide what to do about them. As we said, since it sits upstream of the Subnet Manager, its main job is to select subnet names to be sent to the Subnet Manager, but you might decide to have it bypass the Subnet Manager, and send its input IPs directly to the Screen Manager. In this case, you could have it decide screen names. You could also have it do both.

You will notice that we haven't said where this table should be held: it could be compiled into a load module, stored as a flat file, or held in a data base. Perhaps a file would be appropriate during development, and a load module in production. You will perhaps notice our predilection for tables - this is one of the most important ways of achieving portable code (remember Bucky Pope's class codes, alluded to in an earlier chapter).

The URA table might therefore look something like this:

```
  Old Screen          User Action      Subnet           New Screen
  ================================================================
  A                    CHOICE1         SUBNET1          B
  B                    END                              A
  A                    HELP                             HELP_FOR_A
  HELP_FOR_A           END                              A
  .
  .
  .
  .
```

Figure 21.2

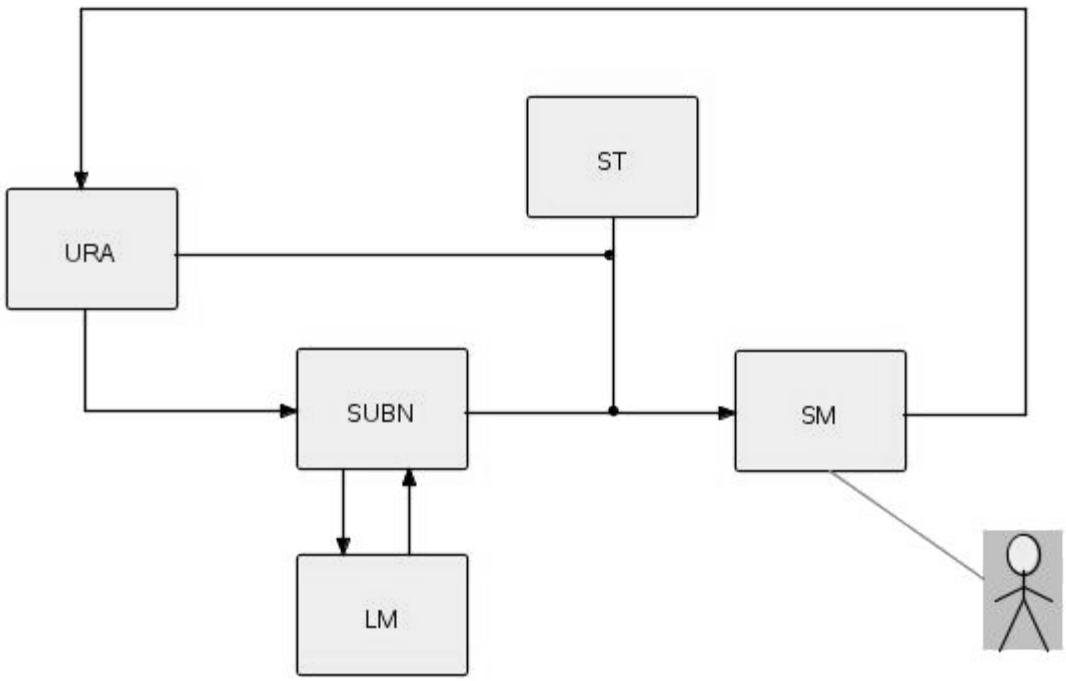
Obviously this table is very easy to modify - in fact, if you add a comment capability (an asterisk in col. 1 means ignore this line), it really becomes self-explanatory.

The last component I am going to describe is the List Manager, another general component. Its fundamental metaphor was sets of lists which persisted in storage, organized by "levels" - thus employees might be on one level, their children, departments worked in and courses taken might be three different lists at the next level. It could accept commands to do various things with these lists and levels, such as "create a new level", "insert a list at the current level", "jump to the next lower level", "pop up one level", "output a list (non-destructively)", "delete a level", and so on. Although (because?) this component was very powerful, it took the most work to manage its input and output. It was very interesting for another reason also - the List Manager perhaps most closely resembled an OO "object", in that it had an internal state, being constantly modified by incoming commands (messages) with or without accompanying data. Its structure seemed to match our perception of what was going on in the prototypical interactive application - i.e. the user would display an employee, then ask to go down one level to find his or her children, pop back to the previous level, and so on. Because it was a single looper process, we could just manage these lists by working with IP pointers - we didn't have to pay the overhead of chaining or unchaining IPs. Also, it provided a focal point, in case we needed to store really big lists, where lists could overflow to disk. We also expected that, when we implemented this design on IMS, it would be very easy to dump all our lists to disk at the end of a transaction, and retrieve them when they were needed again.

In hindsight, the problems we ran into with the List Manager were probably to be expected, but they came as somewhat of a surprise to us! I believe we were still thinking of interactive applications as sequential, so the command-driven, single store made sense. However, it was so convenient to stash things away in the List Manager's storage that we had more and more processes sticking stuff in there and taking it out. The more complex our networks became, the harder it became to control the exact sequence in which the commands arrived at the List Manager. What we had done, of course, was to implement a somewhat more complex array of pigeon-holes, and the non-destructive read-out which seemed so attractive at first caused the

same problems FBP was trying to avoid! Strange sequencing problems started to show up - lists would get attached to the wrong level, lists would show up on two different levels, and so on. In turn, the sequence of the command IPs had to be controlled more tightly, introducing still more complexity. In hindsight, I believe we would have been better off using tree structures flowing between processes, rather than complex data structures within a process. Alternatively, a List Manager should only be fed by a single process, and this is the way I have shown it in the next diagram. Lastly, I believe that the underlying metaphor may not have been quite correct. For instance, suppose the user is stepping through an employee's employment history and decides to start looking at her courses. Should this be made another level? Or are all these lists at the same level? A better metaphor might have been to be able to pop up new windows as new lists are requested. It's also useful to be able to open multiple windows on the same list (but you have to be careful about updates!).

We can now show the final picture. Remember that this is only a skeleton - you can add additional processes to the diagram, and extend it in other ways also. And remember also that the List Manager, although shown in the diagram, is not the only way to manage storage of data.




```
where SM  is a Screen Manager
      ST  starts the network as a whole
```

```
      URA is the User Response Analyzer
      SUBN is the Subnet Manager
      LM  is the List Manager
Figure 21.3
```

What we have described here is the structure we called the DOEM, pronounced "dome", (DFDM On-line Environment Manager), still fondly remembered by some of the people who worked on it! It was at the same time a skeleton structure, a set of components and an approach to designing interactive applications. This is reuse at a higher level than the level we have been mostly talking about up until now, and from that point of view a precursor of the way interactive systems will be built in the future. While the DOEM was a very powerful set of concepts, some of its components were more satisfactory than others in terms of their encapsulation of useful function and the simplicity of the underlying mental image. In some ways, the DOEM fell into the pitfall I have warned about elsewhere in the book - we tried to make it very general, based on our ideas of what a DOEM should provide, without frequent consultations with real users. Or we may have been talking to the "wrong" users. We never did build it for the IMS/DC platform, although we basically knew how to go about it. As it turned out, we didn't need that implementation anyway, for the reasons I am about to relate. This story is salutary, so I am going to tell it in some detail, as a cautionary tale for those embarking on developing reusable code.

Most of the time we were working on the DOEM, we were supporting two projects - let's call them A and B. The intent was to provide team A with an IMS/DC version of the DOEM, and team B with a CMS version. This seemed reasonable because a number of the components could be shared, and, although the CMS version was certainly simpler (single Screen Manager module, etc.), we understood pretty much how to build the DOEM on IMS/DC. However, the two teams' approaches to working with us were very different. The A team tended to be demanding and critical, frequently asking for specialized modifications of components or new facilities just for their own application, while B was more willing to work with us and to stay within the facilities that were already available or in plan. Both projects had the potential to be very important products, for different reasons, and both groups felt that they were getting benefit from DFDM, but both of them required quite a bit of our time, both to provide general support and to code and test the reusable components being supplied for the two environments.

Our development team was a small one and, under the circumstances, was getting stretched very thin trying to support both projects! Finally, management decided that we could only support one of these projects, and, after much soul-searching, they picked B. We started working intensively with B to make sure that the CMS DOEM worked well with their product, and as the two started to come together, we all realized that this had been a good decision. This product is now a successful product in its own right in the Latin American market.

The A team were told that they could continue to use DFDM, but not the DOEM, and that we could no longer afford to give them special support. We really expected them to decide to drop

the use of DFDM altogether, and while this would have been disappointing, we felt that this would be a pragmatic decision on their part. However, at this point, a very strange thing happened: faced with the possibility of losing the use of this productivity tool and having to redesign and rewrite a lot of their code, the A team turned right around and started to solve their own problems using basic DFDM! Instead of having us build complex generalized components, they found simpler ways of doing what they needed, and the result was a less complex, more maintainable system. Their product is also now a success, and is saving the company considerable amounts of money.

Actually, an additional project using the DOEM appeared suddenly on the scene one day, rather to our surprise! It seemed that a bright young contractor had been given the job of building a small interactive system, and had built it in a matter of a few weeks, using the DOEM, without telling any of the DOEM development team! We were very conscious that our documentation was nowhere near adequate at that time, but he said he had no trouble understanding and using it! Of course, he is very bright, but how often does something like that happen using conventional programming tools?

Since fairy tales usually have morals, let me propose the following: "Sometimes it is better to redesign a squeaky wheel than just put more oil on it".

The Screen Manager, ISM1 (actually an earlier version of it) was also used by itself, before we even thought of the DOEM, on an earlier project within IBM Canada, and this project became very successful, not least because Philip Ewing was excited by the concept of FBP (he still is!), and was discovering neat new uses for it all the time. As you may have gathered, ISM1 was a very powerful component, and all by itself considerably simplified the development of interactive applications. Its development predated the rest of the DOEM by several years, so we had used it for several small projects. Here is what Philip has written about our experience on this project (called BLSB).

> DFDM was selected for use on the BLSB project because of the significant productivity improvements that were anticipated. The development team was not disappointed. Significant savings were realized in the following ways:
>
> 1. We were able to prototype more easily, beginning with a simple screen display, and adding functions one by one until the user was satisfied. The full function prototype could be modified to add a new edit or data-base lookup in a matter of hours, without disrupting the existing code.
> 2. Testing was made simpler because we were able to unplug the online screens and feed in test SCREEN REQUEST ENTITIES [abbreviated to SREs - these correspond to the "request IPs" referred to above] from files, and save the returned SREs into separate files based on type of error. In this way all of the application function in the online system could be tested in batch.

3. Building on the experience gained in the function testing, the legacy data was converted to the new database format by feeding in the old data in SRE format (simulating re-keying all of the previous 3 years of data through the new system). The errors were saved in separate files based on the ERROR CODE that the application put into the SRE before returning it to the screen. Each file was known to contain only one type of error. In three iterations through this process we were able to convert and load 64,000 history records with only 12 records needing to be re-keyed manually. In addition to not having to write a separate conversion program, we were also assured that all of the data that was now in the database had passed all of the rigorous editing that had been built into the new application logic.
4. A great deal of effort in the design stage was saved because we could decompose functions to very granular levels before implementing. This meant that less thought needed to be put into the way different functions might affect each other, because different functions were now completely decoupled.
5. The "off the shelf" screen display function alone saved about 700 lines of application coding to handle ISPF panel displays. We did know ISPF before starting this project, but would not have needed to, since all of the ISPF specific code was in a DFDM-supplied function.

"Less than 24% of the functions needed to be coded by the project, the rest came off the shelf. Furthermore, of the ones that we did have to code, the most complex was about 100 lines of code."

A comment made to me recently about the BLSB project: "We allowed 3 weeks for testing, but it worked the first time."!

Another project which was very interesting was a system we built to do project resourcing, called PRORES, designed by A. Confalonieri, and built by myself, using a Screen Manager similar to ISM1 and a User Response Analyzer, running on CMS. This Screen Manager was also driven by IPs with descriptors, but generated and accepted 3270 data streams (extended data stream), rather than using ISPF. It used a WYSIWYG representation of the screen, and was the heart of the prototyping tool which I have mentioned several times elsewhere in this book.

The logic for PRORES was all written in REXX, and, considering that PRORES had to do a very large number of date calculations for each screen, its performance was surprisingly good. The basic idea was that, for each project that you were working on, you just entered a number of person-months and PRORES would generate all the dates and staffing requirements for the standard 5 phases of a project (Requirements, External Design, Internal Design, Development, Implementation), using formulae based on the standard "Volkswagen" shape or "snail curve" that

most projects follow. You could also specify that a project should be "flat", instead of standard. If you constrained the end date to be earlier or later, you would get a more humped or flattened staffing curve. If you didn't specify the start date, it would use the date of the day you ran it on. You could also request a graphical display of a department or division's projects, and it would use PGF, GDDM's Presentation Graphics Facility) to show all the projects in Stacked Bar format on a single chart across time. The Stacked Bar format meant that the project loadings were displayed cumulatively, so the top edge of the diagram showed the total staffing curve for the whole department or division. Management could then flatten and lower the overall curve by shifting projects around, stretching them out or compressing them, or moving projects between departments. Suppose you had two projects PROJ1 and PROJ2, both with the characteristic snail curve:

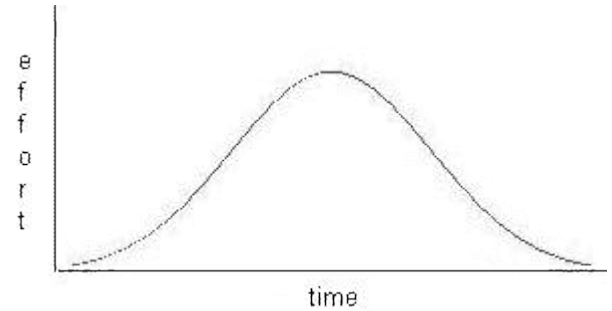



Figure 21.4

Now getting PGF to superimpose them gives the following kind of picture:

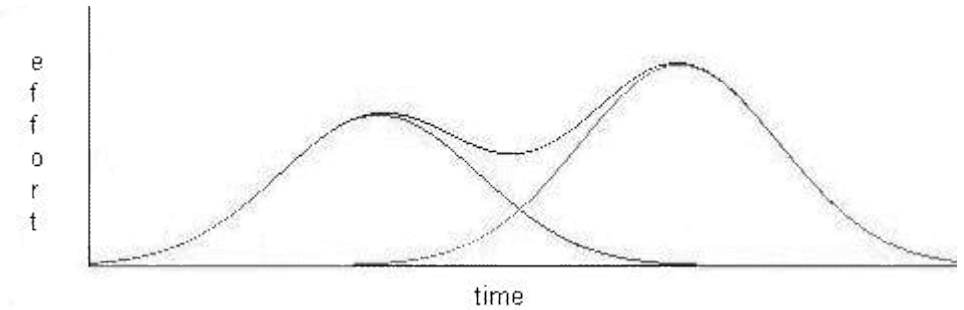



Figure 21.5

The outer "envelope" then shows the total cumulative loading for the two projects. With a relatively large number of projects, you can adjust things so that the top line is flat most of the time. "Flat projects" (projects which had a constant loading over their whole lifetime) could be used to handle things like vacations, education, overhead, etc. All dates were constrained to business days, and once they were all calculated, individual project dates could be modified as desired.

Technically, this project was interesting because of the languages and software involved. It was also a decision assist type application. You would get a screen just full of generated dates, and then, depending on which dates were changed, it would do intelligent things with them. This meant that it was very important for the Screen Manager to report on which fields had been modified. All the calculations were done in REXX. Date displays were handled by means of date input and output routines written in Assembler, driven by the Screen Manager using descriptors, so REXX only saw dates in canonical form (number of days from a reference date). If you pick your reference date correctly, then day of the week is just "date modulo 7", and you can make Sunday 0, Monday 1, and so forth up to 6 (Saturday). This system was used intensively over a short period to help reorganize a portion of our division.

I am going to mention briefly another use I made of the Screen Manager because, although it was not a complete project, I found it very suggestive of the shape of user workbenches to come. Most of the application development on IBM hosts is done today (1993) using ISPF/PDF on TSO or CMS EXECs. Within this kind of workbench, developers use quite a wide range of "languages": HLLs, Assembler, JCL, DBDs, PSBs, MFS, FBP networks (hopefully!), 4GLs, and of course documentation in his or her own national language. All of these are held in different data sets, and have different, although standard, processing applied to them. PDF follows the "action/object" paradigm: decide the action, then select the object. Having to choose the action first means that you always have to know what language the thing you are working on was written in. Also, in PDF you pick the same EDIT for everything, but then usually have to go to a completely different menu to process the text you have entered, and you always have to reenter the object's name, even if you were working on it a few seconds ago! Native CMS is a little different since it is command-oriented, but here you have to remember the command name to do the desired processing. Of course both lists support lists with optional "wildcards", but it is still hard to move a single object through a series of phases (like edit, compile, run test, etc.).

I figured that it would be nice if everything a developer was working on could be treated as an object of a particular type, with a unique name. The developer could just select the object she wanted to work on, and the system would know what language it was written in, and display an action bar showing what actions could be applied to it. So the interface would prompt you for a component name (with optional "wildcards"), or you could ask for all components of a given type, and you could just click on an entry in the action bar, without having to worry about choosing an inappropriate action for the object's type. Make the whole thing table-driven, and you have a very powerful, friendly system for application development - I know, because I built one for the CMS environment! All I had to do was select the object, and an appropriate action bar would come up, which would let me select from a list of CMS EXECs (e.g. EDIT, COMPILE and the most important one of all: DESCRIBE). If the object types are user-modifiable, you can be more specific, i.e. "Assembler source" could be split into "programs" and "macros", or you could have types like "screen", which will generate MFS or BMS, plus declares for the message

layouts. You could drive syntax-sensitive editors for different languages, or for objects of type "diagram", you could make the EDIT option call a picture editor. I also felt that each action should go as far as possible, i.e. if you decide to COMPILE an Assembler source program, what you really want to do is translate it from human format to machine format - there is no particular point in making ASSEMBLE one action and then LINKEDIT another. The COMPILE action could also automatically update control tables for use by tools like MAKE or symbolic debuggers.

The reason I said this prototype was suggestive is that, if you can build this kind of development environment using FBP, and we know that FBP also lends itself to building compilers and text processing software, this conjures up the very appealing image of a totally user-modifiable development environment, built out of communicating standard components. This wouldn't be just a set of tools - this would let developers continuously expand and improve the workbench itself!

Last, but not least, in this rather varied set of examples, there was a project, which might be called an Electronic Information Booth, to provide a visitor to our building with information such as how to find people, information about the building (layout, statistics, etc.), promotions during the month, a "trading post", and the cafeteria menu for the week. Everything was to be highly graphical and menu-driven. I prototyped this using the same screen manager I described above and a User Response Analyzer to implement the paths between the screens. As I mentioned above, this screen manager had a fairly complete graphical specification facility based on polygons, so it was well suited to developing a lot of pictures in a hurry! Since I already had the Screen Manager and User Response Analyzer, it was really just a matter of working with our very talented resident artist, Bob White, to develop the pictures. As it was around Thanksgiving, we decorated the Thanksgiving menu with a rather nice stylized corn-cob. During the Christmas season, we put the Christmas menu on its own screen with a little Christmas tree at the top! Later on, it was decided to implement this application using PCs, but our prototype certainly played a significant role in convincing management of the validity of the idea.

# Chap. XXII: Performance Considerations

I cannot stress too strongly that FBP is intended for building *real* systems, and indeed the various dialects I have been describing in this book have been successfully used for this purpose many times! To achieve this, we *have* to keep performance in mind - not only must a system do what it is supposed to do, but it must do it in a "reasonable" amount of time. While admitting that this is a subjective measure, I submit that, at a given point in time, we generally have an image in our heads of what constitutes a reasonable amount of time for a particular function. While there will almost certainly be disagreements, say between users and developers, we see the use of SLAs (Service Level Agreements) becoming more and more widespread, which force the parties involved to reach some sort of compromise between what one wants and the other will commit to. Response time is another area which is highly subjective, but studies have shown that there are observable differences in behaviour as response time changes, which match the users' subjective impressions. Once people get used to subsecond response time, anything worse is highly frustrating. If it gets really bad, their whole style of interacting with the machine has to change.

Now a system which has to deliver function to real life users has to be reliable, and also has to perform. A system may provide wonderful function, but if you cannot produce adequately performing systems, it will only be of academic interest. Suppose you decide that you are going to interpret English descriptions of function using a natural language parser - it may be a great research project, but, *given present-day (1993) technology*, you probably can't run a bank on it! Can you "run a bank" on an FBP implementation? Yes, we've been using it for a large chunk of the batch processing of a bank's bread and butter transactions for the past 20 years!

Now, not only must the infrastructure be fairly fast, but it must be *tunable*. Key to this, I believe, is being able to form a mental model of what is going on, and where a performance or logic problem may be occurring. If a developer or debugger cannot do this, the reaction may be to want to throw the baby out with the bathwater. Adequate performance is important even when a new system is in the evaluation stage. At this point, overall impressions can be very important.

The system has to have knobs and levers that let the developers tune the system, and the system has to behave in a linear manner. Linearity means that systems must "scale up". As you add function, not only should you not get sudden breaks in continuity - crashes, unexpected results - but ideally the curve should not even be exponential. In conventional programming environments typically complexity, and therefore development time, goes up exponentially as program size. Humans tend to have difficulty grasping the implications of exponential curves.

While it is possible that software could automatically balance two variables, I believe performance is a multiple variable problem. In any given application, a programmer has to juggle resources, algorithms and whole design approaches. Given this complexity, the software should be made as transparent as possible. In FBP, although there are many possible solutions to a problem, the developer can select a solution and then see the results, and can in fact iterate quickly through a large number of possible solutions.

In my experience, the single most important factor affecting performance is the design. In general, shaving a few microseconds off a subroutine will not make much difference, unless it is being performed billions of times. But a different approach to a design can often save seconds, minutes or even hours. Here is a simple example: about 20 years ago, a programmer on a project I was on needed to calculate the day of the week, given a date. He decided he would need a data base, running 3 years into the future and 3 years into the past. Of course the data base would have to be maintained by hand, so he planned for it to be updated once a year. Even without taking into account the manual labour and the difficulty if he was presented with a date outside the data base's range, I just figured that it might be a lot simpler to calculate it mathematically! Eventually this huge effort turned into a macro of about 12 statements - guaranteed until the year 2099! I am sure you are saying that's a trivial example and that we were all ignorant, but I am sure there are examples in your code where a small change of viewpoint can yield a big dividend in performance or maintainability, or both!

However, to be able to take advantage of these changes of viewpoint, you have to have a modular system. With FBP, as we saw elsewhere in this book, you can replace a sort by a table look-up, or a file by a connection; you can move function from one job step to another, or from one system to another, and so on. As the programmer I quoted earlier told me, "One of the things I like about AMPS is that there are so many more ways to do a job than with conventional programming". *There is never just one way to do a job*. And an FBP developer is continually making conscious trade-offs between different factors. For instance, she might decide to favour response time over throughput, so that might tilt the scales towards direct access instead of sorting. Or the old storage vs. CPU time debate. Or within a component, you can trade off state data against code: Boolean switches can either be held in a variable, or can be implemented by choosing between two code paths. It is important to know what options are available to you, and the system must allow you to choose the one you want.

The fundamental trade-off we make in FBP is that we have decided to spend a certain amount of

CPU time to get improved programmer productivity and program maintainability. Of course the industry has been doing this since HLLs and operating systems were invented, so this is nothing new, but you always have to decide how much you are willing to pay. But the amount also has to be controllable - a certain cost may be appropriate for one type of application, but wildly inappropriate for another.

In the case of FBP, the cost is closely related to the number of processes, which in turn is related to what is called "granularity". In general, the finer the granularity, the more CPU time gets used, so you can afford smaller "grains" in an infrequently used part of our application network. Conversely, 5,000,000 IPs going through a path of 12 processes will cost at least 120,000,000t units of CPU time, where 't' is the cost in CPU time of one API call (each process is going to cost at least 2t units per IP - for a receive and send). Depending on the speed of your machine and what 't' is, you may decide that 12 is too many. Reducing this number will probably entail some sacrifice in terms of reuse or modularity, as you will be combining smaller, reusable functions into larger, more ad hoc components.

Periodically, people will write an FBP network and a PL/I program which do the same thing and then compare their performance. This is really comparing chalk and cheese! The maintainability of the two programs is totally different, as a PL/I program adds far more maintenance load to your DP department than does an FBP network. Furthermore, if a bug in a generalized component is fixed, everyone benefits, while if a PL/I program bug is fixed, usually there is no lap-over to other applications.

Susan came to us with a program which she felt was slow compared with a "reasonable" execution time. I noticed that she was using the Assign component to set a 1-byte field in every IP, and there were a lot of IPs. She had been told to make her program very modular, but also she probably felt that Assign was a neat function, and it meant that much less PL/I that had to be written and maintained. I suggested that she add a single statement to each of a couple of her hand-coded PL/I components. There were other things we could do which together improved the running time significantly. Although we "improved" her program to the point where its performance was acceptable, we probably reduced its maintainability slightly. But this is another trade-off for which no general rules can be given - each situation has to be decided on its own merits. I mention this example mainly to stress that it's a complex decision whether to use a separate generalized component or add function to custom components, involving issues of maintainability, performance, predicted use, ROI and so on. There is a strong element of the aesthetic here - nobody *should* make the decision for you, just as nobody can tell you which paintings to like. Maybe someone *will*, but that's life!

The bottom line is that we have to give application designers as many choices as possible, and as much control as possible, and systems which won't or can't do this won't survive. Why has COBOL survived when so many more sophisticated products have fallen by the wayside? Because, even if you don't like it aesthetically, COBOL does provide this control, even if it takes

years to write a system and, once written, it's even harder to maintain. However, COBOL systems degenerate over time, as the developers lose knowledge of, and control over, what's going on under the covers; FBP systems don't.

Now let's talk about specific techniques for tuning the performance of FBP applications.

IN Chapter 20, we talked about the basic MVS synchronization technique of WAIT on ECB. We found that quite often when the same program was written in, say, PL/I and in DFDM, that the elapsed time of the DFDM run would be less than that of the PL/I run, but the CPU time would be somewhat greater. It becomes obvious why this should be so if you visualize an FBP network with 6 processes all doing I/O: provided the I/O is using different channels, drives, etc., it can all be happening concurrently, while the scheduler will always be "trying" to find CPU work to do to fill available CPU time. FBP programs tend to be CPU gluttons, but that is also the designer's choice - if he or she is concerned about this, I/O processes can be prevented from overlapping by synchronization signals. Usually, however, programmers want to reduce elapsed time, e.g. to fit into a processing "window" (time-slot). Today windows are getting shorter and shorter, so there is pressure to reduce elapsed time, even at the cost of more CPU time.

As we described above, using FBP connections in place of intermediate files definitely saves elapsed time, as all the records no longer have to be written out to disk and read back in again, but it will save CPU time as well, due to the reduction in the number of EXCPs (EXecute Channel Program). In MVS, although the EXCP count is maintained as a separate piece of statistical information, people tend to forget that EXCPs are quite expensive in terms of CPU as well.

Here are some statistics recorded during an evaluation of DFDM against an existing PL/I application, done by an IBM I/S site in the US:

```
I/O EXCP requests:            53.6% reduction
CPU Usage:                    16.7% increase
Elapsed execution time:       37.7% reduction
External DASD requirements:   84% reduction
```

Here is what they said about those figures:

> "We attribute the above reductions in resource usage to the improved design of the application (as a result of using structured analysis which is promoted in conjunction with DFDM). The slight increase in CPU usage is a small trade-off when you take into consideration the improved design of the application which should significantly lower future maintenance."

Let's say that you have tuned your I/O as well as possible, and that you are now trying to estimate how much CPU time your application will take. I have found that the main predictor for CPU

time is the number of API calls. On the IBM Model 168 (this was about a 2 MIPS machine), each API call took approximately 10 microseconds. Of course later machines are much faster, but later implementations of FBP are typically written in higher level languages, have more features and do more checking, so the time has probably stayed approximately constant. Just for comparison, I recently did some measurements on THREADS, running on a 33 MHz 80486DX, and the time per API call was approximately 50 microseconds (THREADS is written in C, so this figure could probably be improved by rewriting critical sections of the scheduler in Assembler).

Now, in most networks, there is usually a "back-bone": a path where most of the IPs travel. You could think of this back-bone as much like a critical path in a Critical Path Network. Let us suppose that this path consists of 12 processes, each sending and receiving: then, taking the figure of 10 microseconds, we get a cost of 240 microseconds per IP. If 5 million IPs travel this path, you get a total CPU time for API calls alone of 1,200 seconds, or 20 minutes. You may therefore decide that you want to consolidate some of the number of processes in the application "back-bone" into larger, less generalized ones - as we said above, this will reduce maintainability, but you may well decide that it is worth the price. Of course, you don't have to consolidate all the processes in your network - areas of the network which are visited less frequently, like error handling logic, may be left more granular.

It is common wisdom nowadays to concentrate on developing function first, and then worry about performance afterwards. This is quite valid as long as you don't squander performance up front by poor design. And here of course is where aesthetics comes in - I started my programming career on an IBM 650, which had 2000 10-digit words stored on a spinning magnetic drum, and I believe that it became almost an instinct to make programs as lean in terms of time and storage as possible. But, even in those far-off days, there was wide variation in the performance of code by different programmers. I believe the most successful ones never lose sight of the performance requirement, so that it is a constant undertone to their program designs. If you ignore performance and expect to add it back in later, you will generally be disappointed in the results!

Having said that, I must also warn against the other extreme - Wayne Stevens repeatedly stressed that you will generally not be able to predict where is the best place in your application for you to focus your tuning efforts. You may spend lots of time trimming a few seconds off some code which turns out only to be used occasionally. It is much better to run a performance tool on your application, using representative data, and find out where you should really be putting tuning effort. You can actually use the time you have saved by using an improved methodology to tune the parts of your system which make a real difference. One of the satisfying things about FBP is that, once you have improved the performance of a component, all future users of that component will benefit.

Similarly, you may decide to create a variant of a component to exploit a trade-off, and again you have increased the range of choices for future users. An example of this is a coroutine I built 20

years ago to read disk files faster than the standard reader, but at the cost of more storage. We had a large BDAM data set which had to be read as fast as possible. The regular sequential reader worked fine, as this data set could just be treated as a sequential data set (in MVS BDAM and BSAM files are held on disk in essentially the same format), but I also knew that all the records of a BDAM data set were the same length, which is not necessarily true for sequential files. Now the tracks of an IBM disk drive start at a point called the index point, so, to read a track, the access method has to wait until the index point comes around before starting to read the data (this is called "latency"). If you miss it, you have to wait a full revolution for it to come round again (and we could see this happening in our performance studies). I also knew that each record is physically labelled with a record number on the track, so I figured that it should be possible to start reading anywhere on the track, and just determine the record sequence afterward. Since this technique would be imbedded in an FBP component, it could still generate the records in the correct sequence. You do have to calculate the number of records per track first, but all the records are the same length, so there are standard formulae and tools for doing this. On the minus side is the fact that this approach takes up a track's worth of storage, and your code is somewhat tied to the device's characteristics; on the plus side, it's fast! In our case, this Read Track component was definitely worth it - it didn't get used widely, but, when we needed it, there was nothing like it! Remember also that in FBP there is never just one way to do a job, so nobody is forced to use your component. This component can just be added to the tools available for your programmers to use - they just have to understand its behaviour, not how it works inside. By the way, training programmers to think like users is not that easy! [This example has apparently been obsoleted by new hardware design of DASD, but I still think it's a valid example of a certain type of thinking, so I have left it in the text.]

Another kind of trade-off can be made in FBP by varying the capacity of connections. FBP's connections provide a powerful way of trading off CPU time against storage. All FBP schedulers so far have followed the strategy of having a process continue running as long as possible, until it is suspended for some reason. Now the more IPs there are in a process's input connection(s), the longer that process can go on before a context switch has to take place (assuming there is room in its output connections to receive the output). So, in general, larger capacity connections reduce context switches, and therefore CPU time. But, of course, larger capacities mean more IPs can be in flight at any point in time, so your application will take up more storage. Another way of looking at this is that you are effectively loosening the coupling between processes, so larger capacity connections equals looser coupling, whereas smaller capacity connections equals tighter coupling. For instance, we discovered that sequential readers perform best if their output connections can hold at least a block's worth of records.

The Read Track coroutine described above was used in conjunction with some other techniques to reduce a disk file scanning job from 2 hours to 18 minutes. The original running time of 2 hours was a cause for concern as we knew that the data we would eventually have to process was

several times the amount we were using in the 2-hour run. It therefore became imperative to see what we could do to reduce the running time of the job. We achieved this significant reduction by changing the shape of the network and adding one new component. This example used AMPS and is described in my Systems Journal article (Morrison 1978), but for those of you who may not have access to it I think it's worth repeating here, as it embodies some very important principles.

The application scanned chains of records running across many disk packs. One set of packs contained the "roots" of the chains, while another set of disk packs contained what we might call "chain records". Each root contained pointers to zero or more chain records, which in turn might contain pointers to other chain records, which could be spread over multiple packs. The problem was that our insert and delete code was occasionally breaking chains, so I was given the job of running through every chain looking for broken links - but obviously the scanning program could not use the same code that was causing the problem! So I had to write all new programs to do the scanning job. These programs understood our pointer structure, and there was enough internal evidence that I could always detect a broken chain.

My first approach looked like this:

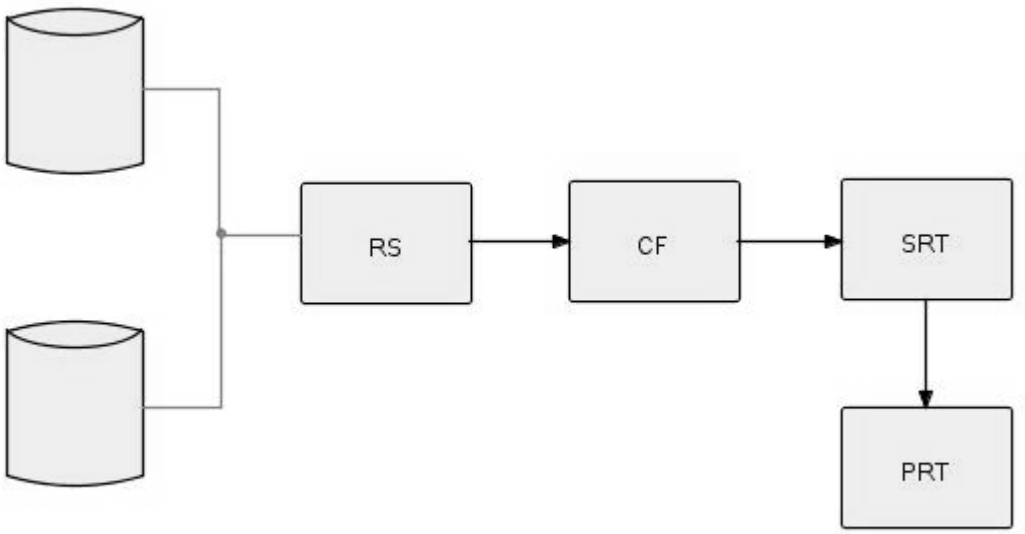



Figure 22.1

where RS means Read Sequential. This was the standard reader - we just concatenated its input files together, so its disk packs were read one after another. CF was the Chain Follower which

follows chains from one record to the next and outputs diagnostic information only when it finds a broken chain. SRT sorts the output of CF to put it into a sequence which is useful for humans, and PRT prints the results of the Sort process.

The reason for the Sort is that the root records were scattered randomly on the root disks, whereas it was more useful to see the errors ordered by, say, account number within error type. Since the number of errors should gradually diminish to zero as program bugs in the access methods were fixed, the overhead of the Sort process would diminish over time also.

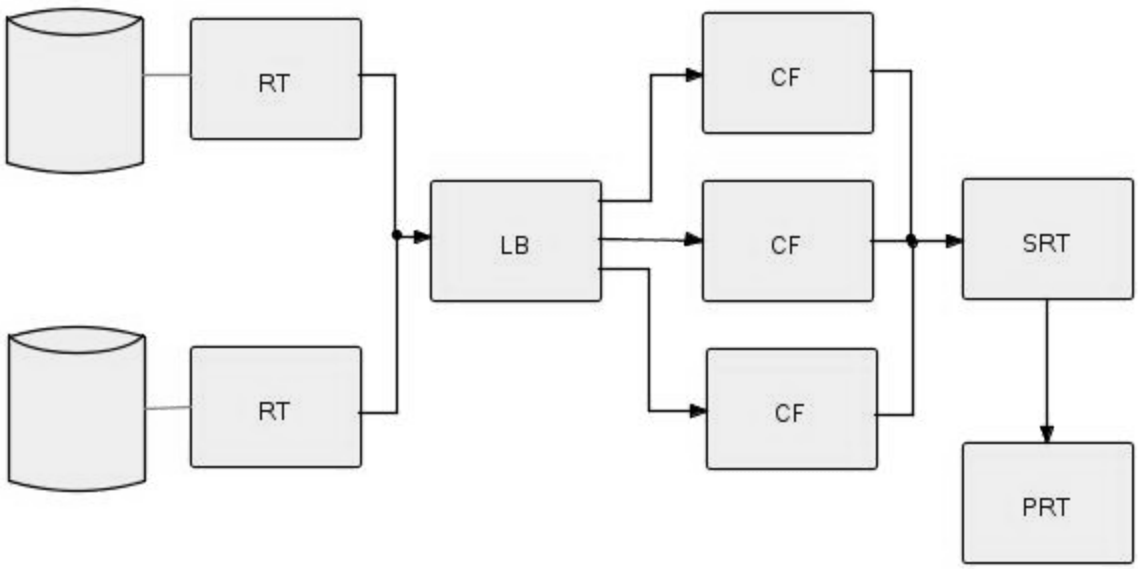
This program as shown above took 2 hours to run, even though we were using a test data base, so we absolutely had to speed it up!

My first realization was that the CF component was heavily I/O-bound, as it was spending most of its running time doing direct accesses to disk records, which usually included a seek, disk latency and then a read of a record into storage. I figured that, if we could run a large number of these processes concurrently, the I/O times of the different processes would tend to overlap. However, if we had too many parallel processes, they would start contending for resources, i.e. channels and arms. I therefore decided rather arbitrarily to have 18 of them running concurrently.

I also thought that, since the roots were scattered across multiple packs, we might as well assign each pack its own reader and let them run in parallel also. And, if I was going to do that, why not use the full track reader I described earlier in this chapter? Its output is data records, in the right order (although this wasn't even strictly necessary for this application), so the next process downstream wouldn't see any difference.

Lastly, I needed something to tie together these two kinds of process: I needed a process which would assign incoming root IPs to one of the CF processes, in a balanced manner. I could have just done this using a round-robin technique, but I wanted to try something a little "smarter": could I select the CF process which had the least work to do, and thus keep the load across all the CF processes in balance? I reasoned that the most lightly loaded process should have the smallest build-up in its input connection, so if I always sent a root IP to the CF process with the fewest IPs in its input connection, this would have the desired effect. I therefore wrote a load-balancing component which checked its output connections looking for the one with the smallest number of IPs. I feel this approach is quite general and could be used in many multi-server situations.

Here is a picture of the final structure:




where RT is the full track reader. LB is the Load Balancing coroutine, and the other processes are as before.

Figure 22.2

- Final result: elapsed time down from 2 hours to 18 minutes.
- Programming cost: one new generalized component (LB) and a change to the network. We might want to count RT, depending on whether we feel this was written for this purpose, or independently of it (it sort of happened along at the right time!). Furthermore, all these generalized components can be used in future applications.

The FBP diagramming tool, DrawFBP, supports the multiplexing function shown above using a simpler notation, which also lets the designer fill in the multiplexing factor - this is the result:

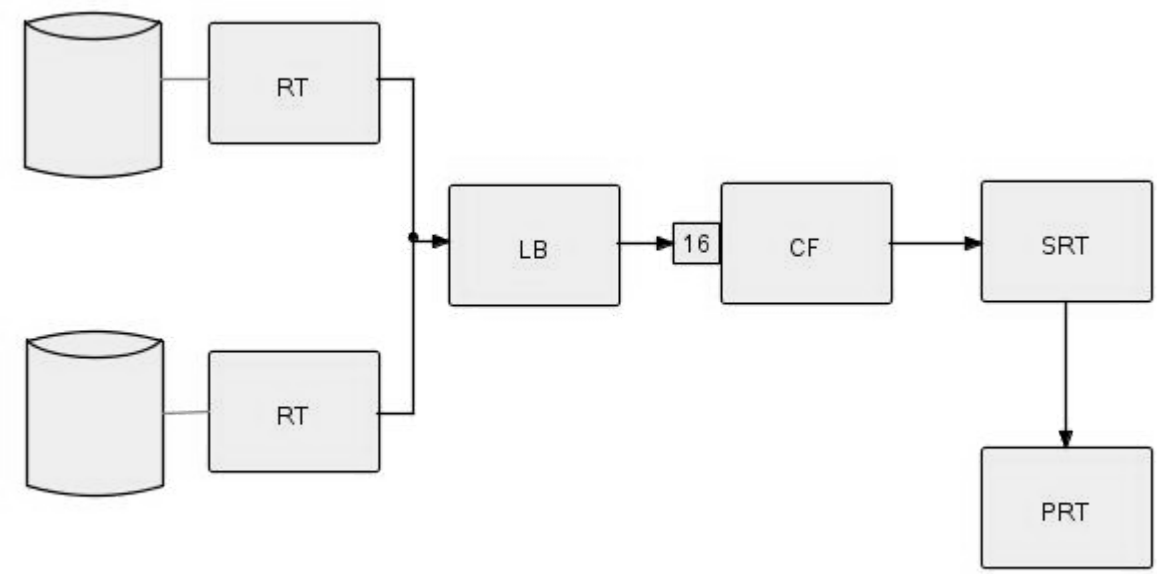



Figure 22.3

You may have been wondering how a component can check how many IPs are in a downstream connection when we have never mentioned a service to do this. This is an area of technology which I'm afraid is not "pure", but which I suspect comes up often when a concept has to come down out of its ivory tower into the world of real applications... We never did build an encapsulated service to count the IPs in a downstream connection and, if we had, it probably would have had to be a lot more complex. Instead we allowed one special component to peer into the insides of the infrastructure, so it effectively became an extension of the scheduler. I understood that it would have to change any time the internal structure of the scheduler changed, but that seemed better than providing a new service for the benefit of just one user. In hindsight it might have been even better to package the whole component as part of the scheduler, since then it would automatically participate in any changes made to the scheduler.

In DFDM, we had a very similar situation with the Subnet Manager, which also did strange things to the innards of the scheduler (you will remember that it can "revive" dead processes). DFDM's Time coroutine is a little bit like that too - this coroutine uses an existing, but unpublished, service to generate "clock tick" IPs, and is the only coroutine which is allowed to use this service.

Since I promised to talk about things which were not unalloyed success stories, I should mention

DFDM's NIP (Next Input Port) service. This is a service which we were enthusiastic about at first, but we later felt less excited about. Also, new people will always be joining projects, bringing different experience and tastes. In fact we did not include it in the Japanese version of DFDM. It is also appropriate to mention it in a chapter on performance, as it was intended as a performance tool. The idea was to be able to monitor which input port data had arrived at, and signal this fact to the component, which could then do a receive from that port. If no data was waiting at any input port, the component code could specify that NIP was to suspend until some data arrived, or that you didn't want to suspend. The latter option provided a way of polling (which I think should be avoided anyway); the former option is much the same as tagging all incoming IPs to indicate their source, and then merging them into a single input port on a first come, first served basis. So all NIP really saved us was the need to allocate a field in each IP to indicate the source number and the need to set it. Perhaps its main appeal was that it matched very closely one's mental model of how a server servicing requests from multiple sources should work - but, as we have said above, this can be handled very nicely, just using FBP's many-to-one connection facility, so you really don't need to provide a separate service to do this.

I mention these situations to describe a type of problem which will certainly come up from time to time. Under the pressures of deadlines, performance requirements or just because you get a neat idea, you will either be tempted to introduce small breaches in the bastions surrounding your system, or you will be persuaded to put in complex facilities for one or two users. Conversely, you may put your foot down and refuse to give in, and lose some customers. Should you give in to these temptations, or should you stick to your guns because you know the possible pitfalls if you don't? I can't answer that for you - all I can recommend is that you try to make sure you have thought through all the pros and cons before you make your decision! One hint: good code (no, make that any code) lasts a lot longer than you would ever expect and, when some dialect of FBP (hopefully) becomes more widely used, it will last longer still. Will you still be proud of your code 20 years from now? I sincerely hope so, because there's a good chance you'll meet it again! And, no, customers are not always right, but they are not always wrong either!

# Chap. XXIII: Network Specification Notations

*[Since the book "Flow-Based Programming" was written, a Java version of FBP has been built, now called JavaFBP. The code specifying a JavaFBP network is described in JavaFBP Network Definitions. This is an example of what might be called a "procedural" specification, and is similar to that used by a number of the systems mentioned in Cognates, such as NIL, developed by Rob Strom and his team at IBM. Although a network diagram can be converted fairly easily into such a specification, they are not easy to grasp directly - but then that is probably true even for the notations shown below when they become fairly large! The real answer, as always, is a multi-level diagram, supporting step-wise decomposition - see DrawFBP.]*

FBP applications start off life as pictures or hierarchies of pictures, which have to be made "machinable" for execution. During most of our twenty years [as of 1994] of experience with FBP, we have never had advanced picture-drawing tools, so we tended to draw networks using pencil and paper, and then convert them by hand into executable networks. Although we have experimented with different notations over the years, the information which needs to be captured has remained fairly constant. It is certainly true that FBP lends itself to visual programming techniques, so what I am going to describe will become less important as time goes on, but, for now, we still have to have notations which allow networks to be input directly into the machine.

The diagrams typically used for Structured Analysis represent processes and their connections. To turn such a diagram into an FBP diagram, which can actually be executed, you only have to add the following data:

- port names (or numbers)
- parametrization
- automatic ports

For ease of understanding, you could always add the following to Structured Analysis diagrams:

- stream descriptions
- process descriptions (e.g. "merge masters and details")
- icons for external objects, users, etc.

The same is true for FBP diagrams. Since we have mostly used pencil and paper, we never really worried too much about the exact diagramming conventions to be used. In fact, if some Structured Analysis diagramming convention is in use in a given shop, it makes sense to just expand on whatever convention was used for analysis.

Clearly there is a great opportunity for powerful picture-drawing tools to assist in the design and development process. It would be nice to be able to "explode" a process and see its structure at the next lower level [DrawFBP, described elsewhere on this web site, supports this], or look at a connection and see the data stream that passes over it. When building networks, it would be even nicer to be able to pick up an icon for, say, Collate, and place it on the diagram, complete with "sticky" ports ready to be connected to its neighbours.

We have drawn a number of pictures in what has gone before, but we have yet to talk about how we get these into the machine to be executed.

The very first FBP software, AMPS, was written in IBM S/360 Assembler language, and networks were also expressed using Assembler macros. Each process was specified using a single macro call, which also listed the named connections between that process and the other ones. In what follows, I will use a simple network and show how it was specified to the system in the various FBP dialects. Let us take as our sample network the following:

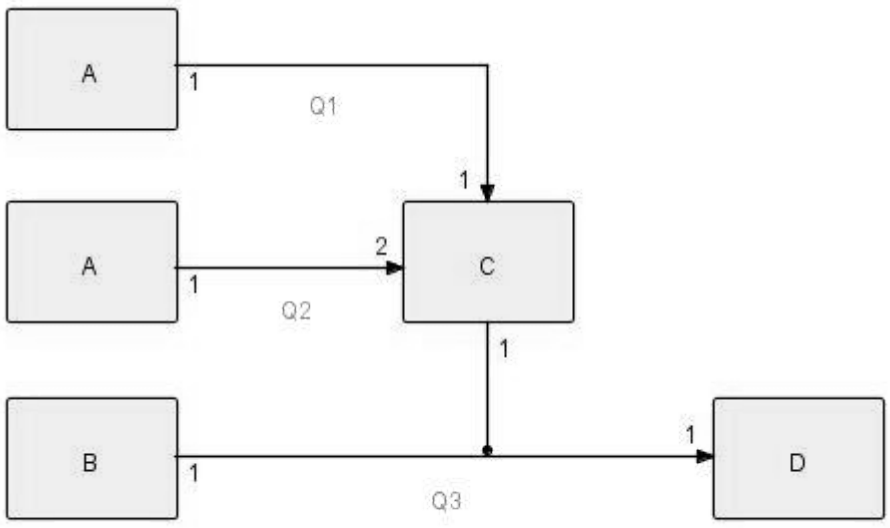



Figure 23.1

In AMPS and DFDM ports are specified numerically, so I have marked the ports as numbers in this diagram. (In DFDM input and output ports were numbered separately, while in AMPS one set of numbers covered both - this diagram follows the DFDM convention).

In AMPS this would have been specified approximately as follows (we will use `PROCESS` rather than the actual macro name used, to show the analogies with present-day terminology):

```
PROCESS PROG=A,AQ=Q1
PROCESS PROG=A,AQ=Q2
PROCESS PROG=B,AQ=Q3
PROCESS PROG=C,Q=(Q1,Q2),AQ=Q3
PROCESS PROG=D,Q=Q3
```

Figure 23.2

where the `PROCESS` macro shows a process, `Q=` lists the queues inputting to the process (feeding the process), and `AQ` lists the queues fed by the process. Communication between processes is thus set up by *naming the connections* (called "queues" in AMPS). [There is no other reason to name connections.] Numbering is sequential, starting with `Q=` and then going on through `AQ=`. Parameters could also be specified in the network, and were added using `PARM=` as an address-type parameter on the `PROCESS` macro, when required. As each macro describes a separate

process, there is no need to distinguish between different uses of the same component (indicated by the `PROG=` parameter).

Notice a concept common to all the varieties of FBP: when more than one output port is connected to one input port, *it is treated as one connection (queue)*. Connections can be many-to-one, but not one-to-many.

Parameters in AMPS were more general than DFDM's: an AMPS parameter could be a data structure of any form, while DFDM parameters were always variable length character strings - DFDM's were consistent with the format of parameters passed to a job step by the MVS operating system.

DFDM has two different notations: *interpreted*, which is used to build the network dynamically at run-time, and *compiled*, where the control blocks for the network are built as a single Assembler program, which when compiled and link-edited results in a ready-to-run network control block structure. The interpreted notation allows a program to be modified and tested many times during a session. Test components can also be added easily to monitor the contents of connections or to create test data. The compiled format does not have the interpretation overhead, but takes a little longer to generate, so is more appropriate when a network is put into production or you want to study a program's performance. Once you are ready to go from interpreted to compiled format, you use a utility program (called "Expand") to do the conversion. The other main difference is that, in the interpreted case, components are loaded dynamically at run-time, while in the compiled case all the components are linked together, along with the network, into a single load module [a single executable block of code, rather like a present-day .exe file] .

Both of these DFDM notations are based on specifying a list of connections, rather than processes. These connections specify the processes which they connect, but are themselves unnamed. Processes are named by component name, plus an optional qualifier, where a qualifier is indicated by a period. The interpreted notation uses arrows (`->`) (originally also ampersands (`&`)) to indicate connections. Port numbers are shown *before and after* the connection mark (if not specified, the default is 1). Connections can be strung together in "paths", with a comma marking the end of a path. Earlier versions of DFDM did not mark the end of a network, but the Japanese version used a semi-colon. So the above diagram in DFDM interpreted notation would read as follows (since in DFDM input and output ports are numbered separately, `C` has input ports 1 and 2, and output port 1):

```
A -> C -> D,
A.X -> 2 C,
B -> D;
```

Figure 23.3

where we have distinguished the second occurrence of A with the qualifier X (any character string would have done).

The corresponding compiled specification would then be:

```
X       NETWORK
        CONNECT A,C
        CONNECT C,D
        CONNECT A.X,(2,C)
        CONNECT B,D
        NETEND
        END
```

Figure 23.4

As can be seen, both notations use the same qualifier notation.

One other major difference between the interpreted and compiled notations of DFDM is that the interpreted notation is hierarchical, whereas the compiled is "flat" (it is "flattened" when it is expanded, so that the whole network becomes a single module). In the interpreted notation, you can define a subnet, call it G, with "sticky" connections, e.g.

```
G:  -> A -> 2 B ->, I -> B
```

Figure 23.5

would mean that G has one input port and one output port which will be connected when G is used. Arrows with an open end are the external interfaces of the subnet, in this example, the arrow feeding A and the arrow coming from B.

G's picture therefore looks like this:

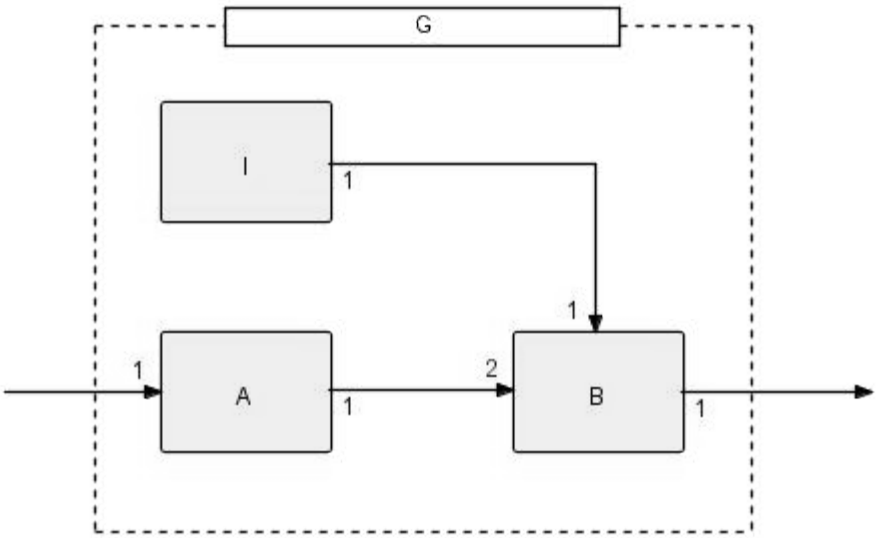



Figure 23.6

`G` has two external ports: one input and one output. These are the arrows which cross the "boundary" of `G` (shown as a dotted line). `G` can therefore be used as an ordinary filter in a network, and, like all components, can be used more than once in a network.

Here is a network which uses `G`:

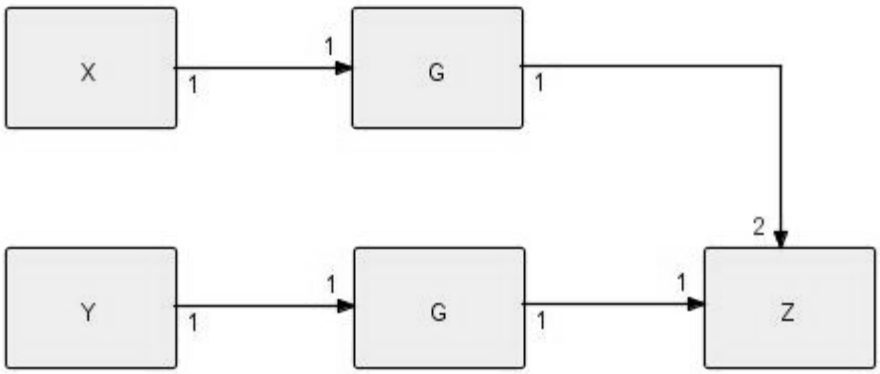



Figure 23.7

This might be coded as follows in the interpreted notation:

```
X -> G -> 2 Z,
Y -> G.2 -> Z
```

Figure 23.8

Note that composite components can be qualified just like elementary components - at this level, the network doesn't know that `G` even is a composite. If this network is then expanded to form a single flattened network, you will see two instances each of `I`, `A` and `B`, as follows:

```
X       NETWORK
        CONNECT X,A.1
        CONNECT I.1,B.1
        CONNECT A.1,(2,B.1)
        CONNECT B.1,(2,Z)
        CONNECT Y,A.2
        CONNECT I.2,B.2
        CONNECT A.2,(2,B.2)
        CONNECT B.2,Z
        NETEND
        END
```

Figure 23.9

The Expand function automatically assigns qualifiers to all the resulting processes. Composite

components like G can be coded in the same file or file member as the network which references them or in a separate file or file member.

When the network using G is expanded, you see that G loses its identity and merely becomes a pattern within the total network. Since I has no inputs, and G has lost its identity at run-time, all the I's will start up at start of run.

One other thing to notice is that the expanded list is not very easy to grasp. People generally find that it is easier to work with the interpreted form for two reasons: you can see the effect of a modification immediately, and also an individual network or subnet is easier to understand when it has no more than about 7 processes (human short-term memory seems to be able to handle 7 plus or minus 2 objects), so you can build up quite complex applications, as long as they are built up out of layers none of which exceed more than about 7 processes. Ideally of course this should be done using a graphic interface, but we did not have such tools available.

In the experimental THREADS system mentioned above [the C implementation of FBP concepts], we have both an interpreted and a compiled network specification. Here is a rather trivial example of a THREADS network in the interpreted notation to illustrate some of its features:

```
'data.fil' ->
    OPT Reader(THFILERD) OUT ->
    IN Selector(THSPLIT) MATCH ->
    IN Replicate(THREPL) OUT[0] ->
    IN Display(THVIEW),
    Replicate OUT[1] -> IN Display,
   '11,2,0' -> OPT Selector;
```

Figure 23.10

This represents the following network:

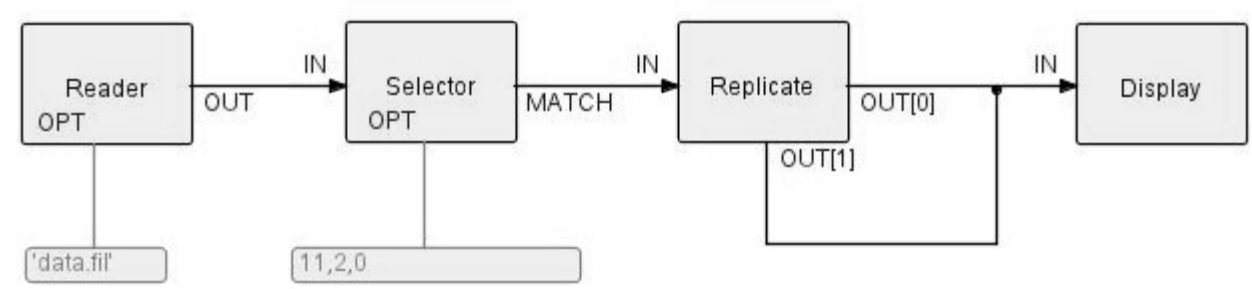



Figure 23.11

The above notation follows DFDM in that you trace a path until you hit a dead end, insert a comma and then start on the next path, and so on until the whole network is specified.

Where DFDM named processes by giving a component name plus qualifier, THREADS names processes directly, and attaches a component name in brackets to one of the occurrences of the process name. Ports are named, rather than numbered, so the notation for a connection is:

```
... process-name port-name -> port-name process-name ...
```

Figure 23.12

As befits the "C" basis of THREADS, elements of array-type ports are identified by

```
port-name [ element-number ]
```

Figure 23.13

numbering up from zero.

The end of a network is marked with a semi-colon.

The other major way in which THREADS differs from AMPS and DFDM is its concept of Initial Information Packets (IIPs). These are shown as quoted strings followed by arrows (indicating that these are really a special form of connection), e.g.

```
'data.fil' -> OPT Reader(PROCR) ....
```

Figure 23.14

This attaches the string "`data.fil`" to port `OPT` of `Reader` in such a way that when `Reader`

does a `receive` from its `OPT` port, the string is turned into a "real" IP and becomes accessible to `Reader`'s code. After processing, `Reader` has to dispose of it as usual. If `Reader` does a `receive` from that port again, it will receive an "end of data" indication.

Automatic ports can be specified by putting an asterisk in place of a port name.

You can also specify subnets following the main network in the same file, or as separate files, which are started with a label (symbol followed by a colon) and terminated with a semi-colon. Here is an example of a THREADS network definition together with some subnets:

```
'data.fil' -> OPT reader(R1) OUT ->
  IN  thcount(THCOUNT) COUNT ->
  MIN display(C1),
  'H' -> MOPT display;
C1:
 MIN => IN process_C(THVIEW),
 MOPT => OPT process_C OUT => MOUT;
R1:
OPT => OPT reader(THFILERD) OUT => OUT;
```

Figure 23.15

Process names only have to be unique within a given subnet, so the two uses of `Reader` do not conflict. You will also notice that the port names on the subnets are identified with the corresponding internal ports by means of an arrow-like equals sign (=>). This is to stress the fact that these are not really connections, but are more like equivalences. In both DFDM and THREADS, a subnet may specify ports which are not used in the network referencing it, but the reverse situation is not allowed (if a port is referenced at a higher level, it must be specified in the definition of the subnet).

THREADS also has a compiled format - it differs from DFDM in that it does not generate the control blocks directly, but is still a list of the connections, in fixed format rather than in free form. The advantage of this is that we can change the implementation of the THREADS Driver and the format of its control blocks, without the user having to recompile all of his or her applications.

I have gone into some detail on the various approaches to notations because, for many jobs, the network notation is all the code a programmer ever has to write! Whether we name the connections or the processes, all we are really doing is telling the machine about a list of connections. As such, sequence really doesn't matter, so, even if you feel this should be called a programming language, it is definitely a non-sequential one! It also has a natural relationship with pictures, so in the future we hope there will be graphical tools which will make entry of this information into the machine even easier. [Such a tool has since been developed - called DrawFBP - and is described in Appendix D, recently added to the online version of the book.]

For debugging conventional languages, we are just starting to get packages which allow the developer to walk through a program interactively - a graphical aid to debugging FBP networks should allow you to monitor the data passing across a connection, or track a single IP as it travels through the whole network and observe its transformations. As with any debugging tool, the real challenge is to provide ways for the developer to build mental models, both of the program the way it should be working, and also of how it is going wrong.

# Chap. XXIV: Related Compiler Theory Concepts

*[References to TR1 in the text refer to an application component which is used in examples earlier in the book - see Chapter 10. TR1 "extends" detail records by multiplying a quantity in a detail record by a unit price obtained from a master record in another file. Processes "up-stream" of TR1 have merged masters and details, so TR1 is fed a single merged stream containing masters and details, and with related records delimited by special IPs called "brackets".]*

It has been noted by several writers that there seems to be a good match between compiler theory and the theory of sequential processes. FBP processes can be regarded as parsers of their input streams.

The component labelled TR1 in the example in Chapter 10 can be considered as a "parser", while the structure of its input stream can be specified using a syntactic form called a regular expression. This allows a number of the concepts of modern compiler theory to be applied to it. Let us express the input stream of component TR1 in that application as a "regular expression", as follows:

```
{ ( m d* ) }*
```

which can be read as:

zero or more groups of:

```
one open bracket,
one master IP (m),
zero or more details (d),
one close bracket
```

(In what follows, I will use brackets to represent bracket IPs, and curly braces will have syntactic meanings.)

Each of TR1's output streams consists of repeating strings of identical IPs. Representing modified masters as *m'*, extended details as *d'* and summaries as *s*, we get the following "regular expressions" for the three output streams:

```
m'*
d'*
s*
```

Each of these output streams has a well-defined relationship with the input stream, and it would be very nice if some notation could be developed to show this relationship clearly and concisely. In fact, this would constitute a complete description of what TR1 does: a component's function is completely described by its input and output streams, and the relationships between them.

In Aho and Ullman's 1972 book they show that a regular expression can be parsed by a deterministic one-way finite automaton with one tape, usually referred to simply as a "finite automaton". This tape contains "frames", usually containing characters, but in what follows we shall think of the frames as containing entire IPs. As each frame is scanned, the automaton may or may not change state, and may or may not perform an "action", such as reading or writing a frame. Whether or not there is an action, the tape continues moving.

A convenient way to represent the various state changes is by means of a "Transition Graph", where each node represents a state, and each arc represents a transition from one state to another (or from a state to itself), and is marked with the name of the IP being scanned off, as follows:

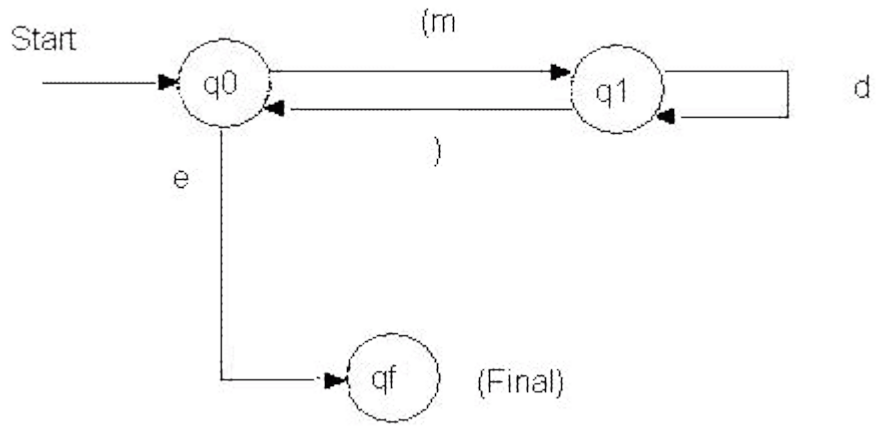



Figure 24.1

In these diagrams, *e* is used to indicate the "end of data" condition; *m*, *d*, *(* and *)* represent masters, details, open and close brackets, respectively. Since an open bracket is always followed by a master in this example, I have shown them combined on a single transition arc.

An alternative to the transition graph is the "State Table". This uses a tabular format, which is often more convenient. The following table is equivalent to the preceding diagram.

| State | Input | New State |
|---|---|---|
| q0 | (m | q1 |
| q0 | e | qf |
| q1 | d | q1 |
| q1 | ) | q0 |
| qf | | Final State |

Figure 24.2

Aho and Ullman next show that any regular expression may be described by a right linear grammar, expressed as a series of productions, each of which describes a pattern in terms of the sub-patterns or final symbols which make it up.

Productions can be understood in a generative sense - i.e. all legal streams can be generated by successively taking choices from the set of productions, starting with *S*. Alternatively, productions can be used to describe relationships between patterns being scanned off by a parser.

A grammar is called "right linear" if patterns on the right-hand side may only occur at the end of a production. Here is a right linear grammar which is equivalent to the "regular expression" given above, using the notation of productions, but using lower-case letters to represent IP types, as in the state diagram above. $\Lambda$ means a null stream.

```
S -> Λ
S -> (mR
R -> )S
R -> dR
```

where the two lines starting with *S* indicate two patterns which both constitute valid *S*'s, and similarly for the two lines starting with *R*.

Since lower-case letters in the productions represent objects that cannot be parsed further (i.e. IPs), upper-case letters may be thought of as representing expectations or hypotheses. Thus *R* in the above productions represents "that part of a stream which can follow a master or detail". Thus, when a master is detected, the automaton's expectation of what may follow changes from

what it was at the beginning of the stream, and, when a close bracket is detected, it changes back. This exactly mirrors the state changes of the finite automaton shown above.

While the right linear grammar shown above adequately represents the automaton traversing the stream one IP at a time, it cannot "see" patterns which consist of more than one IP, and therefore cannot express one's intuitive feeling for the hierarchic structure of the IP stream. For this we must go to a more powerful class of grammar, called the context-free grammars.

The set of context-free grammars (CFGs) is a superset of the set of right linear grammars, in that a pattern symbol is allowed to occur anywhere on the righthand side of a production. Since CFGs are more general than right linear grammars, they require a more complex type of automaton to process them: the "pushdown automaton" (PDA), which has, in addition to its input tape, memory in the form of a "pushdown stack". This may be thought of as containing subgoals or hypotheses established during stream processing, to be verified or discarded.

The following set of productions of a context-free grammar is also equivalent to the regular expression given above:

```
S -> Λ
S -> AS
A -> (mT
T -> )
T -> dT
```

where you will notice that pattern *A* appears at the beginning of the right-hand side of the second production. The first two productions together can therefore be read as follows: a stream consists of zero or more *A* patterns (or substreams).

A pushdown automaton will have "stack states" in addition to "automaton states": in this example the automaton state is trivial, as there is really only a "normal" state (*q0*) and a "final" state (*qf*), to enable the automaton to stop.

In the following diagram, *Q* means "automaton state" and *S* is the "stack state" ( - means empty, and *x* means that there is an *x* at the top of the stack). It will be noticed that "push" and "pop" change the "stack state" in the same pattern we saw in connection with the "automaton state" in the Transition Graph shown above. *Q′* and *S′* mean the new states of the automaton and the stack respectively.

```
Q     S  Input |  Action    Q'    S'
---------------|----------------------
q0    -    (m  |  push A    q0    A
q0    A    )   |  pop A     q0    -
q0    A    d   |  pop A/    q0    A
```

```
                |   push A
                |
q0    -    e    |    -       qf    -
                |
qf              |    Final State
```

```
The actions denoted
push x
pop x
are to be read as "push IP x onto stack," and "pop IP x from stack,"
respectively.
```

```
Figure 24.3
```

You will recognize that this use of the stack exactly parallels the use of a stack in FBP to hold IPs being worked on.

In FBP we restrict the use of the stack as follows: while an IP is in the stack, it is not available for use by the component's logic. It must therefore be "popped" off the stack to make it available for processing. At the end of processing, it must be explicitly disposed of before the automaton leaves that state, either by being returned to the stack, or by being sent or dropped. This is the reason for the pop/push combination - if we weren't using an FBP implementation, we could leave it out altogether, as it leaves the stack state unchanged, but in FBP it is required in order to make A available for processing.

It can be seen from the above that the only function of the state Q is to determine when the automaton has reached a final state - the rest of the time the automaton is in its normal state. The two states of the stack correspond one-to-one with the states *q0* and *q1* in the non-pushdown automaton, so that the stack has a dual function: that of storage for IPs being worked on, and control. This exactly mirrors the way FBP uses a stack for holding control IPs. The process of stacking a subgoal which represents a substream corresponds to the FBP concept of stacking an IP to denote the entire substream.

Where do we go from here? There has been quite a lot of work on describing applications using PDAs, but I am not aware of much work tying them together with data streams (I would appreciate hearing about such work). This seems to me a fruitful direction for more research, and may yield new ways of looking at applications, or even new hardware designs.

# Chap. XXV: Streams and Recursive Function Definitions

*[References to TR1 in the text refer to an application component which is used in examples earlier in the book - see Chapter 10. TR1 "extends" detail records by multiplying a quantity in a detail record by a unit price obtained from a master record in another file. Processes "up-stream" of TR1 have merged masters and details, so TR1 is fed a single merged stream containing masters and details, and with related records delimited by special IPs called "brackets".*

*Collate refers to a general, "black box", reusable component that merges 2 or more streams on the basis of parameters describing the location of control fields in its incoming IPs, and optionally inserts grouping IPs called "brackets" into its output stream. It is described in some detail in Chapter 8.]*

It is easy to see that an FBP component can be regarded as a function transforming its input stream into its output stream. These functions can then be combined to make complex expressions just as, say, addition, multiplication, etc. can be combined in an algebraic expression. A number of languages have used this as a base for notations. Let me take a simple example:

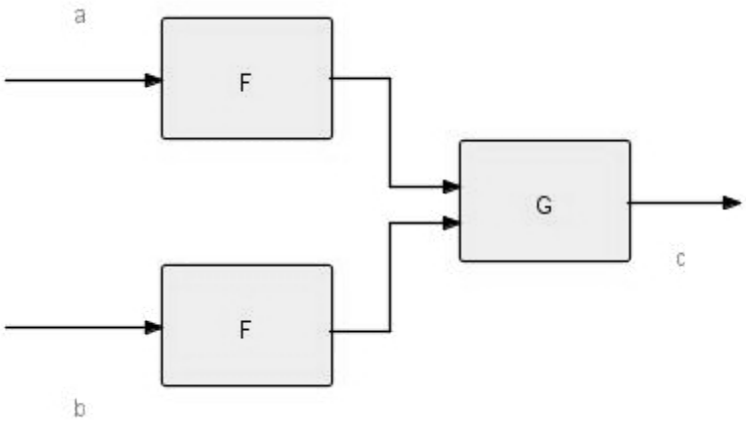



Figure 25.1

If we label streams as shown with lower case letters, then the above diagram can be represented succinctly as follows:

```
c = G(F(a),F(b))
```

I deliberately used *F* twice to underline the fact that the same function can be used many times in the same expression. Of course, this relies on the function having no side effects - this is one of the desirable characteristics of functional languages (and one of the qualities which are desirable when designing FBP components). We will return to this point later on.

Now we have seen that streams are made up of patterns of IPs, which in turn have fields or data items. Is it possible to carry functional notation to the point where we can actually build systems processing real data? I believe it is, but in a way that will not be as "mathematical" as most treatments of recursive programming. In the rest of this chapter, I will develop the concepts of recursive stream definitions. But first, for those of you whose math is a bit rusty, we should talk about what exactly are recursive definitions. The rest of you can skip ahead!

Recursive techniques are often taught using the formula for factorials as an example. A factorial is the product of all the integers greater than zero up to and including the number whose factorial you want to calculate. Factorials can be (and often are) calculated iteratively. i.e.

```
factorial(x):
    a = 1
    do varying y from x to 1 by -1
        a = a * y
    enddo
    return a
```

Now, the classic expression for calculating factorials really does the same thing, but it does it recursively, as follows:

```
factorial(x):
    if x = 1
        return 1
    else
        return x * factorial (x - 1)
    endif
```

Here we see that "factorial" is actually defined in terms of itself, but used on a "smaller" part of the problem. Since we use factorial on *x-1* , and the first test is always to see if we have reached 1, we are actually getting one step closer to our goal each time around the definition. If you want to visualize how this might execute, think of a stack holding the environment for each invocation of factorial. Each time we start a factorial calculation, we push the information we will need onto the stack, so the stack gets deeper and deeper. When we eventually get a true condition on the first test, we can calculate a factorial (1) without pushing a new environment on the stack. Now we can finish all the factorial calculations (in reverse sequence), so that eventually we arrive back at the top of the stack and we're finished. Speakers of German will be familiar with a similar phenomenon which occurs in that language, where several "contexts" may be "stored" until the end of the sentence, at which time each context is terminated with the production of an infinitive or past participle.

The advantage of the recursive definition is that it has no local storage variables. We therefore suspect that this characteristic may have a bearing on some of the problems we identified earlier in this book with the "pigeon-hole" concept of storage. Functions also ideally have no side-effects, so they can easily be reused. If we combine these concepts with some other concepts which have appeared in the literature, we can actually describe (a small piece of) business processing in a way which is free from a number of the problems which bedevil the more conventional approaches. I also believe that if you look back at Chapter 18, you will find that there are similarities between that notation and what will be pursued more rigorously in this chapter. Admittedly this will be a tiny example, but my hope is that someone will be sufficiently intrigued that they will carry it further.

Recursive functions are attractive to mathematicians because they have no side-effects, and therefore are easier to analyze and understand. In W.B. Ackerman's paper (1979) he states: "the language properties that a data flow computer requires are beneficial in their own right, and are very similar to some of the properties that are known to facilitate understandable and maintainable software, ...." Some of these beneficial properties are as follows:

- locality of effect
- freedom from side effects

- "call by value"

Ackerman defines an applicative language as one which does all of its processing by means of operators applied to values. The earliest known applicative language was LISP.

By now you should be familiar with the use of the term "stream" in FBP to describe the set of IPs passing across a particular connection. The stream does not all exist at the same time, but is continuously being generated at one end and consumed at the other. However, it has a "real" existence, and it can be manipulated in various ways. W.H. Burge (1975) showed how stream expressions can be developed using a recursive, applicative style of programming. D.P. Friedman and D.S. Wise contributed a number of papers relating applicative programming to streams by adding the concept of "lazy evaluation" (1976) to Burge's work. This style has the desired freedom from side effects and has another useful characteristic: the equals sign is *really* a definition statement, and can be used for proving the validity of programs as well as for doing the actual data processing. Such a language is called "definitional".

Quoting W.B. Ackerman (1979) again: "Such languages are well suited to program verification because the assertions one makes in proving correctness are exactly the same as the definitions appearing in the program itself." One can put a restriction on assignment statements to the effect that the definition should not assign a value to a given symbol more than once in a single scope. Thus a statement like

```
J:=J+1
```

is ruled out because "J" would have to be given an initial value within the scope, resulting in two assignments within that scope. Also, viewed as a definition, it is obviously a contradiction! A number of writers on programming have described their feelings of shock on their first encounter with this kind of statement, only to become so used to it over the years that they eventually don't notice anything strange about it!

We shall now talk a little bit about the possibility of writing programs in a way that avoids the problem of rebinding variables within a scope, following (Burge 1975) and (Friedman and Wise 1976) in the area of stream functions.

Typically, like the definition of factorial shown above, the desired output is defined in terms of two functions:

- a function of the first item in the stream
- a function relating this item to the rest of the stream
- and a termination rule specifying a value to be returned when the function bottoms out.

The following example resembles the definition of factorial shown above, but moves closer to business applications. Even though it may not much resemble the type of business applications

that you are accustomed to, you will have to admit that it is very compact! Suppose we want to count all the IPs in a stream. Then, analogously to the factorial calculation above, we could write a "counter" function F as follows:

```
F(S) = if  S is null,
     then  0,
     else  1 + F(rest(S))
```

where the result is specified directly by a value, e.g. *0*, or *1 + F(rest(S))*.

With respect to the rest of this notation, functions are expressed using the conventional bracket notation, and sublists will be specified by means of curly braces. For instance, *{w,x,y,z}* is a sublist consisting of *w*, *x*, *y* and *z*. *null* tests for a stream with no IPs in it, i.e. {}; *first(S)* is the first IP of a stream, i.e. *w* in the above example, and *rest(S)* is a list comprising all the rest of the IPs in the stream, i.e. *{x,y,z}*. Notice that, while *rest* returns a list, *first* only returns a single IP.

In this first example, *F* is called recursively to return a value based on processing a stream *S*. At each invocation of *F*, its environment (the part of the stream that it can see) is pushed down on a run-time stack. When an invocation of *F* finds itself looking at an empty stream, the null test returns true, *F* bottoms out, and the stacked environments are progressively popped up, until the original one is reached, at which point the process stops. Notice that there is a family resemblance between recursive definitions which only "recurse" at the righthand end, and right linear grammars (described in the previous chapter). This kind of recursive definition can also have special processing applied to it which maintains the stack at a constant depth.

Now so far we haven't actually used the data in the stream IPs. Suppose therefore that we want to sum all the quantity fields in a stream of IPs. We will introduce the convention *a:x* to mean "field a of x". This notation and the "mini-constructor" notation described below are based on the Vienna Definition Language, developed some years ago by the IBM Laboratory in Vienna. The desired calculation can now be expressed recursively as follows:

```
F(S) = if    S is null,
       then  0,
       else  q:first(S) + F(rest(S))
```

where *q* is the quantity field of an IP.

So far, we have only generated a single quantity from our stream. To do anything more complicated, we will need to be able to generate IPs and string them into streams. To do the former, we will need something which can build an IP given a set of values for its fields. We will use a special function for this which I will call the "mini-constructor" (μ). This takes as its argument a list of selector symbols and values, and returns as a result an IP with those values inserted into the fields designated by the selectors. A selector and its value are separated by

commas, while selector/value pairs are separated by semi-colons. The mini-constructor is a concise way of specifying how new IPs are to be built.

To combine IPs into a stream, we use a variant of the well-known list-processing function *cons*, which was first used in functional languages to join two lists together. The following equivalence holds:

```
a = cons(first(a),rest(a))
```

Friedman and Wise (1976) have extended this concept by removing the requirement that both of the arguments of *cons* be available at the same instant of time. Their "lazy cons" function does not actually build a stream until both of its arguments are realized - before that it simply records a "promise" to do this. This allows us to imagine a stream being dynamically realized from the front, but with an unrealized back end. The end of the stream stays unrealized until the very end of the process, while the beginning is an ever-lengthening sequence of items.

Suppose we want to create a stream of extended details (where the quantity field has been extended by the unit price for the product): if the unit price were repeated in every IP, there would be no problem, as all the information that a particular call to the function requires is immediately available. Assigning selectors *p*, *s*, *d*, *q*, *u*, *e* to product number, salesman number, district number, quantity, unit price and extended price, respectively, we would then be able to define the extended details stream as follows:

```
F(S) = if S is null,
        then null
        else cons (E(first(S)), F(rest(S)))

E(x) = μ(p,p:x; s,s:x; d,d:x; u,u:x; e,q:x*u:x)
```

where *E* is a function which creates a single IP. The expression after the last semi-colon in the definition of *E(x)* would read in English: "set the extended price field of this IP to unit price multiplied by quantity". This would be fine except for the fact that unfortunately the unit price is not in the same IP as quantity!

In fact, in the case of the merged stream being processed by TR1, the unit price for a given product is held in only one IP per substream - namely the master IP for that product. We could define a function "PM" ("previous master"), but that would violate the rule that a function can only "see" an IP optionally followed by its successors. Instead, let us broaden the concept of *first* and *rest* to work with lists of lists (just as LISP and the other functional languages do).

This concept of lists of lists allows us in turn to take advantage of the fact that a stream can automatically be structured into substreams by a Collate type of component. Thus, suppose the output of a Collate run is as follows:

```
((m1,d11,d12),(m2,d21),(m3,d31,d32,d33))
```

where brackets represent bracket IPs. Then to convert this logically into a structure of lists and sublists, merely replace the open and close bracket IPs with curly brackets, i.e.

```
S:= {{m1,d11,d12},{m2,d21},{m3,d31,d32,d33}}
```

Now *first* and *rest* at the list level will return sublists, i.e.

```
first(S):= {m1,d11,d12}
 and
rest(S):= {{m2,d21},{m3,d31,d32,d33}}
```

Note that, analogously to what we saw above with simple lists, *first* reduces the "nesting level" of a list by one level, while *rest* leaves it unchanged. The processing for extended details can now be shown succinctly as follows:

```
F(S) = if S is null,
      then null
      else cons(G(first(S)), F(rest(S)))
G(x) = G'(first(x),rest(x))
G'(x,y) = if y is null,
      then null
      else cons (E(x,first(y)), G'(x,rest(y)))
E(x,y) = μ(p,p:y; s,s:y; d,d:y; e,q:y*u:x)
```

Here the one-argument function *G* is defined in terms of a two-argument function *G′*, whose first argument is always a master IP, and *E* now takes two arguments instead of only one.

In the above it can be seen that we will generate one output IP for each incoming detail (all IPs within a lowest level substream are details except the first one, which is the master). We can also see that *x* in *G′* acts as a place-holder for the master IP - it remains unchanged throughout the whole evaluation of function *G*.

Now let's repeat the three tables from Chapter 18 which describe the same calculation, and you will see that essentially the same information is captured, without the use of recursion, but also without using any variables. It seems therefore that these tables must have an underlying relationship with the functional expressions shown above.

The three tables from Chapter 18 respectively describe

- the relationships between input and output streams:

```
INPUT STREAM
1. Merged Input
 2. Product Substream: REPEATING
  3. Product Master
```

```
  3. Sales Detail: REPEATING
OUTPUT STREAMS
1. New Masters
 2. Product Master: ONE PER Product Substream
1. Sort Input
 2. Sales Detail: [ONE PER Sales Detail]
1. Report-1
 2. Product Summary: ONE PER Product Substream [,NEW]
```

- the layout of the individual IPs, e.g.:

```
Extended Detail:
{
  Product Number: IDENT,
  Salesman Number: IDENT,
  District Number: IDENT,
  Quantity: QUANTITY,
  Unit Price: $CDN,
  Extended Price $CDN;
}
```

- and the derivation rules for computed values:

```
       Name             |       Derivation
  ------------------------------------------------
                        |
   New Master           |
     Year-to-date Sales |   Year-to-date Sales
                        |      [IN Product Master]
                        |         + Product Total
                        |
   Extended Detail      |
     Extended Price     |   Unit Price * Quantity
                        |
                        |
   Product Summary      |
     Product Total      |   SUM of Extended Price
                        |       [OVER Product Substream]
                        |
     Year-to-date Sales |   SAME as IN New Master
                        |
```

All these items [well, many] can be found in a more mathematical form in the recursive expressions given above, but it appears that, in this particular case, the same information content can be expressed almost as succinctly using a quite simple notation which does not require any mathematical expertise at all. We have alluded above to the need to minimize the gap between the business requirement and the means of expressing it. What is the absolutely most concise way

of expressing the requirement "build a stream of IPs containing extended quantities calculated as follows: ..." to a machine? In a 1990 article, K. Kahn and V. Saraswat suggest that it may not even have to be text as we know it - they propose that it may be possible to express both definitions and execution using a visual notation with almost no text at all, except for strings and comments. They point out that a major use of names is to make connections, something they are not particularly well suited for! Now look at the above figures and try to imagine how you could replace all those names by arrows connecting various types of icons (as few as possible of course).

Part of my motivation in this chapter (and in this book as a whole) is to try to shake up some preconceptions about how programming has to be done! We cannot predict where the next breakthrough will be made, so it is important that we remain as open as possible to new ideas and new ways of doing things. The above concepts also suggest some desirable characteristics that these new ways should have, so we can measure whether we are moving in the right direction or away from it. As I shall suggest in the last chapter of this book, everyone should have their minds stretched regularly, and this kind of exercise is nowhere more important than in the computer business!

# Chap. XXVI: Comparison between FBP and Object-Oriented Programming

Object-Oriented Programming (abbreviated in what follows to OOP) has captured the imagination of a sizable, and influential, segment of the computer world, and understandably so, since it promises solutions to many of the problems which confront our industry. Not only does it have proven successes in the area of user interfaces, but it offers the very inviting prospect of libraries of reusable programming components which can be bought and sold on the open market. On the other hand, its very success has resulted in its being broadened until the term OOP now covers a wide spectrum of different technologies with a few basic concepts in common. The literature on the subject is confusing to the uninitiated, and the more one reads about the subject, the more different variations one encounters, all rallying behind the Object-Oriented banner. Given all the excitement, I have spent some time trying to understand what is being offered and what it can do for us. This chapter is the result of this work, and I hope that some readers will find it helpful. Naturally, one of the effects of the diversity of different views about what OOP is is that almost any comment I may make about it can be countered by someone who has a different view, but I base these observations mostly on the most widespread dialects of OOP, so I believe they have some validity.

Before I go any further, I would like to say that I believe FBP shares many characteristics with OOP, but at this point in time I hesitate to call it object-oriented, as there are certain fundamental differences of approach. However, after reading this chapter, some of my readers may conclude that any differences are basically surface differences, and that FBP is an object-oriented technology. Interestingly, Rob Strom, who developed NIL (Strom and Yemini 1983), described in the next chapter, which has strong similarities with FBP, tells me that initially his group thought it important to disassociate themselves from OOP, but recently they have come to feel that OOP is now so broad and there are so many similarities between NIL and OOP that they are now actively working with the OO community.

OOP is also another perfect example of the gap between business and academia that I talked about earlier: a lot of the interesting research work on OOP is hard to apply to business needs, while business badly needs technologies which can ease the burden of developing and maintaining application code. When academics start using payroll applications for their examples, rather than rotating squares and rectangles, we will know that we have turned a corner!

It is generally accepted that the first OOP system was Smalltalk, from Xerox PARC, although some writers identify Simula as the first OO language. It seems that many people today still consider Smalltalk the archetypal OOP language, although it is many ways a "small" implementation of the concepts, by which I mean that it is great for exploring a number of the OOP concepts, but it is not clear that its concepts scale up to large-scale business applications. I have worked with Digitalk's Smalltalk V/PM, so most of my examples will be drawn from that system. C++ is a different, and in some ways more pragmatic, approach to implementing object-oriented concepts, which is gaining increased acceptance, but, since it is a hybrid between OO concepts and a conventional HLL, its users have to contend with a more complex mental model. A different approach to hybridizing a HLL with OO concepts is Brad Cox's Objective-C (1987), which is not as well known as the other two, but also has a number of interesting concepts. All of these languages are basically control-flow oriented, and therefore suffer from the problems we have described in previous chapters. A number of workers in the OO field are starting to recognize this, and I will be describing some of their work later in the chapter.

To lay a foundation for discussing the differences and similarities between FBP and OOP, we need to talk about a few of the basic concepts of OOP for those not familiar with its concepts. The basis of Smalltalk and all OOP systems is the "object", which can be described as a semi-autonomous unit comprising both information and behaviour. OOP objects are usually selected to reflect objects in the real world, and this relationship is a major source of the appeal of OOP to application developers (as I mentioned above, it is also a characteristic of simulation languages, and also of IPs in FBP). Of course, since real world objects vary widely in size and complexity, it becomes far from trivial to decide what the objects in your universe of discourse are going to be. Just as it is in conventional programming, it is extremely important to do a good job of modelling your data before you start an OO design. The approach of Object-Oriented Analysis is somewhat different from that of conventional data modelling, but many workers in the field claim that proper modelling is even more important with OO as an error at this stage can adversely affect your whole design. This is also true of course for FBP.

One very powerful but non-obvious similarity between FBP and Smalltalk is that they both use "handles" to refer to objects (except in the case of Smalltalk integers). When I request a new instance of a class in Smalltalk, I get a set of instance variables "out there", and a handle to let me refer to it, just as we have seen happens when we create a new IP in FBP. We can then do things with this object handle, e.g. send messages to it or use it as a parameter in a message to another object. Smalltalk also looks after "garbage collection" of the object if its handle is no longer in

use - this function could easily be added to FBP, but as I said earlier we're not sure whether it's desirable.

These object handles are what allows objects to talk to each other. Once we have selected classes of objects which will represent the real world objects of interest to our application, the next requirement is that these objects be able to communicate - in short, that their behaviour be cooperative. For this function, Smalltalk uses the expressive metaphor of "message sending": Smalltalk objects are said to send messages to each other, resulting in activity on the part of the receiver, which may in turn send messages on to other objects. This also is a good fit with how we think of the real world. Unfortunately, this Smalltalk terminology is misleading if it suggests any kind of asynchronous message flow, as Smalltalk's "message sending" is purely synchronous: the sender has to wait until the receiver comes back with a reply. In today's terminology of parallel processes, the sender is "blocked" until the reply is received. This mechanism is essentially equivalent to a subroutine call, and this is in fact how it is implemented (with a subtle difference which we will discuss in the next paragraph). Smalltalk does support asynchronism by means of its fork and semaphore facilities, but the basic paradigm is synchronous and, as we have seen above, this restricts the developer in certain fundamental ways. In C++ (and also sometimes in Smalltalk) this is referred to as "method invocation", which is a more accurate description of what is really going on.

Method invocation is essentially an indirect subroutine call. The caller specifies an operation, and it is the class to which the receiver belongs which determines the actual piece of code which is executed. In both Smalltalk and C++, each such piece of code (called a "method") is part of a class and its address is not directly known to its caller. The caller specifies the function desired by naming an object (or the class itself) and the desired function, e.g. it might tell an object of class "rectangle" "rotate 90 degrees". The underlying software then uses the class information of the object to locate the actual code which is to be executed.

Although this seems very straight-forward in the classical OO examples, in practice I found it really frustrating to me as a user, because it is inherently asymmetrical. Many of these requests involve more than one object, so you have to pick one as a receiver, and pass the others (or their handles) as parameters. This means that I, as the user, was never quite sure which object should be the receiver, and sometimes a series of similar functions would flip back and forth bewilderingly. For example, when displaying a series of data objects, I had to use several different messages, some of which were sent to the medium object with the data object as parameter, and some of which were the other way around. Another example: because of this problem, Smalltalk has problems with such simple commutative operations as + and *. Smalltalk V/PM has actually implemented a facility where, if an operation fails, the system reverses it and tries again. This function is only available to primitive operations, and is not even used there consistently. You also have to be careful not to write methods which go into a closed loop! Although some OO dialects, like CLOS, select the method based on the classes of more than one

participating object, I would expect that allowing method selection to be based on several classes not only would result in even larger numbers of methods, but could result in significant management problems.

The indirect call characteristic of OOP systems does provide a degree of configurability, since it is true that the caller does not have to know the name of the subroutine which will actually be executed. In addition, since different classes can support the same function identifier (sometimes called the "selector") in different ways, you get an additional useful characteristic sometimes called "genericity", which some writers consider the basic characteristic of OOP systems (many others don't, though). However, the requester of a function does have to be able to locate the object that it wants to send the message to and also has to specify the name of the desired function, e.g. "print" or "rotate", so we still have a configurability problem, once removed, unless the process of identifying the recipient object can be completely externalized from the requester's code. Remember, to achieve full configurability we need to be able to hook together components into different patterns without modifying them in any way, which also means having an independent specification of how things are connected. This can only be done today by having "high-level" methods which specify how things are hooked together. I find it interesting that, in most of the literature, the orientation of Smalltalk is very much towards building new classes, rather than towards reuse. Applications are developed mainly by cloning old methods, with its attendant problems, rather than by using black box code. The very idea of allowing a developer to modify the behaviour of an existing class, even if only for his or her own purposes, runs counter to the reuse concepts described earlier in this book.

Two last comments about genericity: my (limited) experience is that application developers don't use it very much, and its main triumphs seem to be in the GUI area. When asked to give examples of genericity, writers on OO always seem to pick "display" and "destroy". It may be that, in business, you don't often use the same messages for different classes. For instance, at the numeric value level, subtracting a number of days from a date is quite different from subtracting a date from another date, or a date from days (you can't), so the user has to be very aware of the types of the operands. To me this means that it doesn't buy you much to be able to call them all "-". In fact, in Smalltalk you often see message names like "subtractDaysFromDate", to tell the user what types the message expects (there is no type checking at compile time, so this is particularly important). Now, if you don't make much use of genericity, all you have left is the indirect call mechanism, which should be part of any programmer's toolkit anyway!

The following three attributes seem to be present in all OOP systems to a greater or lesser extent, but they are given different weights by different writers: genericity, encapsulation and inheritance. We have already talked about genericity in OOP. Genericity is also implicit in FBP as the same IP can be sent to different processes to achieve different results (and usually is), or components can be designed to accept a narrower or wider range of possible input formats as determined by reuse considerations. For instance, a Collate could accept only two input streams,

or 'n' input streams. It could accept just one input IP format, or many, determined by descriptors as we described above.

Inheritance is claimed by some to be the major characteristic of OOP, and it is certainly an important concept, but my personal view and that of other people I have talked to is that its use should not be pushed to extremes. As long as inheritance is used to reflect the fact that things in real life can usually be grouped into classes which are subsets and supersets of other classes, it works quite well, and would in fact fit in quite well with the IP type concept that is implemented by descriptors in FBP. For instance, a file might contain records representing vehicles, which you would then "specialize" into Volkswagens, Pontiacs, etc., based on a code within the common part of the records. Some processing would then be valid for all vehicles, other processing just for Pontiacs. If a message cannot be answered by a Pontiac, it is passed up to the "vehicle" level. Generally, as you move down the class hierarchy, you add more attributes - so start off with the set of attributes common to all vehicles. When you discover that a file record represents a Pontiac, you now know how to read the remaining attributes. This concept could in fact be added quite naturally to the descriptor mechanism of FBP.

The major difficulty with classification, however, is that, as soon as you try to become more analytical about what a class really is, things start to get more confusing. What seems clean and intuitive when applied to oak and fir trees becomes less clear when you look at it more closely. In fact, the OO concept of "class" seems to involve several different concepts which are combined in different combinations in different OO implementations. For those interested in this topic, there is an interesting recent article by W. Lalonde and J. Pugh (1991) which attempts to separate out the different ideas underlying the idea of "class". To give you some flavour of this debate, consider the difference between a square and a rectangle from an OO point of view. There was a recent interesting exchange of letters on this topic in Communications of the ACM, triggered by a letter from J. Winkler in the Aug '92 issue: in a hierarchy of geometrical shapes, a square is usually defined as a rectangle with all four sides equal. From one point of view, it is therefore a subclass of rectangle. However, subclasses usually have more instance variables (attributes) than their superclasses, while a square can be completely specified using only one measurement, instead of two. As if that weren't bad enough, OO rectangles can accept messages asking them to change individual dimensions, e.g. "set height to:". If you change a rectangle's height to be the same as its width, does it change to being a square, or must you create a new intermediate class - that of "square rectangles"? The point is that this is an example of specialization by the addition of constraints. There needs to be some general mechanism to specify constraints on objects, and we also have to decide whether to use the constraint, e.g. by allowing one dimension to change the other, or just use it to detect errors on the part of the client, e.g. "violates constraint - please check dimensions". The heading on Winkler's letter is "Objectivism: 'Class' Considered Harmful" (Winkler 1992)!

While human beings naturally try to classify the world to make it easier to grasp, the real world

may resist being so classified. As a non-zoologist, I had imagined that all mammals had been neatly categorized long ago, so I was amused recently to run into this description of the difficulty zoologists encounter in trying to classify the hyrax (Krishtalka 1989): "They resemble a cross between a rhinoceros and a rodent. ...the hind limbs have three toes (rhinos), one of which ends in a long claw (rodents), the other two in hooflike nails (rhinos)...." The list goes on for a bit, then Krishtalka writes: "Such a smorgasbord of physical traits earned a dyspeptic taxonomy.... Recent opinion is divided between a horse-rhino-hyrax evolutionary connection and a sea cow-elephant-hyrax linkage." While this kind of confusion can actually be amusing, our tendency to make snap classifications and then act as if they were the whole truth may actually be harmful, either to ourselves or to others: while everyone today with a reasonable education knows that whales are mammals, not fish, the old mental association may be what allows officials to refer to "harvesting" whales. We can certainly talk about "harvesting" herring, but we don't talk this way about tigers, cattle, butterflies or people, so why whales? If you are interested in this area of linguistics, you should take a look at the work of the linguist B.L. Whorf (1956), alluded to elsewhere in this book, on how the words we use affect our actions.

As we move into the world of business programming, we run into situations where class hierarchies may seem very natural at first sight, but in fact are really not appropriate. For instance, it might seem natural to assign a bank account object to one of a set of account type classes: SAVING, CHEQUING or COMMERCIAL. This way, a deposit could be sent to an account and automatically cause the right piece of code to be invoked as a method. While this seems quite attractive at first, in fact, at best this would result in a number of very similar methods which would have to be separately managed and maintained. At worst, it could make it very difficult to develop new, hybrid offerings, such as a chequing account which offers daily interest. Banks have found that it is better to make this kind of processing "feature-oriented" - one should decide what are the atomic features of an account, such as interest-bearing or not, bankbook vs. statement, cheques to be returned or not, and then implement them under switch control to produce the various types of account processing. Hendler gives a somewhat similar example (1986), using professions. He points out that while professions are often used as examples of classes, they may not be mutually exclusive - a person might be both a professor and a doctor - so a person may carry attributes which relate to both of these professions. Mixed classes provide a possible solution, but this technique has problems as well. In FBP, the "tree" technique seems a natural way to implement this kind of thing (see Chapter 12, on Trees), as the data associated with each profession can be held in separate IPs attached to the IP for the person.

In spite of what I have said above, I do believe that one of the most important contributions OO has made towards changing the way application design is done is that it has moved data to the foreground. Programmers coming to FBP from conventional programming have to undergo precisely the same paradigm shift: from concentrating on process to concentrating on data. Typically, in FBP, as we have seen in the foregoing chapters, we design the IPs and IP streams

first and then decide what processes are needed to convert between the different data streams. In OO you have to decide on the object classes, and then decide what messages each class should be able to respond to.

For many OO enthusiasts it is this concept of "encapsulation" which is the central concept of OO. In fact, this is not a new concept at all (one of Dijkstra's famous remarks was that programs should be "like pearls"), and Parnas wrote one of the seminal articles on encapsulation in the early 70s (Parnas 1972). Encapsulation simply means the idea of having the vulnerable insides of something protected by a protective outer coating, sort of like a soft-centred candy (or a turtle). This is obviously a good design principle, and the reader will notice that FBP components in fact have this characteristic, as they are free to decide what IPs they will accept into themselves, and can do more or less validation of their input data, depending on how reliable their designers judge their data to be. Encapsulation can also be implemented at the network level, by having outer processes protect inner ones, or by inserting transformer processes into the network. This is a better solution than building the validation into every component, as the processing component can just provide the basic function, and the designer can request more or less validation by adding or removing editing processes. In OO, an object is encapsulated together with all of its methods, which involves predicting all the services that an object may ever be requested to perform. This, however, is very hard to do, and may result in a never-ending stream of requests for enhancements as new requirements come up. How can one predict all the functions that, say, steel might be used for? Remember Wayne Stevens' story about an airline attendant using a hearing set to tie back a curtain (recounted elsewhere in the book)!

In FBP, we always encapsulate processes and can also encapsulate IPs if desired - the former occurs automatically as nobody has access to the internals of a process except the supplier: users can only know its inputs, outputs, parametrization and some behavioral aspects, such as what it does when it sees a closed output port. As far as protecting IPs is concerned, a number of techniques are available, as required by the designer, and it is quite possible to have IPs whose structure is never seen by application code. However, FBP does not insist that we predict all the processes that will ever handle a particular IP type. Rather, the emphasis is on deciding which IP types a given process will accept or generate. Instead of having to predict all the uses that steel might be put to, we only have to decide which materials we can build a bridge out of. The latter seems a much more manageable problem!

Because Smalltalk's "message sending" terminology sounds like data flow, it is often thought that OO should be relevant to distributed systems design, but in fact, as Gelernter and Carriero point out in an article analyzing the differences between their Linda (described in the next chapter) and OO (Carriero and Gelernter 1989), it is actually irrelevant to it. In fact, as they say, a truly distributed message passing system has to be built on top of an OO system, just as it does on top of a conventional subroutine-based approach. Here is a quote from a paper by another of the gurus of this area, Barbara Liskov, and her coworkers: "We conclude that the combination of

synchronous communication with static process structure imposes complex and indirect solutions, and therefore that it is poorly suited for applications such as distributed programs in which concurrency is important" (Liskov et al. 1986). It is interesting that "basic" FBP occupies the "asynchronous, static" quadrant of Figure 2-1 of this article, while the addition of dynamic subnets moves FBP into the "asynchronous, dynamic" quadrant, which the authors of this article say is unoccupied to the best of their knowledge. Interestingly, they go on to say, "Although such languages may exist, this combination appears to provide an embarassment of riches not needed for expressive power." Our experience, on the contrary, is that adding a dynamic capability to asynchronous communication can be extremely productive!

Most OO implementations are synchronous, so the basic primitive is the indirect call through the class. As I said elsewhere in this book, our experience with FBP tells us that the subroutine call is not the best foundation on which to build business applications. A "call" can in fact be simulated very nicely by issuing an FBP "send" followed by a "receive". This will have the effect of suspending the requester on the "receive" until the downstream process returns an answer, just as a "call" suspends the caller. Gelernter and Carriero make the same point and go still further in the above-mentioned article:

> "In our experience, processes in a parallel program usually don't care what happens to their data, and when they don't, it is more efficient and *conceptually more apt* [my italics] to use an asynchronous operation like Linda's "out" than a synchronous procedure call.... It's trivial, in Linda, [or FBP] to implement a synchronous remote-procedure-call-like operation in terms of "out" and "in" [FBP "send" and "receive"]. There is no reason we know of, however, to base an entire parallel language on this one easily programmed but not crucially important special case."

A call which spans multiple machines is sometimes called Remote Procedure Call (RPC), and a number of the people working on distributed systems have pointed out the inappropriateness (as well as poor performance) of this algorithm when building complex distributed systems. K. Kahn and M. Miller (1988) point out the problems of basing a design for distributed systems on RPC. They also stress the desirability of having a single paradigm which scales up from tightly coordinated processes within a single processor to largely independent cooperating processes, perhaps on different machines.

FBP and Linda (we will talk about Linda in more detail in the next chapter) are fundamentally asynchronous, whereas Smalltalk-style OO is synchronous. The real difference here is that, although the methods of an object are the only routines which can have access to the object's internal data, *when* these methods are to be executed is determined by other objects, whose methods in turn are driven by other objects, and so on. While such synchronous objects show autonomy of data and behaviour, they do not have autonomy of control. As such, I feel that synchronous OO objects are more similar to FBP IPs than they are to FBP processes. In a

Smalltalk (not counting "fork") or C++ application, there is actually only one process. This can lead to counter-intuitive solutions. For example, in a recent book about C++ (Swan 1991), in an example involving a simulation of people using elevators, the class Building (which is really running the whole simulation) apparently has to be treated as a subclass of the class Action. The problem, of course, is that there is only one process, external to all the objects, which is basically "Run the simulation".

If you cast your mind back to the Telegram problem described in Chapter 8 [this is the problem where text is read in from a file and must be written out in records of a different size, without breaking individual words], you will remember that the conventional programming solution required several of the routines to be invoked repeatedly using handles to maintain continuity. This solution maps very nicely onto an OO "collaboration diagram" which changes subroutine calls into "message sends" and "replies" between objects (remember the caveat about what "message sending" actually means). Here is basically Figure 8.10, recast into OO terms (I have created 4 "stream" objects: 2 word streams and 2 I/O streams):

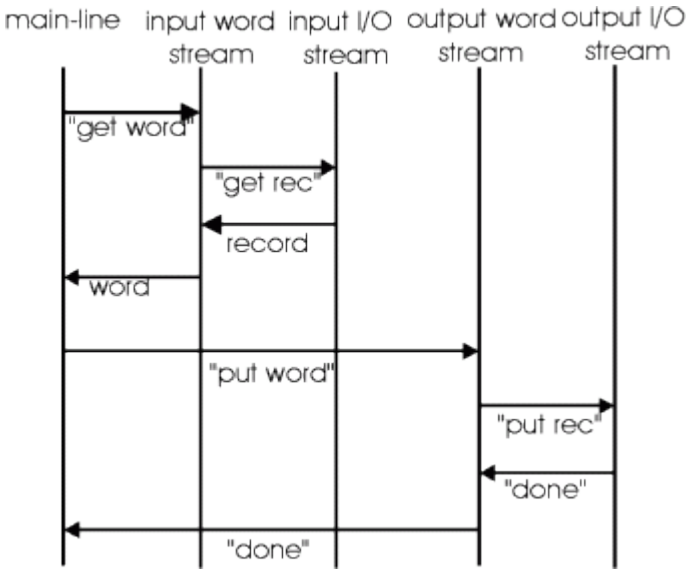



Figure 26.1

While this solves the problem of subroutines which have to maintain continuity between successive invocations (the infrastructure maintains the continuity), this is still a purely synchronous solution. Now let's show an FBP solution to this problem (from Figure 8.2):

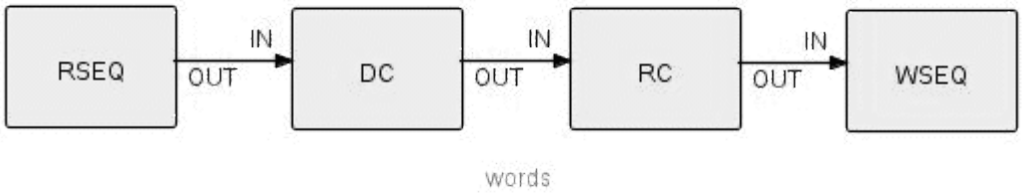



Figure 26.2

[In case you didn't figure it out... in this diagram, RSEQ means "Read Sequential", WSEQ means "Write Sequential", DC is "DeCompose" and RC is "ReCompose".]

Not only is this much easier to grasp intuitively, but it uses reusable components, plus it is very obvious how the function can be extended if the designer ever needs to.

I have tried to show in the earlier chapters that asynchronism is liberating, and I hope I have managed to convey some feeling for its power. In fact, many of the leading thinkers in OO also realize the need to add asynchronism to OO to relax the tight constraints imposed by the von Neumann machine. Many of today's advanced machine designs in fact require these asynchronous design concepts (for a survey, see a recent (1990) article by Gul Agha). Agha uses the term Concurrent OOP (COOP) to describe his approach, which combines the concept of "actors" with OO. Another term you may run into is "active objects", which *act*, as opposed to "passive objects", which *are acted on*. In modern user interfaces we already see functions which behave much like active objects, e.g. printers (for printing objects), shredders (for destroying objects), and so on. You just drag and drop the icon (small graphical symbol) of an object, e.g. a file, onto a shredder icon - this is like pressing the start button on a trash compactor. Before it starts, however, the shredder politely asks you if you really want to do this. This is another characteristic of this kind of object: they can independently gather information for themselves. Once the shredder or printer has started, the user is then free to attend to other things.

Another researcher who feels that basic OO has to be broadened by the addition of asynchronism is de Champeaux at Hewlett-Packard. He is looking at the use of a trigger-based model for inter-object communication. Here is a quote from an article about OO research directions [that appeared] in the Communications of the ACM: "This model [where the sender is suspended until the receiver sends the result back] is not rich enough to describe all the causal connections between objects an analyst needs to model." (Wirfs-Brock and Johnson 1990) Interestingly, de Champeaux's work suggests that a richer interaction model than (data-less) triggers is necessary. One of the forms he is looking at is "send-no-wait" (where data and the trigger are simultaneously transmitted). One of the chapters in a recent book (Kim and Lochovsky 1989), is

called "Concurrent Object-Oriented Programming Languages", written by C. Tomlinson and M. Scheevel, and provides an excellent survey of this new thinking about ways to combine OO with concurrency. Again, Brad Cox, who is the inventor of Objective-C and one of the acknowledged gurus of OO, has come to feel that OO alone is not adequate for building large systems. He came to the conclusion that FBP concepts should be implemented on top of Objective-C, and then could be used as building blocks for applications. Using a hardware analogy, he refers to Objective-C as "gate-level", and FBP as "chip-level". He had in fact already started experimenting with processes and data flows independently when he found out about our work and contacted me. He has advanced the idea that the time is ripe for a "Software Industrial Revolution", much like the previous Industrial Revolution which has so totally transformed the world we live in over the past couple of centuries. Like Brad, I believe many of the tools for this revolution are already in place, but many writers have remarked on the enormous inertia of the software industry - this has always struck me as ironic, given the incredible rate of change in the rest of the computer industry.

Let us try to show with an example some other differences between synchronous OO and FBP. It is quite hard to find an example which lets one compare the two technologies fairly, as the synchronous orientation of most OO work means that their examples tend to be synchronous as well. However, given that batch programs are not going to go away (in fact, there are good theoretical reasons why they never will), I will use as an example Brad Cox's example of calculating the total weight of a collection of objects in a container: say, pens and pencils in a pencil holder. While being totally procedural, it is an example of the "small batch" logic which is also handled very well by FBP.

The basic design mechanism in this kind of procedure is the collaboration diagram, of which we gave an example above. At any point in time we will have three objects: a requester, a container and an object within it. The interaction is then as follows:

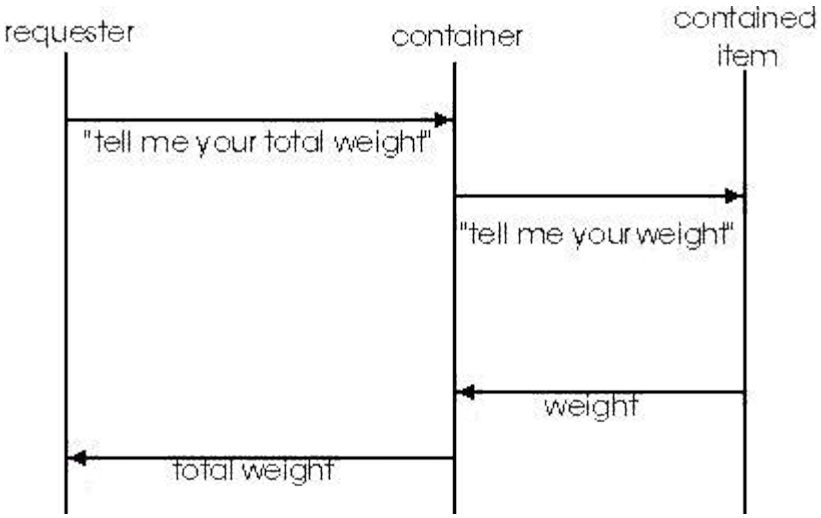



Figure 26.3

I can still remember my feeling of dismay at seeing the right-to-left, returning flows in the above diagram - these mark this diagram as being call logic, rather than flow logic. Every pair of lines represents a client-server relationship - OO people call this "delegation", but it is not delegation as humans practise it. Rather it is like standing over someone, and saying, "Now type this line; now type this line". In fact, client-server relationships make much more sense when the relationship is asynchronous, allowing the client to go about his/her business while the server is doing its thing. Human beings don't see any point in delegating work to others unless it frees them up to do something else. This kind of interaction is also not "cooperative" as FBP understands the word. In FBP all the processes are at the same level - there is no boss. In the above diagram, while there may well be situations where either object can drive the other, one of the objects still has to be the driver (as long as one stays with passive objects only). There is very definitely a boss, and it is the object at the far left.

The logic for the "compute total weight" method of the Container object is a loop which steps through its contained items. It could be described by the following pseudocode:

```
set total weight to tare weight of self
get first item 'a' in (attached to) self
do as long as get is successful
        send message to 'a' to get its weight
        add result to total weight
        get next item 'a' in (attached to) self
enddo
return total weight
```

Figure 26.4

This method needs functions to "get first" item and "get next" item within the container. These functions would return an item's handle, plus an indication of whether the request was successful. Once an object has been located, the container can send messages to it.

Although the same general logic can step through a variety of different collection structures (you basically need different method subroutines for each collection type), there is a basic assumption in the above logic, namely that all the items in the collection are available at the same time. As we have seen in previous chapters, this is not really necessary (since only one item is handled at a time), and may not even be possible. In addition, our experience with FBP tells us that this function should really be designed as a reusable component which is usable as is, in object code form, without needing any modification or recompiling. Most programming systems tend to present their ideas from the standpoint of someone writing new code, whereas FBP experience tells us that people don't want to write new code if they can get something off the shelf which does the desired job. Key to this (and also to being able to distribute such systems, now or later) is the requirement to avoid calls - as we pointed out above, the subroutine call mechanism forces tight coupling, whereas we want the data being generated by a procedure to go onwards, not back. The only way I know of to achieve all these goals is to design the function as a stand-alone function which uses ports to communicate with its neighbours. This results in a component with the following shape (you will recognize this as a "reference" type of component):


Figure 26.5

This component accepts a stream or multiple substreams of IPs and generates one IP containing the total weight (or one per substream). Since the container has weight (its tare weight), let's provide it as the first IP of the (sub)stream. This diagram is really a fragment of an enhanced collaboration diagram connecting multiple processes with one-way flows instead of a single process talking to itself with two-way flows!

The logic of the above process can be represented by the following pseudocode (which should be familiar from earlier chapters):

```
create IP to contain total weight
receive from port IN using handle 'a'
```

```
set total weight (in weight IP) to (tare) weight of 'a'
send a to port OUT if connected
        else drop 'a'
receive from port IN using handle 'a'
do as long as receive is successful
        add 'a's weight to total weight
        send a to port OUT if connected
                else drop 'a'
        receive from port IN using handle 'a'
enddo
send weight IP to WEIGHT port
```

Figure 26.6

Not surprisingly, it has the same general structure as the method pseudocode shown above, but there are certain key differences. The logic shown above can process any data stream for which "a's weight" is defined for each IP in the stream. Incoming IPs are passed on to OUT (if it is connected), and the weight goes in an IP of its own to the port called WEIGHT. Remember Gelernter and Carriero's remark that "processes in a parallel program usually don't care what happens to their data." Since "receive" and "send" can be suspended until data or queue slots, respectively, are available, this routine works even though not all IPs are in storage at the same time. We now have a portable component which can compute the total weight of any stream of IPs for which "weight" is defined.

In addition, in OO, this function has to be a method contained in any collection class for which you might need to perform this function, whereas in FBP, once this function has been built, we can use it (just by referencing it in a network) on any data stream which conforms to certain conventions, without having to modify the definitions of any of the classes involved. As we said above, "a's weight" has to be defined for each IP in the stream. However, we can even parametrize the attribute name, so we can use the same object code to get a "total x" from all data streams for which "x" is defined. Instead of having a myriad small, special purpose, methods for every different class in the system, we arrive at robust, flexible, functions which are highly portable, e.g. (in this case) a function to determine the "total x" for any x which is defined for the IPs in the stream. In fact, we could even generalize this function still more: you could use a very similar structure to get the maximum or minimum weight of all the contained items. Of course, in this case "tare weight" would not be too relevant, but whether we are adding the contained item weights together or taking their maximum could also be provided as a parameter to our component.

To recast this function in OO terms, we would need to provide some kind of configurability. Assuming that we follow OO and make the "send" and "receive" functions "messages" to objects, then the objects "send" and "receive" talk to could actually belong to any of the following object types: other processes, streams, connections or ports. The only one of these which would not

reduce the component's portability would be ports, unless the names of the other objects were passed in as parameters to the process. However, the latter alternative would clutter up the component's parameters with connection information. Port names would be the way a process identifies its "own" ports, and could be instantiated by a function very like THREADS's "define ports" service (see Appendix), which would accept port names and return an object handle. The "compute total weight" process logic can then send messages to its ports, to do receiving or sending, using normal OO syntax. We will of course need some kind of Driver or "connection" engine to connect our processes together using these ports together with a list of connections, to give us our desired configurable modularity, but this is outside the component logic.

The last thing we need to decide before we can recast our component in OO terms is how to determine the "x" of a given IP. There is no problem conceptually with making this a normal OO "message", as "get first" and "get next" will have returned a handle to an IP, which we can then send messages to. However, how should we name the function of obtaining "x" for the subject IP? Based on FBP experience, I suggest that the simplest technique is to have a generic "get" and "set" function which accepts the field name as a parameter (or even multiple field names to reduce the overhead). OO purists may feel that it is better to have multiple "get" and "set" methods - one of each per field - but this leads to a very large number of almost identical method subroutines.

Whether we implement attributes as OO methods or by using subroutines hung off the descriptor, we can do other things than just retrieve real data. We could also use these techniques to make sure related field values are kept in step (data integrity), or to support "virtual fields" (fields which are computed as they are needed). Thus a request for the number of children of Joe could scan Joe's attached IPs (where Joe is a "tree" structure) and return the result. The requester need not know whether the field is real or virtual. Such a mechanism would let the data designer either go for computation speed at the expense of having to maintain duplicate data, or, on the other hand, go for highly consistent data at some cost in performance. Another capability is what is sometimes referred to by the name "daemons": this involves the ability to automatically trigger events when a field value changes or passes some maximum or minimum. When combined with asynchronism, this could be a very powerful structuring tool for building business applications.

One important topic I want to address is the issue of granularity. All discussions of both OO and FBP eventually come up against this topic: how "big" should FBP processes and OO classes be? The lower end of FBP granularity is determined by the fact that IPs normally have multiple fields and often represent objects in the outside world. You could chop a business IP up into one IP per field, but then you would have to pay a lot of overhead to recombine it to write it on a file, data base, screen or report. The granularity of a language like C++ is approximately the same as that of FBP: objects very often correspond to file records. Many Smalltalk objects are at this level, but Smalltalk also makes much smaller pieces of data objects, such as amounts of money. Even integers are treated as objects, although the implementation for these is a little different for

performance reasons. Smalltalk is able to be much more granular than FBP, but only because of its synchronous nature - "attribute objects" stay together because there is no tendency for them to drift apart in time. I believe the granularity of asynchronous systems will naturally tend to be coarser, unless counteracted by expensive (re)synchronization mechanisms.

As I talked to people about OO, however, I came to realize that there is one area which OO (Smalltalk anyway) does address which is absolutely unique to it, and in fact takes care of a problem which has been worrying me for several years: the need to be able to prevent illegal operations on data fields, e.g. to stop currency values from being multiplied together, or dates from being added (this was referred to as a problem above). However, this ability can only be taken advantage of if one does everything in OO, rather than combining it with existing HLL facilities. As we said before, the vast majority of HLLs are based on mathematical ideas of data, and treat numeric fields as dimensionless. They thus cannot provide intelligent handling of most of the numeric values one runs into in business applications - these are either dimensioned numeric quantities (like money or weight) or aren't even in the pure numeric domain (e.g. dates). In HLLs, all these types of data are compressed into a single numeric format which is indistinguishable from other numeric values. In Smalltalk all accesses to data values are via methods, so we are not forced to throw away our knowledge about what fields really represent. A "multiply" operation can be resolved to one or more methods which know how (or whether) to do the appropriate operation on the fields involved. Hybrid approaches lack this power, and any attempt to combine OO with conventional HLLs in the same process vitiates this checking ability. Some of the newer HLLs provide similar forms of checking, e.g. Ada, so a possible solution is to restrict the "business logic" parts of an application to using an OO language or one of these newer compilers.

If all of the above seems unduly negative, it is mainly that I feel a need to put OO into a proper perspective. OO is a simple technique, whose main importance is that it has started a sea-change in the way programmers think. It is definitely a step on the way, but my 20 years of experience with FBP tell me that, if we stop at this point, eventually frustration on the part of programmers is going to win out over the initial excitement. While I recognize that learning and using OO is an important learning experience, it is a hard way to learn, and, in its present form, an expensive way as well, as without configurable modularity the result will only be marginally more maintainable code. Configurable modularity can be added to OO, as can multithreading, just as they can be to conventional programming, and it is exactly the combination of these which starts to open up interesting possibilities.

In an FBP environment, it is possible and, I believe, highly desirable to mix processes running different languages, some OO and some non-OO. For instance, one process might be running a pure OO language, another one COBOL, another Assembler, and so on. Such a mixture would require that IP layouts become a public interface between processes, but note that this public interface should preferably be IPs associated with their descriptions. We now have a natural role

for IP descriptors: to allow us to retain the IP attributes' domain information, which could be exploited by OO, across processes which do not use this information (e.g. ones written in existing HLLs). OO processes could in fact be protected by interface processes which turn IPs into some format acceptable to the OO language chosen. Such a combination of processes could even be packaged as a composite component, giving what seems to me to be the best of all worlds!

Wayne Stevens suggested a few years ago that objects might split very naturally into "process-like" and "data-like" objects, where, essentially, process-like objects would correspond to FBP processes, and data-like objects to IPs. In the phrasing I used above, data-like objects are passive, while process-like objects are active. Process-like (active) objects are able to act without necessarily always having to be triggered by an event external to them. In traditional OO systems, all objects are passive, and the whole assemblage is triggered by one (non-object) trigger that starts the whole thing running. This approach is obviously going to suffer from the same difficulties as traditional hierarchic non-FBP programs. If, instead, some of the set of objects can be active, we can start to capitalize on our experience with FBP. You will also notice that FBP processes have their own internal working storage, which looks very much like an OO object's "instance variables". Having process (active) objects and data (passive) objects looks like a very good way to combine the strengths of these two complementary technologies. In fact, with the appropriate infrastructure, different objects can be coded in different languages. Since, as we have shown above, one of the basic reuse mechanisms in FBP is the external definition of connections, we could also add a "driver" and "network" object: this would be an active object using the network definition as reference data.

My belief in the potential for combining the strengths of these two approaches is bolstered by the fact that, in FBP, we have actually built processes which behaved much like objects, and also by the observation that traditional OO applications often have objects that should really be separate processes. An example of the former is the List Manager which I described in Chapter 21. This component managed multiple lists arranged in levels. An example of the converse is a Data Base Manager object, which accepts requests to get, insert or delete data. As we have seen in earlier chapters, this is better implemented as an asynchronous process, rather than driven synchronously by other objects. Our List Manager suffered from the problem that it was very sensitive to the exact sequence of requests, which made it hard to use in a highly asynchronous environment. It would have been better implemented by externalizing the lists as FBP trees, so that one or more processes could work on these trees asynchronously. In other words, objects with overly complex internal data will be hard to use when we start to have more processes running in parallel. I expect this will apply even more noticeably as we start to distribute logic across multiple processors.

A number of writers in the OO field have started to explore the possibilities of active objects. In Chapter 1 of a collection of essays compiled by Kim and Lochovsky (1989), O. Nierstrasz makes the point that systems which mix active and passive objects would not be uniform, and this seems

a valid point. However, one possible solution is offered by a system called Emerald (Black et al. 1986), which was designed for implementing highly distributed systems, and which maintains uniformity across all its objects by allowing every object to have a single process in addition to methods. Not all objects may activate their processes, but the potential exists for them to do so. This suggests a very workable generic structure for all the objects in a combined FBP/OO hybrid.

OO research and development seems to have entered a stage of accelerated growth, and it is very exciting to me that some of the newer work bears an uncanny resemblance to FBP! A dichotomy seems to be developing between the synchronous and asynchronous OO approaches, just like the one we have seen in non-OO. A number of OO researchers believe it is the asynchronous approaches which will turn out to have the most to contribute to the programming art in the long run. More and more of these people are discovering the power of active processes to broaden OO and make it better match the real world. Tsichritzis et al. (1987) have used the concept of active objects in knowledge processing - they call their objects KNOs (KNowledge Objects). KNOs can also have a complex structure, analogously to FBP composite components. Still more recently, Nierstrasz, Gibbs and Tsichritzis have collaborated on another paper on Component-Oriented Software Development (1992) which approaches FBP even more closely, but is still based solidly on traditional OO concepts. While their terminology is different from that of FBP, many close correspondences between the two can be established. They use the term "script" to mean "a set of software components with compatible input and output ports connected". While scripts can be data flow or object-oriented, the data flow version corresponds closely with FBP networks. "Scripting" means the construction of scripts, so the term "visual scripting" is defined as "the interactive construction of applications from prepackaged, plug-compatible software components by direct manipulation and graphical editing". In their article they talk about reusable components, ports, SACs (scripts as components) and visual scripting - all ideas that have direct counterparts in FBP. The same article goes on to describe an application of these concepts to multimedia called the "visual museum". "Media objects" (which are active objects, i.e. processes) work on "media values", which are

> "...temporal sequences.... Media objects produce, consume and transform media values.... Media objects, in turn, are grouped into multimedia objects by specifying the flow of values from one object to another - we call this flow composition.... flow composition actually produces applications...".

Another remark in the same paper that I found interesting was,

> "One benefit of flow-based composition is that new functionality can be added, or removed, by simple modifications to the script".

In the conclusion of their article they stress a number of the points I have made elsewhere in this book: the difficulty of generalizing to create good reusable components, and the economic and

project planning impediments to producing such components. This equation of objects = processes seems to be gaining acceptance: the article describing A'UM (Yoshida and Chikayama 1988) matter-of-factly describes the system as consisting of "streams" and "objects" (for more on this interesting system see the next chapter). They then go on to say that of course streams can be objects also - which seems very close to what we were saying earlier about the possibility of treating IPs as objects.

From an FBP point of view, the concept which I feel is missing from traditional OO (not from the work on active objects) is the concept of "transformer" processes (many of the media objects described in the above-mentioned paper are explicit transformers). As Nan Shu (1985) has pointed out, much of business programming has to do with transforming data from one format to another. The paradigm of passing a stream of data packets through a transforming process seems to fit very naturally with this image, but this does not seem to fit well with traditional OO. Since the traditional OO paradigm specifies that only the methods of a class should know an object's internal state (which is presumably held in some canonical form), this would seem to imply that transformations are only of interest at the boundaries of an application (when one is bringing in or outputting "foreign" files, reports or screen data). In practice, as businesses build bridges between more and more of their applications, we will spend quite a lot of time converting data between different formats. Some of these applications will be vendor-provided, so the users will have even less control over their data formats. If these applications are OO, how will their classes be merged with the corresponding classes of the users? Data conversions will also be required for many of the common data transportation techniques - this will become more and more important as we move towards distributed systems. Thus, you might decide to convert binary data into character format to simplify transportation between PCs and hosts. Descriptor-driven transformers in an FBP environment will provide a simpler paradigm and will help to make all this run smoothly. Interestingly, in the paper I was talking about above, the authors also feel that multi-media applications will require a wide range of transformations of media values into different forms, depending on the various uses they need to be put to.

I found it significant that many of the media objects in the paper on Component-Oriented Software Development have names which are verbs, rather than nouns, e.g. render, interpret, provide (in FBP, processes are usually verbs, while IPs are nouns, e.g. customer, account, department). Traditional OO essentially works with nouns, with the verbs relegated to the methods - this has the effect that, for instance, to record the fact that a student has taken a course, you express this by having the student send messages to the course, or the course to the student. From an FBP viewpoint, it seems more natural to handle this with a process which transforms the student in well-defined ways. So an OO approach which is perhaps closer to FBP's way of thinking would be to send both student and course to a separate "attacher" object, which has the ability to associate students and courses. This object would be an active version of the general category of object called "dictionaries" in Smalltalk. These are two different, not necessarily

incompatible, viewpoints.

It seems quite probable that powerful hybrid FBP/OO systems will be built within the next few years [still waiting! JPM]. Such a combination may well have some interesting and unexpected capabilities. Unfortunately, one problem the OO part of it is almost certain to have (until the new highly parallel hardwares appear on the market) is that of performance: for every field in every information packet accessed by an OO method, you need a call, plus logic to locate the method. In FBP, you can totally hide the layout of your IPs, and you can control your performance - accessing a field may be as little as one instruction, or it may be a complex function: it is your choice. Incidentally, the performance issue makes it doubly important to be able to select which part of an application is to be written in an OO language, and which in a conventional language.

The performance problem and the many arguments in favour of the asynchronous process approach to applications design lead me to believe that, if OO starts to be used for business production programming, it will be the concept of active objects (process objects) which will turn out to be more productive for OO than the original indirect call mechanism. To me active objects seem to be a natural evolution of OO in a direction which will eventually converge with FBP. If one can say that conventional OO (static objects) provide autonomy of data and autonomy of logic, then active objects also provide autonomy of control. Without the last, I believe it is not possible to build the systems we need in the future.

After I wrote the above, I came across the following comment by C. Ellis and S. Gibbs in Kim and Lochovsky (1989):

> "In the future, as we move beyond object-oriented programming, it is likely that one of the useful enduring concepts is that of 'active objects'."

I agree absolutely! Over the next few years I believe that we will see more and more OO proponents talking about the advantages of active objects. I applaud this as it will expose FBP to a wider audience, but it may leave the programming public with the erroneous impression that FBP is a rather complex extension of the basic OO set of concepts. In reality, as I have shown in the foregoing pages, FBP can in fact be implemented with quite simple software and yet yield great gains in productivity, while OO can only do this if it incorporates advanced concepts which seem to be converging with FBP. To quote Ellis and Gibbs again on this matter:

> "Although we foresee that object-oriented programming, as we know it today, is close to its deathbed, *we foresee tremendous possibilities in the future of active object systems* [my italics].... Vive l'objet actif."

The celebration may perhaps be premature, but, if you have read this far, you will have some idea why so many of us feel so excited about these concepts!

Before leaving this topic, I would like to make a last point which I consider vitally important: any

evaluation of a programming technology must be done in the context of building and maintaining real business applications. There are only three reasons I am aware of for adopting a new technology: performance, productivity and maintainability. Even if a new technology allows us to get applications working faster or sooner, if it does not result in significant gains in maintainability, it may not be worth the effort. As I said earlier, we have to try out potential tools on the day-to-day concerns of business programmers, rather than on artificial, theoretical puzzles, no matter how intellectually stimulating they may be! When we have an OO application which processes every one of over 5,000,000 accounts 5 days a week at a bank, is easy to maintain and does not use prohibitive amounts of resources, we will truly be able to say that OO has come of age!

# Chap. XXVII: Related Concepts and Systems

*[This is a survey of the stage the "related" work was at up until 1994. Of course, many other systems have appeared since then, and indeed there are now companies expressly founded to capitalize on these concepts and similar ones. Some of the systems described in this chapter are still alive and doing well – but maybe this chapter is chiefly of interest as a historical record.]*

This chapter provides an overview of some of the work going on in universities and business which has concepts in common with FBP. My main problem with trying to give a good overview is that new software is appearing faster than I can keep pace with it! Every time I open a magazine I see work that has resonances with FBP, which I therefore resolve to read. Then I have to follow up their bibliography, which results in more to read, and so on!

The above is a roundabout way of apologizing to all those researchers and developers out there: if I have omitted your favourite piece of software or research effort from what follows, please accept this apology and send me information about it for my ever-growing files!

In a similar vein, I will summarize what I perceive as the salient features of each item, and I may have misunderstood their thrust. In most cases I have had no contact with the authors, and I will inevitably tend to look at their work through FBP-coloured glasses. Again, please contact me to exchange ideas or to throw brickbats!

For the above reasons, what follows is probably only a fraction of what is out there - I just hope I have included enough to pique your interest.

FBP seems to me to stand on three main "legs":

1. asynchronous processes
2. data packets (IPs) with a lifetime of their own
3. external definition of connections

You will find one or two of these in many systems, but it is the combination of all three which makes FBP unique. And even when you find all three of these items present in a system, you will

find that they have been used to address a particular area or problem, such as distributed systems. I believe many of their developers have not yet understood the potential of this combination for improving *all* application development.

The first system listed below is not even an application development system, but a simulation system. In the preceding chapters we have often remarked that FBP allows simulations to be "grown" into full-scale applications, and the developers of GPSS saw this as a very intriguing possibility.

**GPSS (General Purpose Simulation System)**

This is a simulation system developed by Geoffrey Gordon and his team at IBM in the early 60s. I am including it because it shares all three characteristics above (the data packets were called "transactions"), and it was a very successful system for simulations. I also include it because it profoundly influenced my thinking about how to do application development. Gordon had hopes of extending it to run as an operating system control program, and did quite a bit of work in this area. Many people have also dreamed that it should eventually be possible to "grow" an application from its simulation, and in fact this was tried out in 1963 (30 years ago!) using GPSS, and was quite successful. Using FBP this process is now very straight-forward.

**MASCOT (A Modular Approach to Software Construction, Operation and Test) (K. Jackson and H. Simpson 1975)**

This system, built by Ken Jackson and his team at the Royal Radar Establishment in England, shares all three of the above characteristics also. I believe it has become quite widely used within the military establishment in England. Jackson's motivations for developing MASCOT were similar to ours, plus a desire to make interrupt-driven software resemble the remainder of the computer load, as a lot of their signals work involves handling asynchronous interrupts. In MASCOT an IP is called a "message", as in much of the work listed below, and a connection a "channel". The basic MASCOT unit of software is a subnet, called a "subsystem", and is drawn graphically, and then converted into a textual representation. MASCOT's services are somewhat lower level than FBP's, but its channel management is very similar overall. Here's a quote on how well the developers feel they met their objectives:

> " ... the overall philosophy of MASCOT operation ... is to process data when it is available, to explicitly wait when no data is available and to pass on a stimulus to adjacent data users when data of interest is passed on.... Further, the use of the control queue within the interrupt handling software enables the expression of interrupt handling software to be unusually straightforward. The elimination of polling and searching within the kernel is complete...
>
> "Conclusion

> "MASCOT provides a very basic yet sound machine-independent kernel to produce a suitable environment for real-time programming.... Finally MASCOT looks to be a promising base upon which to build the high integrity systems which are the subject of current research."

You'll note the term "message": this reflects the idea that processes send messages to each other. Some of FBP's IPs are definitely messages, but our experience is that the use of messages as the main communication vehicle implies a finer process granularity than the level we have found to be most productive.

**CHIEF**

This seems a classic case of parallel development, as this work must have been going on at about the same time as ours was going on in Canada, but was totally independent of it. J. Boukens and F. Deckers, working for the Shell Company in Holland, came up with a system which shows a number of really remarkable similarities to FBP. In fact, the main difference is that their diagrams are vertical, whereas FBP's are horizontal! Another difference is that they allow multiple consumers for a connection (causing the records to be automatically copied), while in FBP we do not allow this. On the other hand, FBP dialects usually have an off-the-shelf Replicate component, which has the same effect. CHIEF's ports are numbered, as in AMPS and DFDM. To convert one of their diagrams into machinable text, they list the connections (called buffers), naming their producers and consumers. A list of such specifications is called a "cooperation". To program individual processes they developed their own interpretive language, whose execution was integrated with the multithreading driver. There is also a discussion of deadlocks.

When reading their paper, and especially the discussion after their presentation, I felt very strongly their excitement over these concepts, and perhaps some frustration that others weren't feeling this excitement too...

Actually, speaking of parallel evolution, after I had been working on these concepts for a few years, I came across two papers dating back quite a few years: one was written in 1967 (32 years ago!) by two people at the Rome Air Development Center, E. Morenoff and J.B. McLean, entitled *"Inter-program Communications, Program String Structures and Buffer Files" (1967)*, describing a program interconnect structure based on the control of data flow via intermediate queues. For a single processor, queues could be maintained in main memory. For a distributed processor structure, queues could be held on disk.

A few years later (1971) R.B. Balzer published a paper on what he called *PORTS*, describing work done at RAND where modules could be shielded from other modules by utilizing software commands such as "connect", "disconnect", "send" and "receive". It is interesting that this paper is one of the most cited papers in the field, but, according to private communications with the

author, little came of it.

**STREMA**

Another related system I came across in my reading was developed by Ian Clark of IBM UK (1976). It is described as a graphic conversational language for specifying and running application processes. STREMA uses a relational model and is intended to allow relational data to be treated in a uniform manner with flat files *and subroutines*. In STREMA, all these are made available to the programmer as "streams", which resemble most closely the processes of FBP. You can specify graphically how "streams" are connected, and what happens to the fields in the records travelling through them - Clark uses the term "component" to describe what a field resides in as it is in transit through a given stream (not to be confused with FBP "components"). Streams drive each other, are described by a "relator", and may be subject to constraints on their components. Components (fields) have values, but they also have status: one of UNDEFINED, VALID or INVALID (similar to DFDM's dynamic attributes). As a record enters a stream, what happens is determined by the stream's "relator", and the constraints on, and status of, the incoming components. Constraints may be such things as bounds on a value, type specifications, or forcing a value not to repeat nor descend in a run. This concept can support processes as diverse as applying subroutines to streams, collating data streams, or getting data from or writing data to a relational table. Combining the concepts of relators and constraints simplifies a lot of the logic conventional programs have to do validating fields and deciding what to do if things go wrong. Clark has done a good job of combining a number of useful concepts into a single framework. Again, there are similarities with FBP, even though his thrust was slightly different.

**LGDF**

R.G. Babb II (1984) uses the term "large-grain data flow" (LGDF) to describe a level of granularity like FBP's, which he describes as a compromise between fine-grained data flow and traditional approaches. He points out that, under LGDF, the lowest level programs can be written in almost any language (he uses Fortran). He also starts the design process by showing data flow dependencies in diagram form and then converting them to what he calls a "wirelist". Here's a quote:

> "We have come to some unexpected conclusions... The most surprising is that sequential programs are often easier to design and implement reliably when based on a parallel (asynchronous) model of computation."

In case you start getting confused as you read this chapter: please take note that the term "message" used above is not the same as the Smalltalk "message", which is basically a linearized subroutine call, comprising a selector (which indirectly specifies the code to be executed) and the parameters to be passed to it. The more common use of "message" is similar to its meaning in colloquial English: a short piece of text carrying instructions and/or information from one

process to another.

**NIL**

This system, developed by Rob Strom and his team (1983) at the IBM Research Center at Yorktown Heights, has some very strong similarities to FBP, except that it is a programming language, rather than a coordination language. The original motivation for NIL seems to have been for programming communication software - you will perhaps have noticed that the multiple layers of communications software can be implemented as pairs of complementary processes. They also wanted good inter-process protection, which, as we have seen above, means minimizing side-effects. Like FBP, NIL also allows applications to be built up out of communicating sequential processes; only one process can own a data object at a time; "send" is "destructive" (the sent object can no longer be accessed by the sender); and so on. Strom makes the point that the ability to have processes on the same machine run in a single address space makes the cost of a message exchange very low, comparable to the cost of a subroutine call, and also makes it possible to have larger numbers of smaller-grained processes.

While it does not have an explicit coordination language, NIL is so powerful that this kind of language can easily be added - a parent process has enough facilities available that it could build a running program based on a file which specifies the connections between processes. NIL was deliberately designed to allow dynamic modification of networks, where processes can be created dynamically, and also ports of the created process can be dynamically connected to ports of other processes, both at process creation time and during subsequent execution of the process. The way it does this is by making ports objects, so that the process of connecting ports happens at run-time, under control of what are called "capabilities".

Just as in FBP, the only way processes can affect each other is via the communication channels. Strom and Yemini (in a recent paper on NIL) point out that this fact, plus an extension to strong typing in the NIL language called "typestate checking", provides a high degree of security. Since NIL has its own compiler, it can enforce typestate checking, and thus eliminate the risk of moving data to an address defined by an uninitialized pointer. DFDM, on the other hand, was explicitly designed to interface with existing languages: S/370 Assembler, PL/I and COBOL, and THREADS is C-based, so this risk is present with these implementations, but it can be minimized by good programming techniques and inspections. DFDM and THREADS also adopt the strategy of invalidating a pointer to an IP after that IP has been disposed of - this ensures that an erroneous attempt to access that IP afterwards will cause an immediate crash. Like the NIL group, we also found that this kind of environment provides very good isolation between processes. I believe there were very few cases where one process damaged another one's code or data.

Strom's group has since developed a follow-on to NIL, called *Hermes* (Strom et al. 1991) because Hermes was the "process-server of the gods"! They are currently building a "global

desktop" which will allow applications to be developed graphically. This is reminiscent of the work we did generating running programs from diagrams, alluded to in Chapter 2, but the global desktop will also allow programs to be connected up and reconfigured while they are actually running. It is designed to be used by people who may or may not be knwledgeable about programming. Strom's team has also been doing interesting work on what they call "Optimistic Recovery" - recovery strategies based on the idea that failures are rarer than successes, so one should go ahead with logic on the assumption that things will work and only undo it if there is a failure. Their infrastructure keeps the required information so programmers really don't have to think about recovery.

In the previous chapter I mentioned Rob Strom's remark about his team's decision to think of themselves as part of the OO world. In that chapter I also alluded to FBP-like work coming out of the OO fraternity, such as the media objects of Nierstrasz et al. (1992).

**Parallel Logic Programming**

There is an ever-growing series of parallel logic programming systems like *Parlog*, *Vulcan*, and a number of projects proceeding under the aegis of ICOT in Japan. PROLOG itself can be combined with FBP to give some very interesting capabilities, so it is not surprising that some of these projects are starting to look a lot like FBP: for example, I find *A'UM* by K. Yoshida and T. Chikayama (1988) very interesting. Incidentally this article has an excellent bibliography. The subtitle of this article is itself quite evocative: A Stream-Based Concurrent Object-Oriented Language - all the good words in a single title!

This same article started me thinking again about "streams". In A'UM and some of the other systems related to it, a distinction is made between "streams" and "channels". If I understand it right, in A'UM, a "stream" runs from one source to one destination, whereas a "channel" may contain more than one stream, coming from different sources: the items in each stream must stay in sequence relative to each other, but the streams in a channel are not constrained relative to each other. In A'UM only one reader is allowed for a channel, while in Tribble's paper on channels (Tribble et al. 1987), he allows multiple readers for a channel. The authors of A'UM feel that not allowing multiple readers makes their semantics sounder and the implementation simpler. Our experience tends to support this view.

In FBP we define a stream as the set of data IPs passing across one connection, but we also allow multiple sources (but only one destination). It may well be that the distinction between stream and channel is more rigorous than the FBP concept, and I look forward to seeing how these concepts evolve. We also pragmatically allow a stream which is accepted by one process to be passed on to the next, e.g. a stream of detail records might flow from a Reader, through an Edit, and on to a Process component. Some writers might prefer to call these multiple streams which just happen to contain the same data. I admit in hindsight that our concept of streams is a little fuzzy at the edges, but I feel it has never caused confusion in practice, and has been implemented

in all of the FBP dialects. Multiple destinations, on the other hand, have never been implemented in any of our implementations, partly because it is not clear whether the data should be replicated or should be assigned randomly to the receivers, like Linda's piranhas (see below) - in any case, both solutions can be realized very easily by means of generalized components.

Hewitt's *Actors* take processes down to the finest possible granularity: "Hewitt declared", to quote Robin Milner (1993), "that a value, an operator on values, and a process should all be the same kind of thing: an actor." This approach has considerable theoretical attractiveness, but in my view, to be practical, it basically has to be implemented as hardware, rather than software. There are also of course a number of projects growing out of Hewitt's Actors, which also seem to be on a converging path with all the other work (albeit at the more granular end of the scale), e.g. Agha's COOP (1990).

**BSP**

In Chapter 1, I said I would describe L.G. Valiant's work (1990) in a little more detail, so this is as good a time as any! BSP stands for "bulk-synchronous parallel", and it is of considerable interest because the author proposes it as a new "bridging model", to replace the current, von Neumann bridging model. He stresses that he is proposing it neither as a hardware nor a programming model, but to "insulate software and hardware development from each other, and make possible both general purpose machines and transportable software." The BSP model is defined as comprising the following three attributes:

1. A number of components,.....
2. A router which delivers messages point to point between pairs of components,
3. Facilities for synchronizing all or a subset of components at regular intervals of L time units, where L is the periodicity parameter. A computation consists of a sequence of supersteps. .... After each period of L time units, a global check is made to determine whether the superstep has been completed by all the components. If it has, the machine proceeds to the next superstep. Otherwise, the next period of L units is allocated to the unfinished superstep.

Now look back at Chapter 20, and you can see that Valiant's third attribute is very similar to FBP's use of subnets in checkpointing, except that he checks for completion on a regular basis - in FBP implementations, we usually count the processes, and then wait until that number have terminated. Otherwise it is very similar.

Valiant describes a variety of implementations, plus their appropriate performance measures, such as packet switching networks, a hypercube-connected computer and optical crossbars. Here is an interesting comment in his conclusion: "... if the relative investment in communication hardware were suitably increased, machines with a new level of programmability would be obtained." Note the juxtaposition: not just improved performance, but improved

programmability.

**UNIX and its descendants**

There is another group of related approaches, based on the very popular UNIX(tm) system. In these systems, the connectivity seems to be less rich, but the data passing between the processes is more like IPs or file records than messages. UNIX supports the concept of "pipelining", where the output of one process becomes the input of another, and so on repeatedly. This is definitely a form of configurable modularity, and I found a lot of their experience using this technique relates closely to things we discovered using FBP.

You will find the word "pipe" used quite often for what we call "connection" in FBP. In UNIX the '|' operator represents what it calls a "pipe". Processes can be assembled into working systems by connecting them together using this operator. For instance, suppose a user enters:

```
ls | pr -2 | lpr
```

The effect is for 'ls', 'pr' and 'lpr' to be assembled on the fly in such a way that the "standard output" of 'ls' feeds the "standard input" of 'pr', and so on. So this command means "list the files in the current directory; format the result 2-up and send the results to a line printer". This is very similar to the interpreted mode of DFDM and THREADS. UNIX's equivalents to ports are the file descriptors 0 (standard input) and 1 (standard output), which are automatically open whenever a process gets control. What flows between the UNIX processes is streams of characters, rather than structured IPs, so the metaphor is not as powerful as FBP's, nor does UNIX pipelining support complex networks. On the other hand UNIX's character string orientation makes it a very suitable for text manipulation, and a large number of the well-known UNIX components address this application area.

**OS/2 Interprocess Communication**

Orfali and Harkey (1991) list four techniques for interprocess communication in OS/2: anonymous pipes, named pipes, queues and shared memory. Anonymous pipes can be accessed by means of a write handle and a read handle and are mostly used by parent processes to communicate with their descendants, while named pipes allow communication between unrelated processes. Named pipes are accessed using standard OS/2 file services, so code does not have to distinguish whether it is writing to a file or to a named pipe - this will be determined by the file name.

**CMS Pipelines**

This system was developed by John Hartmann of IBM Denmark for the CMS environment. It is also consciously modelled on UNIX, but since it is specialized for the CMS environment, it is record-oriented, rather than byte-oriented. It supports more complex topologies than UNIX does by means of a notation for ending one pipeline and starting a new one attached to a previous

"stage" in the pipeline definition (using a label notation). I therefore see it as a halfway house between UNIX and FBP (again developed independently of FBP). A program may also dynamically redefine the pipeline topology by replacing itself or one of its neighbours with a newly defined pipeline.

Here is an example of a CMS Pipelines pipeline:

```
pipe < logged users | split , | take 5 | console
```

This is a CMS command to set up a pipeline which reads the file called LOGGED USERS, splits each record into multiple records, using the comma as the delimiter, selects the first 5, and displays them at the console.

Pipelines components are written in REXX, using a SUBCOM environment for the "pipe" services.

**CSP (Communicating Sequential Processes)**

This seminal work by Tony Hoare (1978) has been the basis for a large amount of work by other writers. Here, the external specification indicates which processes are running concurrently (using the '||' operator to indicate parallel execution), not how they are connected. The actual connectivity is implied by the send and receive commands, which must explicitly name the process being communicated with. Connections are assumed to have zero capacity.

As an example of the CSP coordination notation, here is Hoare's notation, given in his article, for what I have referred to above as the Telegram problem, as follows:

```
[west::DISASSEMBLE||X::COPY||east::ASSEMBLE]
```

Here "west", "X" and "east" are process names, while the capitalized names stand for program sections (analogous to FBP components).

The problem is that DISASSEMBLE has to know the name "X", COPY has to know the names "west" and "east", etc. Hoare's orientation seems to be towards 'write new', rather than reuse. He does mention port names as a possible solution to this problem, but doesn't stress the fundamental paradigm shift involved in changing from 'write new' to reuse, nor the importance of finding a good notation for combining black box components.

Interestingly, he also makes the same point that Carriero and Gelernter made about a subroutine being a special case of a coroutine.

You will notice the frequent occurrence of processes, connections and sometimes ports, with different names being used from system to system. Also configurable modularity at any more complex level than that of UNIX requires some agreement of names (or numbers) between the outsides and insides of processes. NIL avoids this by making ports local to a process, but

allowing a parent process to pass information about connections to the child process in the form of parameters.

**Linda**

Now let's move off in a different direction: Carriero and Gelernter, whom I have mentioned above, have developed and written extensively about a very interesting system called *Linda* (1989), which has stirred up a lot of interest in academic circles. Instead of IPs, Linda uses "tuples", ordered lists of values. "Tuples" are created in "tuple space", just as FBP IPs are created in space managed by FBP, not by the components. Unlike FBP's IPs, however, a tuple just floats in tuple space until retrieved by a process which knows its identification (or part of it). Professor Gelernter uses a neat analogy in a recent Scientific American article to explain his concept of a "tuple". Imagine two spacemen working in space building a space station: one of the workers has finished working with a wrench and wishes it to be available for other workers - he or she can put it "down" (so that it follows its own orbit in space), and the other worker can then pick it up whenever convenient. In the same way, tuples or FBP IPs have an independent existence and follow their own orbit from one worker (process) to another.

How does the spaceman actually pick up the wrench? Here is chiefly where Linda diverges from FBP: in Linda, access is done by pattern matching. A process may need a tuple with value X in field Y - it just has to request a tuple matching those specifications, and it will eventually receive an appropriate tuple. Receiving a tuple can be destructive or non-destructive ("consume" or "read"). If more than one tuple matches the specification, the system picks one at random. If there are none, the requesting process is suspended. Values are not communicated back from the receive to the tuple.

One other important feature of Linda is that of "active" tuples - these are tuples which execute code at the moment of creation, and then turn into ordinary "passive" tuples. You can do distributed logic such as matrix operations this way, where each tuple does a calculation when it is created and then becomes a passive matrix element. Perhaps the nearest to this in FBP is something like the Subnet Manager, which takes a passive chunk of code and turns it into a process.

Another Linda image which is evocative is the "school of piranhas". Here a number of processes lie in wait for a tuple, and when it appears in the tuple space any one of them can grab it. This is an effective technique of load balancing. In Chapter 22, I described a performance improvement technique where we had 18 occurrences of a process executing the same disk traversal logic. In this case, we had to provide connections from the client to the 18 servers, so we had to have a "load balancing" process in between - the piranha technique would do this without the need for the extra process.

To summarize the essential difference between Linda and FBP, Linda is a *bus* while FBP is a *tram* - Linda has more degrees of freedom but I believe is more expensive in terms of resources.

If one Linda process generates a series of tuples with an extra field containing numbers in ascending sequence, you could have another process request them in sequence. So Linda can simulate FBP. Conversely, FBP can just as easily simulate Linda's associative style of retrieval by having one or more processes manage an associative store. It seems to me that Linda and FBP are so closely related that systems of the future may combine both approaches. After all, there may be areas of an application where tracks are more appropriate, and others where you want more degrees of freedom - sort of like containers moving from rail to ship and back to rail. Here is a final quote from Gelernter and Carriero (1992):

> "In general we see computation and programming languages as areas in which further progress will be slow, incremental and, in many cases, of marginal importance to working programmers. Coordination languages are a field of potentially great significance. A growing number of groups will play major roles in this work."

I couldn't agree more!

**RIG, Accent, Mach**

These are a series of network operating systems (Rashid 1988), each one evolving out of the previous one. RIG (Rochester's Intelligent Gateway) was an interprocess communication facility allowing processes to communicate by means of information packets. RIG allowed any process to access any other's port using a pair of integers (process number.port number). While this allowed port numbers to be passed around as data, it meant that port defined services could not easily be moved from one process to another. Also, if a process failed, this information could not be signalled back to processes depending on it. Processes therefore had to register their dependence on other processes with a special process which was notified of process death events (called the "grim reaper"). This makes me wonder if systems like Linda may not suffer from the same problem.

Accent evolved out of RIG, by defining ports to be capabilities as well as communication objects and adding an integrated virtual memory management system which allowed processes to transmit large objects (RIG could only transmit 2K bytes at a time). Here is a quote about Accent:

> "Experience with Accent showed that a message based network operating system, properly designed, can compete with more traditional operating system organizations. The advantages of the approach are system extensibility, protection and network transparency."

Mach then evolved out of Accent because of the need in their environment for complete UNIX compatibility, and "to better accommodate the kind of general purpose shared-memory multiprocessors which appear to be on their way to becoming the successors to traditional

general purpose uniprocessor workstations and timesharing systems." Among other things, Mach splits the Accent concept of "process" into "task" (basic unit of resource allocation) and "thread" (basic unit of CPU utilization).

**Advanced Hardware Designs**

Another characteristic of the three "legs" of FBP listed above is that they could actually describe a network of independent processors, all interacting to do a job of work. This approach to building super-powerful machines is getting a lot of attention lately, as it is generally recognized that the single processor is running out of steam. Although we can probably make individual processors smaller and faster (after all, a bit is simply a choice between two states, for instance two states of a molecule), you start to run into limitations such as the speed of light or the potential damage which can be caused by a single cosmic ray! A lot of work is going on on linking multiple processors together to achieve enormous amounts of computing power without any one processor having to be incredibly fast. Suppose we put 1000 processors together, each running at 20 MIPS (millions of instructions per second) - this would provide 20,000,000,000 instructions per second. Since such a machine is normally thought of as being oriented towards scientific calculations, so that the instructions would tend to be floating-point operations, this machine would be a 20 gigaflop machine. Multiply either of these factors by 50 (or one by 5 and one by 10), and you are into the "teraflop" per second range (1,000,000,000,000 floating point operations per second).

The two main approaches here are *multiprocessors* and *multicomputers*. I am indebted to my son, Joseph Morrison, for some of the following comments. 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. I found the article by P. Gaughan and S. Yalamanchili (1993) a good survey, as well as providing some interesting solutions and simulation results. Of course, I am not a hardware person, but it does seem that some of the techniques described would support FBP quite nicely.

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. There is an enormous amount of research going on in the areas of multiprocessors and multicomputers. This is a whole area of specialization within computing research, and one which I expect I will never get to know very much about! However, a lot of people who do have some understanding of this area see a good match with FBP. Applications designed using FBP should map very naturally onto a system of communicating processors. If you have more processors than processes, this mapping should be pretty easy; if less, then some processors are going to have to time-share, just as the present implementations of FBP do today. Here is a quote from an article (Cann 1992) comparing FORTRAN with functional languages for programming tomorrow's massively parallel machines (remember that we related FBP to functional languages in an earlier chapter):

> "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.
>
> "In summary, the functional paradigm yields several important benefits. First, programs are more concise and easier to write and maintain. Second, programs are conducive to analysis, being mathematically sound and free of side effects and aliasing. Third, programs exhibit implicit parallelism and favour automatic parallelization. Fourth, programs can run as fast as, if not faster than, conventional languages."

One last point about the data flowing between these computing engines: a lot of the

mathematically oriented work with big computers (and most of this work is mathematically oriented) seems to assume that what should travel between the processors is either values, like integers or strings, or messages. I actually started out with values in my early work, in 1967, but became convinced over the years that you should send whole "things" (entities, records or IPs), which stay together as they travel, rather than low-level datatypes (e.g. integers or strings) which are not in the application domain. Our experience is that, if you decompose a record into individual values, it may be so expensive to recombine them that it's not worth it.

**FBP without FBP**

Maybe now is the time to talk about how to do FBP without FBP software. This could be a stepping stone to full FBP for a number of companies. Many of the basic concepts can be simulated with conventional languages without any special software. A friend of mine took our AMPS course, but then went to work on a COBOL application. In those days we had no FBP support for COBOL, but he told us AMPS had helped him write better COBOL programs, by suggesting better ways to structure his code.

Of course, you could even use JCL, using files for your connections, if you didn't care about overhead. Remember that the steps in a job must execute in a fixed order, so you would have to string out your network into almost a straight line - and forget about loop-type networks. Another similar approach is to use secondary transactions in IMS - this has actually been tried several times, but the overhead prevents the granularity from being made fine enough to be really productive.

A number of Data Flow COBOL products available in Japan generate pure COBOL from data flow specifications, which can then be compiled and link edited like any other COBOL program. They support rather simple networks, but require no special run-time software.

An article (Kar 1989) written by a senior engineer with Intel Corporation, shows how to do calculations using data flow, rather than synchronous code. He gives the actual C code to do this. He sees what he calls "data flow multitasking" as a way that we can use today to write programs which will not only be easy to port to the powerful, multiprocessing machines coming on stream during the 90s (and he should know!), but which will take advantage of these machines' capabilities. In the summary section he says,

> "Data-flow multitasking is a promising solution to the challenges software faces over the next decade. *It involves looking at a sequential program through new eyes* [my italics]. ... Data-flow multitasking is particularly relevant for real-time applications."

Along somewhat similar lines, some of my early data flow research was done using a single APL program to simulate the scheduler. In this case, different processes were implemented as sections of a single APL program, and control was transferred between them by using a computed "goto". This would be very easy to do in any HLL. Another approach would be to use single data areas to

represent connections, together with a switch to indicate whether a given area is occupied or not. The scheduler could then cycle looking to see which process has data waiting to be processed. Wayne Stevens pointed out (1991) that non-looping FBP components are very similar to subroutines, and therefore networks consisting mostly of non-loopers should be easy to implement directly using a HLL.

At the beginning, I mentioned the growing use of these concepts for programming distributed systems. I would like to mention the recent IBM announcement of **IBM Messaging and Queuing (MQSeries)**. Similar efforts by other companies are listed in a recent article (Moad 1993). IBM plans to bring out a set of products which will allow asynchronous communication between a large set of IBM and non-IBM platforms. There will be a standard interface based on the concept of queues, which will relieve programmers from the complexities of making different applications communicate between different vendors' hardware and software. Instead of having to use one set of macros for VTAM, different commands for CICS, still another set for IMS, all applications will use the same simple set of MQI (Messaging and Queueing Interface) calls. This concept seems to me to be completely compatible with the application structuring ideas presented in this book. In Chapter 15, we talked about the ability to "cleave" networks across systems - the combination of FBP and MQSeries or similar software should provide a very powerful way of splitting an application across multiple systems or locations, or of moving functions from one node to another as desired. As I said in that chapter, cleaving applications between different computers introduces new problems of synchronization, but they will definitely be solved! It doesn't make sense to try to pretend that a distributed system is one big, synchronous application. If you request data from a remote location, you want to be able to do other things while the data is working its way through the system. Systems are getting just too big. Also, many of the systems which are being connected "speak different languages", so we are seeing the development of standards which will allow them to interpret each others' data formats. I predict that those problems which will inevitably arise will be solved, but not necessarily by means of one general solution for every problem.

As you read this chapter, I hope you have got some impression of how ideas spring up independently in different places and times, how they flower in unexpected places, how they cross-pollinate and give rise to interesting new hybrids. You have to be a botanist to keep track of it all! There are many other concepts and languages, other than the few I have mentioned here, which have points in common with FBP, and which have certainly been influential in the field of computing, but there is not room in this book to do them justice. They have cross-fertilized each other and in many cases only industry archivists know which led to which. Examples which spring to mind are: *SIMULA*, *MODULA-2* (and now *MODULA-3*), *Concurrent Pascal*, *Ada*, *Lucid*, *Occam*, ....

It would be wonderful if any readers who are expert in these different languages could share their knowledge and insights with the rest of us. I have occasionally dreamed of collecting all the

developers and theoreticians of concurrent, stream-oriented, modular systems together in a huge symposium, and seeing if we can't get some synergy going. I have found that there is something about data-orientation which seems to allow practitioners of different disciplines to communicate (just as IPs do for different languages!). I sincerely hope that we won't waste our energy in internecine wars, as has happened in some disciplines, but that we will all be able to work together towards a shared goal. There are far more similarities than differences in our work, and if we can get a real dialogue going, who knows what we might achieve together!

# Chap. XXVIII: Ruminations of an Elder Programmer

*[This article harks back to a technology (unit record machines or EAM) so inconceivably ancient that many of my readers were probably not born then - yet this was the only form of business automation many companies used in those days!*

*DFDM was the second implementation of FBP at IBM.]*

The following was sent to me by a colleague named Art Huber a few years ago - you can get some idea of his amusing way of expressing himself from what follows. It is reprinted with his permission.

**Subject: DFDM thoughts**

Last night, when I couldn't sleep because of a cold, I thought about DFDM. Why not?

I thought about the analogies we use as an aid to understanding it. None will be completely satisfactory, but I have been working on one that I have not heard previously. I think it is most useful for old farts such as me. Less experienced programmers and analysts, whose synapses are not already connected hierarchically don't need this one.

Once upon a time there were no computers. There were, however, unit record machines. There were collators, sorters, keypunches, printing accounting machines, calculating summary punches, etc.

I think a DFDM network is very similar to a job in a unit record shop.

Batches of records move from machine to machine.

Each machine performs a single operation on the data.

Each machine is a general purpose machine that is programmed to customize it for the required task. Note that by "general purpose", I do not mean that it can perform any task; I mean that it is not tied to a particular record format or job.

There are some significant differences, of course, between DFDM and a unit record shop.

The DFDM "machines" run at electronic speed rather than mechanical speed.

They are cheaper to build. If you need a new special function machine, you can create it in software. Consequently, there is no economic advantage to creating multi-purpose machines as there would be with real unit record machines.

DFDM machines can be replicated as needed. The new copies don't cost anything to operate and don't take up floor space or use additional electricity.

DFDM machines are not restricted to 80 character records.

The records will not fall on the floor between steps.

Parameters that tailor DFDM machines can be much simpler than unit record plugboards. For example, in DFDM, a field can be defined by specifying its starting position and length. Unit record machines require a character by character definition.

Because setup is so much cheaper (since the DFDM machines are virtual), it is viable to process batches of one record with DFDM.

DFDM machines need not be limited in the number of accumulators they contain, since they are implemented in software.

Jobs do not have to be scheduled based on machine availability, since DFDM machines are virtual. Of course, they are implemented on real machines (computers) and are constrained by availability of computer resources.

Isn't this fun?

I offer this for your consideration. It does point out that the job of "programming" in DFDM is very different from what we think of traditionally. We may even have an opportunity for old unit record analysts.

## AND EVERYTHING OLD IS NEW AGAIN!

# Chap. XXIX: Endings and Beginnings

I did not want to call this chapter "Conclusion" as I hope it is more of a beginning than an ending. We have certainly travelled a long distance, and if you have stuck with me you are definitely to be congratulated! However, the journey in many ways is just beginning. We have come a long way, but these are only the first small steps.

Programming in the year 2010 is going to be very different from what we are used to today, it is going to *feel* very different, and it is going to be done by different types of people from the practitioners of today. Brad Cox wants to get his "Software Industrial Revolution" started, and I agree the concepts are there and the time is right. He points out that today we are still software craftsmen, making once-off items one piece at a time with very simple tools. If you want to read about how things were done before mass production, read "A Book of Country Things" (Needham 1965). These were smart people who knew their materials, and many of their techniques were excellent. But you didn't go to the store to buy a standard part - you made whatever you needed yourself. Now, isn't it rather strange that so many programmers today would still rather build a new piece of code than (re)use one that already exists! Some of them certainly do it because they love it, and there is a role for these people to play, but modern business cannot afford to continue to have code built laboriously by hand. In the old days, people did this because they *had* to; today, we have had to divide up the work - some people do the designs, others make them. Machines make all of us more productive. Are people less happy today? I believe only a Luddite would maintain that the old days were better - yes, some of those old tools were works of art, but we have magical things available at our fingertips which one of those old-timers would give his eye-teeth for. Cheaper doesn't have to mean nastier. If you believe that programmers don't reuse on account of their love of their craft, I think that's wrong. Most programmers don't reuse because it's just more trouble than building new. But, you see, our experience with FBP has been completely different - in one case I mentioned earlier a programmer used an off-the-shelf component to do a function, even when she could have done the same thing by adding a single line to an adjacent PL/I component she was writing! So there are profound psychological differences between the two environments.

I believe that an important part of the change is that application developers who are trained in FBP are moving away from procedural thinking towards what you might call "spatial" thinking.

Procedural thinking is quite rare in ordinary life, so it's not surprising that people find it difficult, and that its practitioners are (viewed as) different in some ways from ordinary mortals.

If you start to look for examples of procedural thinking in real life, you discover that there are in fact very few areas where we do pure procedural thinking, but there is one which we have been doing since we lived in caves, and that is cooking. Some years ago at the IBM Research Center in Yorktown, Dr. Lance Miller started studying the differences between recipes and programs. He noticed that recipes often had implicit parallelism, e.g., "add boiling water" implies that the water must have started boiling while some other step was going on, so that it would be available when needed. The term "preheated oven" is so common we probably don't even notice that it also violates sequentiality. People say and write things like this all the time without thinking - it is only if you try to execute the instructions in a rigidly serial manner (i.e. play computer) that you may run into some surprises!

The other thing he noticed was that the individual steps of the process very often start with a verb and then add qualifiers, e.g. "boil until done", "beat egg whites until soft peaks form", so you know very early in the sentence what kind of activity you will be doing. In programming, we usually bury the "verb" deep inside a nest of do-loops to get the same effect, e.g.

```
do while a....
   if c is true
      then
         do until b...
            compute
         enddo
   endif
enddo
```

The effect of this is that you don't know what operation you are going to do until you are deep inside a nest of do-loops or conditional statements. This would be like telling a visitor to town, "Until you come to the fountain after the church across from the train station, keep going straight". The visitor would get a pretty strange impression of your town and its inhabitants!

The other main kind of procedural behaviour in fact is just this one: following directions. Have you ever tried following someone's directions, only to find out that they forgot one item, or someone changed a street sign, and you are now facing north instead of west (if you can even figure that out). If you want to make a grown man or woman cry, ask them to assemble a child's tricycle, whose instructions have been translated into English from another language by a speaker of a third language, and which probably describe the wrong model anyway, if you could figure out what it meant. A lot like programming, isn't it? Maps are much easier because they let you visualize relationships between places synoptically, so you can handle unexpected changes, make corrections, and even figure out how to get to your destination from somewhere your informant

never imagined you'd land up! On a recent trip to England, I found myself very impressed with the amount of information packed into the signs announcing "roundabouts": general shape of the roundabout, angles, destinations and relative sizes of roads entering the roundabout - and all specified visually!

This is something like the difference between conventional programming and FBP - FBP gets you away from procedural thinking and lets you think map-style.

It occurred to me recently that we finally have a unique opportunity, both to take advantage of the CPU power available on most programmers' desks today and to actually use this power to take advantage of natural human abilities. Just like other human abilities which we take for granted, visual processing is extremely difficult to program into a computer, and requires a lot of computer horsepower. However, many programmers today have powerful machines sitting on their desks which are basically being used as "dumb" terminals. A few years ago, Bucky Pope of IBM did a study in which he concluded that "editing" text really doesn't exercise a machine much - most of the time the programmer is just thinking. So entering a linear string of COBOL doesn't take advantage of the power on his or her desk. And what power! If you remember Chapter 22 on Performance, I said that it took 10 microseconds to do an API call on an IBM 168 (on which we ran a bank - 5,000,000 accounts) 20 years ago, and 50 microseconds on the machine on my desk at home today - so that means I have the processing power to run 1/5 of a bank right here on my desk (yes, I know that's stretching it a bit!). What we might call "visual programming" could actually start to take advantage of the relatively enormous processing power available to each programmer today. And visual programming means much more than drawing pictures to represent logic - it means developing a synergy between human and machine which takes advantage of one of the human's strong suits, not his or her weakest. A recent article by K. Kahn and V. Saraswat (1990), alluded to earlier, which I found absolutely visionary, describes an approach to a totally visual style of programming, which would not only have a visual syntax, but would also show computation using visual transformations. Software supporting this would have to be able to perform and understand topological transforms, just as humans do without effort. Interestingly, my group at IBM built a visual animation showing the creation and movement of IPs through an FBP network, and showed this at a CASE conference a few years ago - this proved a very effective way of conveying some of the basic concepts of FBP. We have also speculated that visual interaction with a picture of a network would be a very natural debugging environment. At this juncture, 7 years from the end of the century, I believe we have the computer horse-power to make such approaches economically feasible - now we just have to develop the technology.

Up to now, I have concentrated on technology, and I confess to being technologically-oriented, so assume we have gotten these details out of the way. However, we also have to look at the sociological and psychological factors. What will be needed to get such a technology into use in the workplace? Well, for one thing it is 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 noncommunication, we are not going to be able to make the transition to a new way of thinking. Business has gotten the impression that it has to become a bunch of mathematical geniuses to do the new programming, because the academics are broadcasting that image. So it retreats into its corner, and keeps trying to build and maintain systems using linear COBOL text. However, I don't believe that we all have to become mathematicians - nothing in this book would give a bright 16-year-old any difficulty at all.

Instead, at this point in time, business people are more willing to hire hundreds of COBOL programmers than to invest in new technologies. The problem is, if you were a CIO (maybe you are), which technology would you invest in? Well, currently it is not a very hard economic decision - it makes more sense to stay with the COBOL coders. At least you can do anything with COBOL (no, I am not refuting everything I have said in this book) - but it takes ages to do it, and the result is almost unmaintainable. Now suppose you were running a cotton plantation a century or so ago, using all manual labour - not very productive, but at least output was predictable, if slow. Imagine, now, that some city slicker comes along with a cotton-picking machine, which he claims is going to improve productivity enormously. But you, the land-owner, figure that you are going to have more highly trained people running it, it may break down at awkward moments, it's going to need parts from halfway across the country, and so forth. You'll do the "smart" thing, right? It wasn't until the prevailing morality started to take into account the feelings and needs of *everybody* involved *and* the technology reached a certain level of maturity that the balance tilted in favour of using technology (I wish I could say that this has happened universally, but at least it's a start). I date many of my feelings about technology to a visit I made to a match factory as a schoolboy (circa 1948) - that was the nearest I've ever come to seeing humans used as robots, and I never want to see anything like it again...

I often think the attitude of business is best summed up by a cartoon I saw a few years ago - a scientist type is trying to sell a medieval king a machine-gun, and the caption says, "Don't bother me - I've got a war to fight"! Why should we change the way we do application development? Everyone's happy with the status quo, right? I don't think so. I think in fact there is a general dissatisfaction with the way things are now - it's just that nobody has shown a clear way to solve it that will benefit all the stakeholders. Over the last few decades, there has been an unending series of snake-oil salesmen, each peddling their own panacea. Why do you think each new remedy gets adopted so enthusiastically? Because there is a real need out there. So far, they have all turned out to be inadequate in some way, and usually get dropped. This is understandable, but it is very important that we learn from each such experience, so that collectively we move forward, rather than just jigging in one place all the time.

But, to give all groups a chance to take potshots at me, I am afraid academia is partly to blame as well. Some academics, I am afraid, are doing the modern equivalent of fiddling while Rome burns. A professor of computer science told me a few years ago that, at his university, a thesis on

application development technology just wouldn't be accepted. I found this shortsightedness absolutely incredible, and he agreed! It would seem to be obvious that application development technology is fascinating stuff, and it's even useful! Hopefully this attitude has changed in the years since, but I'm sure it has cost application development research a good decade or two. Even if theses on application development are now being written, how do we get these ideas into programming shops around the world?

I have a warning for any academics who may be reading this - there is so much stuff to read now in this area, that it would be quite understandable if you ignored papers like the one that appeared in the *IBM Systems Journal* back in 1978, because they aren't full of Greek letters. However, we were using these concepts to run a real live bank, and building up real experience trying to make these concepts work. This experience is *priceless*, and perfectly complements the interesting theoretical work that you people are doing all over the world.

Inflexible systems not only cost money, but they contribute to users' perception of computers as inhuman, inflexible, and oppressive. How many times have you been told, "It's the computer", when confronted with some particularly asinine bit of bureaucracy? We know it's not the hardware's fault but too often it's the fault of some short-sighted or just over-burdened programmer. Does the public know this? If they do, they've probably been told, "Yes, we know it's awkward, but it would cost too much to change it." Why does it cost so much? If there is only one message I want to leave you with after reading this book, it's that *the root cause of the present state of programming is the von Neumann paradigm*. It's that simple, and that difficult (you know we humans prefer things to be complex but easy, like taking a pill, but life isn't like that).

We started this book talking about how we have to relax the tight procedural nature of most of today's programming, imposed, not so much by the von Neumann machine, as by the mistaken belief that we still have to code according to the von Neumann model. Internally, today's computers are no longer tightly synchronous - nor are the environments that they run in. This can now be seen as a special case of a much larger issue: the key to improving productivity and maintainability in application development and to making programming accessible to a wider public is to make the world inside the computer match more closely to the world outside it. The real world is full of many entities all doing things in parallel: you do not stop breathing when I draw a breath. It is therefore not surprising that we were specifically designed to be able to function in such a world, and we get frustrated when we are forced to only do one thing at a time.

In a similar development, but at a different scale, we are starting to distribute our systems across different machines and/or different geographical locations. Imagine a world of multiprocessor machines communicating over LANs, WANs or satellite links across the whole world, and you get a vision of a highly asynchronous, massively parallel data processing network of world-wide scope. Since this is clearly the way our business is going, why should we have to be restricted to using synchronous, non-parallel, von Neumann machines as the processing nodes?! So our

applications would be networks, networking with other networks, extending eventually around the planet.

Fact is starting to catch up with fiction: John Brunner is thought to have originated the use of the word "worm" in his 1975 novel, *The Shockwave Rider*, to describe an autonomous program travelling the computer net. Hackers (using both the positive and negative connotations of that term) have developed computer viruses, whose behaviour mirrors that of biological viruses. If you have had a system attacked by one of these critters, it may be hard for you to think kindly thoughts about what are usually perceived today as nuisances at best, and at worst something downright dangerous and destructive. However, a lot of recent work by responsible scientists has suggested that crossbreeding these two species may result in very powerful tools for making computers more user-friendly. Read what E. Rietman and M. Flynn (1993) have to say about worms and viruses in the future. They describe scenarios such as: worm programs assembling personalized newspapers for subscribers, using data extracted from databases all over the world; "vampire worms" taking advantage of available CPU time to do useful jobs (at night, mostly - hence the name); viruses automatically inserting hypertext buttons into text databases; viruses doing automatic database compression and expansion as time becomes available, scavenging dead data, monitoring for broken data chains, and on and on. In fact, there is already a precursor of this kind of facility roaming the net, called "Gopher" (Martin 1993), which lets you look at the weather in Australia from a terminal in Ohio, and generally roam around "gopherspace" from any terminal connected to Internet, or from a terminal talking to a remote computer that can access the Internet. Gopher now seems to have been upstaged by the WorldWideWeb, which sounds truly marvellous (Cramer 1994)! [I seem to have been truly prescient here - remember this was written in 1994!] As an interesting aside, Rietman and Flynn also point out that worms could be a useful method to "*program massively parallel computers"* (their italics).

Talking about fact catching up with fiction, some recent work coming out of Xerox PARC is even more incredible. Instead of creating make-believe environments, which people can move through using the computer (Virtual Reality), how about enhancing our real-world environment with a myriad of small computing devices that we can talk to, using voice and gestures, *and that talk to each other*, perhaps about us? How about an office that automatically adjusts the temperature and humidity to suit whoever is occupying it, and plays soothing music if you want it to? How about children's building blocks that can be assembled into working computerized structures, or a pipe that you can use to point at a virtual blackboard, has a small microphone and speaker inside it, and perhaps monitors its user's health as well?! A lecturer once asked us: "Where will computers be used?" and he answered: "Anywhere that it makes sense to put the word *intelligent*." For example, not just intelligent cars or planes, but intelligent offices, desks, and blackboards, intelligent bookshelves - maybe even intelligent paper and pencils. This work goes under the general name "ubiquitous computing" or "ubicomp", and you can read about it in Vol. 36, No. 7 of *Communications of the ACM* (Wellner et al., 1993). This enormously exciting

work seems to me to be totally compatible with everything that has gone before in this book.

The IBM scientist Nat Rochester once described programmers as working more closely than any other profession with what he called pure "mind-stuff". If you imagine a world-wide network of "mind stuff", then this corner of the universe is starting to see a new type of intelligence, or at least a new vehicle for intelligence. Baird Smith of IBM used to describe software as "explosive" while hardware is "implosive". While software, which is built out of mind-stuff, is becoming more and more powerful and complex, hardware is getting smaller and smaller (although more complex!) and cheaper and cheaper - as it should do, since, after all, its basic building material is sand! This is a perfect example of what Buckminster Fuller called "doing more with less".

I think we are starting to see a truly massive convergence of ideas. It is unfortunate that most of the science-fiction which has been accepted by the main stream is dystopian, because most good science fiction is very optimistic and upbeat. It has always surprised me how few programmers, who live their professional lives on the cutting edge of change, actually read science fiction. Does this reflect a perception that what they do is not imaginative and exciting? Is this still another vicious circle?

J.W. Campbell Jr., the editor of Analog, originally Astounding, was an inspiration to a generation of science-fiction readers. He taught us the value of considering new ideas objectively, avoiding the extremes either of rejecting ideas out of hand because they are new and different, or accepting them instantly without proper evaluation. It was from his editorials that I learned the idea of "rope logic", which I believe this book embodies: the ideas described in this book are not built up on a basis of Aristotelian syllogisms, but as a multitude of small threads. While individually none of the threads may be very strong, they complement each other, resulting in a rope of logic which, in its totality, is strong enough to move pyramid blocks around (and maybe even a whole industry)! By the same token, you, my readers, may be able to snap a few of these threads, but my conviction is that the rope as a whole will remain as strong as ever. Am I deluding myself? The only possible test is the systems built using these concepts, and they are some of the sturdiest systems the companies that use them have in their collection.

While it may be true that some science fiction is naïve, some of the most exciting and forward-looking thinking going on today is described in the fiction and fact pages of today's science-fiction magazines. Read them and stretch your minds! If we have the will, we can make pretty good lives for ourselves. In my experience it is the pessimists (have you noticed they usually call themselves realists?) who show a certain naïveté - the naïveté of believing that things will go on just the way they are, that there will not be technological, political, commercial, social, artistic or spiritual breakthroughs. In fact, the pace of change in all areas is on an accelerating curve - if you doubt this, just look back 50 years and see how far we have come. We are only limited by our imaginations, and that's a resource that is never going to be used up! Even if you feel solving the world's problems is too big a task, I see no reason why we can't at least tackle the smaller task of making programmers more productive and programming more fun - there is no law that says

work has to be drudgery. In fact, when one masters a medium, and the medium fits the hand, there is a feeling of being at one with one's tools which can border on the transcendent. That's what training is about - not to turn out a generation of button-pushers, but to produce "masters". Our goal in our profession is not to be able to push a button and turn out a payroll program, but to be more like those ancient Celtic artisans who made a drinking vessel a work of art. Were they having fun? You just bet they were! And so can we - so why not start right now?! And if this isn't sufficient justification by itself, wouldn't it be neat if we could have happier, more productive programmers, working for companies that are more responsive to their clients, and saving everybody money as well (and therefore improving the quality of life for you and me). Utopian? Perhaps, but if we have a goal, we can start moving in that direction, and we can measure how much progress we have made towards getting there. If we have no goal, then we'll be just wasting energy running round in small circles. Programming is a part of life, so this idea really shouldn't surprise anybody!

So what's going to happen over the next couple of decades? I used to think that if you built a better mousetrap, everyone would beat a path to your door. I learned differently by personal experience - and then I read Kuhn (1970), who put it all in perspective for me. Did you know that the phlogiston theory didn't yield instantly to the oxygen theory the moment someone did the deciding experiment? Some people *never* really took to this weird oxygen idea! Me, I'm betting on the next generation. Here we have a new paradigm which is clearly superior, but experience shows that it may take as much as a generation for it to catch on. But it could take a lot less than that! By the way, if you want to learn more about paradigms, how they affect how we think and how they get adopted, Kuhn's ideas have been extended over the last few years in a series of thought-provoking books and video tapes by Joel Barker (e.g. Barker 1992). Necessary reading!

As I have tried to show in Chapter 26, some of the most advanced practitioners of OOP are discovering FBP or something very close to it. There will be people who say programming will never become simple, or that the man or woman in the street will never see it as enjoyable. I personally believe that computers and people will always need go-betweens, just as people from different cultures do (sometimes so do members of the same family!). Programmers are skilled at interpreting between people and machines and there will always be a need for their services. However, by giving programmers inadequate tools, and then blaming them if they aren't successful, we scare off the very people who would be best at this work. It is time we stopped doing that! It's never too late to realize that we took a wrong tack a few decades ago, and change direction.

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. We also have today the potential to create a new era of productivity, based on a marketplace of reusable components. Wayne Stevens, who is responsible for a number of the ideas in this book, was very optimistic about the potential of

these concepts and was tireless in promoting it within the DP community, putting his not inconsiderable reputation behind them. Where I was the *paradigm shifter*, to use Joel Barker's phrase, he was the *paradigm pioneer*. He believed, as I do, that these concepts will have a big effect on our future as an industry, and I deeply regret that he was not able to see them become widely accepted in his lifetime. I have always liked the phrase, "Will those who say it can't be done please move aside and let the rest of us get on with doing it" - this was very much his attitude to life! I and my colleagues over the years have had a glimpse of the future of the DP industry, and I have tried to share this vision with my readers. I hope that you have enjoyed reading about it as much as I have enjoyed telling you about it!

# Appendix A: THREADS Network Specification and Component API

In Chapter 23, we described the THREADS notation for linking components into networks - this will also be repeated below in summary form. However, perhaps even more important are the conventions for writing components, as these are what will allow independently developed components to be combined into a single network without the requirement that they be developed by the same person or even in the same location (apart from the normal requirements for consistency of IP and stream formats).

Here is a simple component written in C to copy IPs from IN to OUT. Let's call this rather unoriginally `copydata`. I am using the "large" model and C calling convention.

A THREADS component is a regular C subroutine, so, as you might expect, the first few statements are the declares.

```
/* Include for THREADS service call prototypes */
  #include "compsvcs.h"
  /* This header file also includes definitions of "anchor"
     and "port table" */
  /* Prototype for component code */
  int copydata(anchor proc_anchor)  {
  /* Initialization of "port table" - the array dimension
     must equal the number of ports */
  port_ent port_tab[2] = {{"IN\0",0,0,0},{"OUT\0",0,0,0}}
  /* Declares for variables used in component body*/
  void *ptr;
  int value;
  long size;
  char *type;
  /* Component Body  */
    value = dfsdfpt(proc_anchor, 2, port_tab);
```

```
    value = dfsrecv(proc_anchor, &ptr, &port_tab[0], 0,
         &size, &type);
    while (value == 0)
    {
     value = dfssend(proc_anchor, &ptr, &port_tab[1], 0);
        if (value != 0)
          value = dfsdrop(proc_anchor, &ptr_2);

      value = dfsrecv(proc_anchor, &ptr, &port_tab[0], 0,
           &size, &type);
    }
  return(0);
}
```

The first action is always to initialize the port table using a call to `dfsdfpt` - the 2nd parameter of the call to `dfsdfpt` must match the dimension declared for the port table. This number must also match the number of port names specified in the parameters to `dfsdfpt` . Note: `dfsdfpt` modifies the port table, so this must not be defined as constant.

The return statement causes deactivation (but not necessarily termination, as described above). The return code value on the return statement determines whether the process is willing to be reactivated: a value of 5 or greater forces termination even if data is available.

**API Calls:**

Here are the API calls available to components in THREADS:

```
Create a Packet:
  value = dfscrep(proc_anchor, &ptr, size, type);
Define ports:
  value = dfsdfpt(proc_anchor, port_count, port_tab);
Receive an IP:
  value = dfsrecv(proc_anchor, &ptr, &port_tab[port_no],
           elem_no, &size, &type);
Send an IP:
  value = dfssend(proc_anchor, &ptr, &port_tab[port_no],
           elem_no);
Drop an IP:
  value = dfsdrop(proc_anchor, &ptr);
Pop an IP off the stack:
  value = dfspop(proc_anchor, &ptr, &size, &type);
Push an IP onto the stack:
  value = dfspush(proc_anchor, &ptr);
Close a port element:
  value = dfsclos(proc_anchor, &port_tab[port_no],
```

```
            elem_no);
```

**Parameters:**

Note that the `ptr, size, type` and port table element parameters all have `&`'s attached, except in the case of `dfscrep` , where only `ptr` has it. The `&`'s are required because these parameters may be modified, and C parameters are all passed by value (except for strings and arrays). In some cases a particular service does not set them, but for consistency `&`'s are used for all of them (except `dfscrep` ).

`port_count` and `elem_no` are all binary integer variables (`int`). `size` is binary long. `dfscrep` is restricted to a maximum of 64000 bytes.

- `proc_anchor`: a variable of type anchor (defined in `thxanch.h`)
- `ptr`: a void pointer used to point at IPs
- `size`: a long variable containing the IP's size
- `type`: a null-terminated string of up to 32 chars
- `port_count`: parameter to `dfsdfpt` - must match the dimension of the defined `port_tab` array for this component
- `elem_no`: this specifies the element of the appropriate port to be used in a send, receive or close. These are only required for array-type ports, where they number up from 0. For non-array ports, this parameter must be 0.
- `port_tab`: this parameter is declared as an array of type `port_ent` (see below), where each element corresponds to one of the ports known to the component; the whole structure is passed to `dfsdfpt`, while specific elements of `port_tab` are passed to `send`, `receive` and `close` using `&`.
- `port_name_n`: port name to be used by `dfsdfpt`; the number of these must match the `port_count` parameter

All pointers in the following declarations are "far" pointers.

**Declare for `port_ent`:**

```
 struct _port_ent
  {
     char port_name[32];
```

```
      void *reserved;
      int elem_count;
      int ret_code;
   };
   typedef struct _port_ent port_ent;
```

After a call to `dfsdfpt`, `elem_count` in each `port_ent` instance will be set to the number of connected elements for that port, and `ret_code` will be set to 0 or 2, depending on whether that port was connected or not.

**Declare for anchor (`thxanch.h`):**

```
struct _anchor {
    int (* svc_addr) ();
    void *proc_ptr;
     } ;
 typedef struct _anchor anchor;
```

**Service return codes:**

```
dfscrep:
        0       OK
dfsdfpt:
        0       OK - but some elements of port_tab may have their
                ret_code fields set to 2, meaning "port name not
                found"; a receive or send from or to such a
                port_tab element will result in a return code value
                of 2.
dfsrecv:
        0       OK
        1       port element closed (end of data)
        2       port element not defined or not connected
dfssend:
        0       OK
        1       port element closed
        2       port element not defined or not connected
dfsdrop:
        0       OK
dfspop:
        0       OK
        2       stack empty
dfspush:
        0       OK
```

```
dfsclose:
        0       OK
        1       port element already closed
        2       port element not defined or not connected
```

**Limitations:**

There is a limit of 4000 elements per array port.

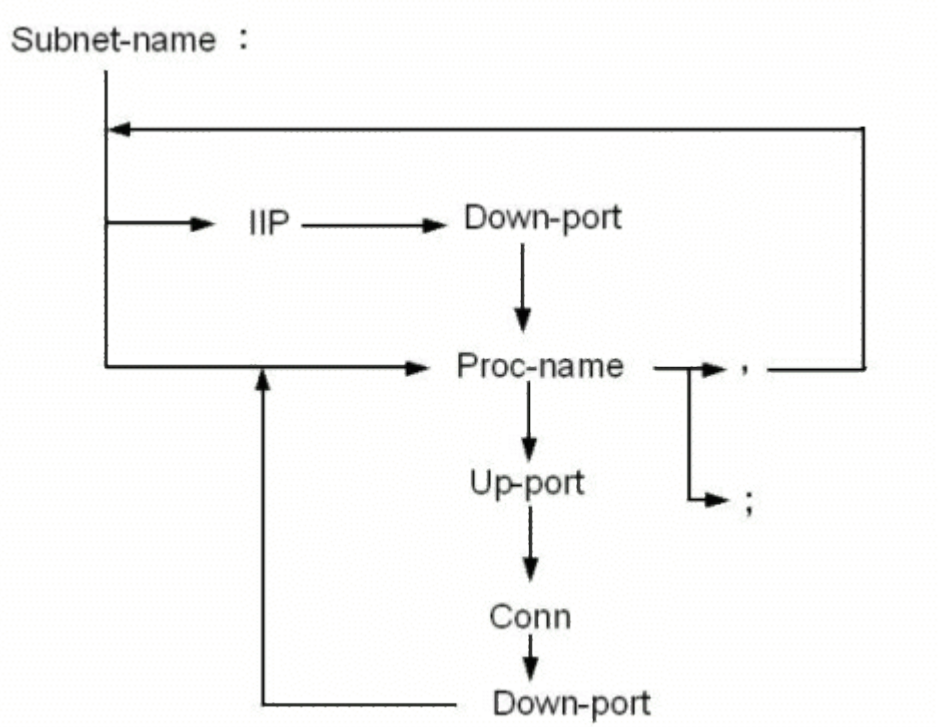
Working storage for a component should not exceed a few thousand bytes (including the working storage of any subroutines it calls). If the component needs more than this, it should use one of C's dynamic allocation functions (e.g. `malloc` or `calloc`) to allocate the additional storage.

**Network Notation**

Networks are defined initially using a free-form notation, described briefly in Chapter 23. For instance,

```
'data.fil'->OPT Reader(THFILERD) OUT -> IN Selector(THSPLIT) MATCH -> ...,
Selector NOMATCH -> ...
```

The general syntax for networks is quite simple, and can be shown as follows (using a variant of the flow notation which has started to become popular for defining syntax):




I have labelled the ports above and below the connection indicator (arrow with optional capacity figure) "up-port" and "down-port" to indicate that they are upstream and downstream, respectively, of the connection.

The main network may be followed by one or more subnets, which have basically the same notation (each one starting with a label and finishing with a semi-colon). However, subnets have to have additional notation describing how its external port names relate to its internal ones. Since this association is like an equivalence, we use the symbol => to indicate this relationship. Thus,

```
, port-name-1 => port-name-2 process-A port-name-3,
```

indicates that `port-name-1` is an external input port of the subnet, while `port-name-2` is the corresponding input port of process-A. Similarly,

```
, port-name-1 process-A port-name-2 => port-name-3,
```

indicates that `port-name-3` is an external output port of the subnet, while `port-name-2` is the corresponding output port of process-A. For an example, see the example of subnets given in Chapter 23.

**Other syntactic elements:**

```
IIP:
          *--------------------------*
          |                          |
          |                          |
   --- ' -*-*- any-char-ex.-quote ---*
            |
            |
            ' ---
```

Two consecutive quotes are taken to mean a single quote.

```
Up-port: Down-port:
   -*- port-name -*- [ --- element-number --- ] -*--*-
    |             |                              |  |
    |             |                              |  |
    |             *------------------------------*  |
    |                                               |
    *----- * --------------------------------------*
```

`element-number` does not apply to the external port names of subnets. The asterisk indicates an automatic port (see Chapter 13).

```
Conn:
   --- -> -*- ( --- capacity --- ) -*-
            |                       |
            |                       |
            *-----------------------*
```

`capacity` does not apply to the external port names of subnets.

```
Proc-name:
   --- process-name -*- ( --- comp-name --- ) -*-*- ? -*-
                     |                          | |     |
                     |                          | |     |
                     *--------------------------* *-----*
```

The component name can be specified on any occurrence of the process name. The question mark indicates that tracing is desired.

In Chapter 13 we alluded to the fact that we can specify that a process "must run at least once" - this is specified by means of an attribute file for the component, which has the name of the component, and an extension of `atr`, e.g. `thcount.atr`. These files must be provided by the supplier of a component, and must have a specific format. So far, only the "must run" attribute has been defined - it is specified by coding one of the character strings `must_run`, `Must_run` or `MUST_RUN`, with optional preceding blanks, in the attribute file for a given component. If no attribute file is found for a component, default values are assumed to apply (the default for the "must run" attribute is "need not run").

As mentioned in Chapter 23, there is also a compiled (or compilable) format for specifying networks - while this is obviously not appropriate for hand-coding by human users, it is easy enough to generate by means of software. It is a data-only C program, and specifies a network, together with all referenced subnets. Its importance is that this will be the source form for networks owned by customers, so THREADS must guarantee that any enhancements to it will be upwards compatible. These guarantees are essentially encoded in the C headers which it uses: `thxiip.h`, `thxanch.h` and `thxscan.h`.

`thxanch.c` has been given above. The other two are defined as follows:

```
thxiip.h
struct _IIP
   {
      int IIP_len;
      char datapart[512];
   };
typedef struct _IIP IIP;
```

```
thxscan.h
   struct _cnxt_ent {
     char upstream_name[32];      /* if 1st char is !,   */
     char upstream_port_name[32]; /* connxn points at IIP */
     int upstream_elem_no;
```

```
     char downstream_name[32];
     char downstream_port_name[32];
     int downstream_elem_no;
     union cnxt_union {IIP *IIPptr; void *connxn;} gen;
     int capacity;
     };
typedef struct _cnxt_ent cnxt_ent;
struct _label_ent {
        char label[32];
     char file[10];
        struct _cnxt_ent *cnxt_ptr;
        struct _proc_ent *proc_ptr;
     char ent_type;
        int proc_count;
        int cnxt_count;
        };
typedef struct _label_ent label_ent;
   struct _proc_ent {
     char proc_name[32];
     char comp_name[10];
     int (*faddr)();
     void *proc_block;
     int label_count;
     int trace;
     int composite;
     int must_run;
     };
typedef struct _proc_ent proc_ent;
```

Note: the above structures are not the internal control blocks of THREADS - they are an encoding of the free-form network specification notation. This separation will allow THREADS to be extended in the future without requiring network definitions to be reprocessed.

# Appendix B: Syntax of JavaFBP (Java Implementation of FBP) and Component API

## Network Definitions

In my book, "Flow-Based Programming", I describe the syntax of the network specifications of various FBP dialects that were in existence when the book was written. JavaFBP, the Java implementation of the FBP concepts, did not exist at that time, so this web page has been added describing the syntax of JavaFBP network definitions.

In JavaFBP we not only code components in Java but also define the networks as Java programs. The source code is on SourceForge under Subversion (SVN) for the Flow-Based Programmming project: SVN for FBP .

There is also a jar file - JavaFBP jar file - on the FBP web site.

One advantage of defining the network as executable code, as compared with other approaches that merely list connections in a language-independent way, is that the network can contain additional logic. This logic then controls the way the network *is defined*, rather than the way it *runs*. Some may regard this as a defect, rather than as an asset, and both views can certainly be defended, but one of the neat things it enables us to do is to adjust multiplexing levels of various structures in the diagram using a table of values (remember the multiplexing example in Sample DrawFlow Diagram). One merely retrieves a value from a table for the degree of multiplexing in a particular structure in the diagram, and this value is then used both as the index of a loop invoking the `connect` statement, and as the index for the elements of an array-type port (see below for both of these terms).

Since the way the syntax relates to the underlying diagram may not be all that clear, a brief description is in order. At the end of this page, I have given an extremely simple JavaFBP component.

Any JavaFBP network definition starts as follows:

```
public class xxxxxx extends Network {
    protected void define() {
```

where `xxxxxx` is the Network name, including the usual imports, copyright statements, etc. Of course you will have to import classes for JavaFBP objects, such as Network and Component, as well as any JavaFBP "verb" classes you may be using.

The network definition is terminated with:

```
}
        public static void main(String[] argv) throws Exception {
                new xxxxxx().go();
                }
}
```

In between the beginning and the ending statements defining the network, you specify a list of connections, using the following methods, which I will refer to as "clauses":

- `component` - define an instance of a component (an FBP "process")
- `connect` - define a connection
- `initialize` - define a connection including an Initial Information Packet (IIP)
- `port` - define a port on a process

Every component instance *must have a unique character string* identifying it, which allows other component instances or initial information packets (IIPs) to be attached to it via a connection.

The following method call:

```
component("xxxx")
```

returns a reference to a component instance. The first reference to this particular component instance must specify the component class to be executed by that occurrence. This is done by coding

```
component("xxxx", cccc.class)
```

where `cccc` is the name of the Java module to be executed.

Similarly, a port is identified by a `port` clause, e.g. `port("xxxx")`.

A port may be an array-type port, in which case the port clauses referring to its elements have index values, as follows:

```
port("xxxx",n)
```

where "n" runs up monotonically from 0. Each element of the port array will be connected to a different component occurrence or IIP.

A `connect` or `initialize` clause may contain the relevant `component` clauses, together with their corresponding `port` clauses, embedded within it, as e.g.

```
    connect(component("Read", ReadText.class),
        port("OUT"),
        component("Splitter1", Splitter1.class),
        port("IN"));
```

or the `connect` and `component` portions may be in separate statements, provided `component` precedes any `connects` that reference it.

A `connect` contains:

- "from" `component` clause
- "from" `port` clause
- "to" `component` clause
- "to" `port` clause

Optionally a `connect` may have a fifth parameter: the connection capacity, specified as an `int`. If this is omitted, the default value is used: 1 for testing, or 10 for production (this currently has to be changed by hand in the Network.class).

If an asterisk (`*`) is specified for the "from" port, this is called an "automatic output port", and indicates a signal generated when the "from" component instance terminates (actually the port is just closed, so no packet has to be disposed of).

If an asterisk is specified for the "to" port, this is called an "automatic input port", and indicates a *delay* - the "to" component instance does not start until a signal or a close is received at this port.

If `*SUBEND` is specified as a port name on a subnet, a packet containing null is emitted at this port every time the subnet deactivates, i.e. all the contained components terminate. It doesn't have to be named in the port metadata. This null packet is emitted for all activations, including the last one.

An `initialize` clause contains:

- a reference to *any* object

- a `component` clause
- a `port` clause

as e.g.

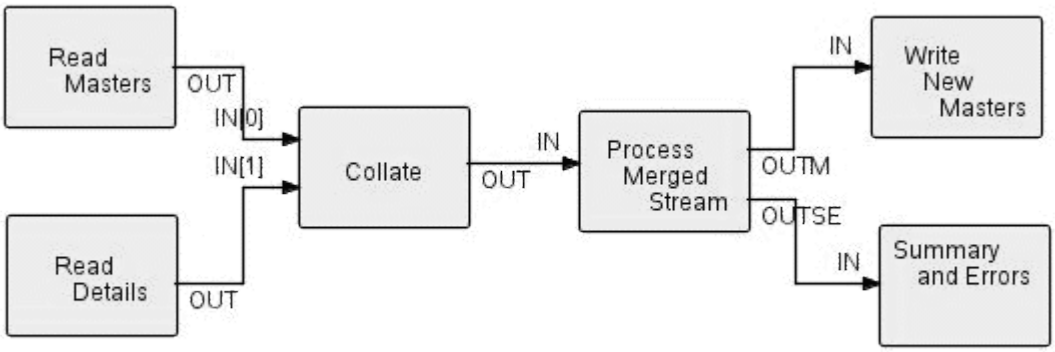
```
    initialize(new FileReader(
             "c:\\com\\jpmorrsn\\eb2engine\\test\\data\\myXML3.txt"),
          component("Read"),
          port("SOURCE"));
```

However, it has been recommended that IIPs should be strings, rather than arbitrary objects, to facilitate future graphical management of networks.

One last point: any number of "from" ports can be connected to a single "to" port; only one "to" port can ever be connected to a given "from" port.

## Sample Network

Let us code up a network implementing the following picture:




First list the `component` clauses, together with the component classes they are to execute (assuming that component classes have been written to execute the various nodes of the diagram), e.g.:

```
component("Read Masters",Read.class)
component("Read Details",Read.class)
```

```
component("Collate",Collate.class)
component("Process Merged Stream",Proc.class)
component("Write New Masters",Write.class)
component("Summary & Errors",Report.class)
```

Now these `component` clauses may either be made into separate statements or they can be imbedded into the `connect` statements that follow. Here are the connections in the diagram, *without* imbedded component clauses:

```
connect(component("Read Masters"),port("OUT"),component("Collate"),
   port("IN",0));                        // array port
connect(component("Read Details"),port("OUT"),component("Collate"),
   port("IN",1));                        // array port
connect(component("Collate"),port("OUT"),
   component("Process Merged Stream"), port("IN"));
connect(component("Process Merged Stream"),port("OUTM"),
   component("Write New Masters"),port("IN"));
connect(component("Process Merged Stream"),port("OUTSE"),
   component("Summary & Errors"),port("IN"));
```

Each item in this list is a separate Java statement.

We can now add the class designation to the first `component` clause referencing a particular component occurrence, giving the following:

```
connect(component("Read Masters",Read.class),port("OUT"),
   component("Collate",Collate.class), port("IN",0));   // array port
connect(component("Read Details",Read.class),port("OUT"),
   component("Collate"),port("IN",1));                  // array port
connect(component("Collate"),port("OUT"),
   component("Process Merged Stream",Proc.class),port("IN"));
connect(component("Process Merged Stream"),port("OUTM"),
   component("Write New Masters",Write.class),port("IN"));
connect(component("Process Merged Stream"),port("OUTSE"),
   component("Summary & Errors",Report.class),port("IN"));
```

Now "Read Masters" and "Read Details" use the same Java class, so we need some way to indicate the name of the file that each is going to read. This is done using Initial Information Packets (IIPs). In this case they might usefully specify FileReader objects, so we need to add two `initialize` clauses, as follows:

```
```

```
  initialize(new FileReader("c:\\mastfile"),
                        component("Read Masters"),
                        port("SOURCE"));
  initialize(new FileReader("c:\\detlfile"),
                        component("Read Details"),
                        port("SOURCE"));
```

Note that, since both "Read" component occurrences use the same class code, they naturally have the same port names - of course, the ports are attached to different IIPs.

Remember that back-slashes have to be doubled in Java character strings!

"Write New Masters" will have to have an IIP to specify the output destination - perhaps:

```
  initialize(new FileWriter("c:\\newmast"),
                        component("Write New Masters"),
                        port("DESTINATION"));
```

Note also that this IIP is not a *destination* for the Writer - it is an object *used* by this component occurrence so that the latter can figure out where to send data to.

Add the beginning and ending statements, and you're done! The actual sequence of `connect` and `initialize` statements is irrelevant.

Here is the final result:

```
public class xxxxxx extends Network {
  protected void define() {
    connect(component("Read Masters",Read.class),port("OUT"),
      component("Collate",Collate.class),port("IN",0)); // array port
    connect(component("Read Details",Read.class),port("OUT"),
      component("Collate"),port("IN",1));// array port
    connect(component("Collate"),port("OUT"),
      component("Process Merged Stream",Proc.class),port("IN"));
    connect(component("Process Merged Stream"),port("OUTM"),
      component("Write New Masters",Write.class),port("IN"));
    connect(component("Process Merged Stream"),port("OUTSE"),
      component("Summary & Errors",Report.class),port("IN"));
    initialize(new FileReader("c:\\mastfile"),
      component("Read Masters"),
      port("SOURCE"));
    initialize(new FileReader("c:\\detlfile"),
      component("Read Details"),
```

```
      port("SOURCE"));
    initialize(new FileWriter("c:\\newmast"),
      component("Write New Masters"),
      port("DESTINATION"));
    }
  public static void main(String[] argv) throws Exception{
            // as of JavaFBP-2.4
     new xxxxxx().go();
  }
}
```

## Alternative (Simplified) Notation (JavaFBP-2.0+)

In the latest release of JavaFBP, we have introduced a new, simplified notation, in addition to that shown above. In this notation `connect` specifies two character strings, and `initialize` specifies an object and a character string. In both cases, the second character string specifies a combination of component and port, with the two parts separated by a period. Array port indices, if required, are specified using square brackets, e.g.

`"component.port[3]"`

The old port notation will still be supported, but is only really needed when the port index is a variable. When debugging, it will be noted that the square bracket notation is used in trace lines, even when it was not used in the network definition.

Component names must of course not include periods or most special characters, but they may include blanks, numerals, hyphens and underscores, and they must be associated with their implementing class using a (preceding) `component` statement.

Here is the above network using the new notation:

```
public class xxxxxx extends Network {
 protected void define() {
   component("Read Masters",Read.class);
   component("Read Details",Read.class);
   component("Collate",Collate.class);
   component("Process Merged Stream",Proc.class);
   component("Write New Masters",Write.class);
   component("Summary & Errors",Report.class);
   connect("Read Masters.OUT", "Collate.IN[0]");
   connect("Read Details.OUT", "Collate.IN[1]");
   connect("Collate.OUT"), "Process Merged Stream.IN");
```

```
   connect("Process Merged Stream.OUTM", "Write New Masters.IN");
   connect("Process Merged Stream.OUTSE", "Summary & Errors.IN");
   initialize(new FileReader("c:\\mastfile"), "Read Masters.SOURCE");
   initialize(new FileReader("c:\\detlfile"), "Read Details.SOURCE");
   initialize(new FileWriter("c:\\newmast"),
              "Write New Masters.DESTINATION");
   }

public static void main(String[] argv) throws Exception {
                 // as of JavaFBP-2.4
        new xxxxxx().go();
   }
  }
```

Here is a network example showing how variable port numbers can be used with the LoadBalance function to define an (admittedly fairly trivial) self-balancing network. This also shows a slightly different way of specifying the `define` function.

```
public class TestLoadBalancer {
  public static void main(final String[] args) {
    try {
      new Network() {
        @Override
        protected void define() {
          int multiplex_factor = 10;
          component("generate", Generate.class);
          component("display", WriteToConsole.class);
          component("lbal", LoadBalance.class);
          connect("generate.OUT", "lbal.IN");
          initialize("100 ", component("generate"), port("COUNT"));
          for (int i = 0; i < multiplex_factor; i++) {
            connect(component("lbal"), port("OUT", i),
                component("passthru" + i, Passthru.class), port("IN"));
            connect(component("passthru" + i), port("OUT"), "display.IN");
          }
        }
      }.go();
    } catch (Exception e) {
      System.err.println("Error:");
      e.printStackTrace();
    }
  }
}
```

## Sample JavaFBP Component

**A more complete description of the API is given in the next section.**

This component generates a stream of 'n' IPs, where 'n' is specified in an InitializationConnection (specified by an `initialize` clause in the foregoing). Each IP just contains an arbitrary string of characters, in order to illustrate the concept. Of course any copyright information included is up to the developer.

```
package com.jpmorrsn.fbp.components;

import com.jpmorrsn.fbp.engine.*;

/** Component to generate a stream of 'n' packets, where 'n' is
* specified in an InitializationConnection.
*/

@OutPort(value = "OUT", description = "Generated stream",
       type = String.class)
@ComponentDescription(
       "Generates stream of packets under control of a counter")
@InPort(value = "COUNT",
       description = "Count of packets to be generated",
       type = String.class)
public class Generate extends Component {
static final String copyright = "Copyright .....";
        OutputPort outport;
        InputPort count;

  @Override
  protected void execute() {
    Packet ctp = count.receive();
    if (ctp == null) {
      return;
    }
    count.close();
    String cti = (String) ctp.getContent();
    cti = cti.trim();
    int ct = 0;
    try {
      ct = Integer.parseInt(cti);
    } catch (NumberFormatException e) {
      e.printStackTrace();
    }
    drop(ctp);
    for (int i = 0; i < ct; i++) {
```

```
      int j = ct - i;
      Integer j2 = new Integer(j);
      String s = j2.toString();
      if (j < 10) {
        s = "0" + s;
      }
      if (j < 100) {
        s = "0" + s;
      }
      s = s + "abc";
      Packet p = create(s);
      outport.send(p);
    }
  }
 /* As of JavaFBP-2.3, the information in introspect() is now covered
  by the metadata, which has at the same time been expanded to add
  descriptive information.
  introspect() is no longer needed - it will be ignored if present.
        public Object[] introspect() {  // was 'private' - must be 'public'
                return new Object[] {
                "generates a set of Packets under control of a counter" ,
                "OUT", "output", String.class,
                        "lines read",
                "COUNT", "parameter", Integer.class,
                        "Count of number of entities to be generated"};
                }
   */
@Override
        protected void openPorts() {
                outport = openOutput("OUT");
                count = openInput("COUNT");

                }
}
```

The scheduling rules for most FBP implementations are described in the chapter of my book called Scheduling Rules.

The previous Java implementation of FBP (javaFBP-1.5.3) presents an IIP to a component *once per activation*. This has been changed as of JavaFBP-2.0 to *once per invocation*. In practice this will only affect "non-loopers" (components that get reactivated multiple times).

There are a few minor changes to component code as of JavaFBP-2.0:

- As good programming practice, we now feel that IIP ports should be closed after a receive has been executed, in case it is attached to an upstream component (rather than an IIP),

and that component mistakenly sends more than one IP - this statement has accordingly been added to the above example.

- The drop statement now takes the packet as a parameter, rather than being a method of Packet.
- The send is now unconditional - it either works or crashes.
- We are adding a "long wait" state to components, specifying a timeout value in seconds. This is coded as follows:

```
double _timeout = 2;    // 2 secs
     ....
 longWaitStart(_timeout);

//   activity taking time goes here
  longWaitEnd();
```

- Typically, the timeout value is given a default value in the code, and overridden (if desired) by an IIP.
- While the component in question is executing the activity taking time, its state will be set to "long wait". If one or more components are in "long wait" state while all other components are suspended or not started, this situation is not treated as a deadlock. However, if one of the components exceeds its timeout value, an error will be reported (Complain).

and a major change - metadata, as shown in the above component. This works as follows:

- Input and output port names will be coded on components using Java 5.0 "attribute" notation. This metadata can be used to do analysis of networks without having to actually execute the components involved. Here is an example of the attributes for the "Collate" component:

```
@OutPort("OUT")
@InPorts({
  @InPort("CTLFIELDS"),
  @InPort(value = "IN", arrayPort = true)
  })
public class Collate extends Component {
```

- Note that, as Java metadata does not (as far as I know) support multiple entries with the same name, we have provided the additional metadata terms @InPorts and @OutPorts. When only one is needed (as for @Outport in this example) the "plural" term can be omitted.
- As shown above for "CTLFIELDS", when no attributes are needed within an @InPort or @OutPort statement, the short notation (no "value" clause) can be used.

- Input ports do not necessarily have to be connected, even though attributes are specified for them; output ports, however, must be.
- MustRun is also specified as metadata, rather than as an interface, as it was in version 1.5.3, i.e.

```
@MustRun
```

A new service has been added for component code as of JavaFBP-2.2:

- `<output port name>.isConnected` returns `boolean`

To support `isConnected`, a new metadata attribute called `optional` has been added to @OutPort, e.g.

- `@OutPort(value = "OUT", optional = true)`

## JavaFBP Component API

```
Component Metadata:
Note: when "value" is the only parameter, the "short" form
   (see above) can be used
@ComponentDescription
  parameters:
    - value (String)
@InPort
  parameters:
    - value (String)
    - arrayPort (boolean)
    - description (String)
    - type (class)
@OutPort
  parameters:
    - value (String)
    - arrayPort (boolean)
    - description (String)
    - type (class)
    - optional (boolean)
@InPorts
  parameter: list of @InPort references, e.g. @InPorts( { @InPort("IN"),
     @InPort("TEST") })
@OutPorts
  parameter: list of @OutPort references, e.g. @OutPorts( { @OutPort("ACC"),
     @OutPort("REJ") })
@MustRun
```

```
@Priority(Thread.MAX_PRIORITY)   // default is NORM_PRIORITY
```

Packet class:

```
/**
* A Packet may either contain an Object, when type is NORMAL,
* or a String, when type is not NORMAL.  The latter case
* is used for things like open and close brackets (where the
* String will be the name of a group. e.g. accounts)
**/
Object getAttribute(String key);  /* key accesses a specific attribute */
Object getContent();  /* returns null if type <> NORMAL */
```

Component class:

```
/**
* All verbs must extend this class, defining its two abstract methods:
* openPorts, and execute.
**/
Packet p = create(Object o);
Packet p = create(Packet.type (int) t, String s);
drop(Packet p);   //  Note this change!
longWaitStart(double interval);   // in seconds
longWaitEnd();
/** 3 stack methods - as of JavaFBP-2.3
**/
push (Packet p);
Packet p = pop();  // return null if empty
int stackSize();
```

InputPort interface:

```
Packet = receive();
void close();
```

OutputPort class:

```
void send(Packet packet);
boolean isConnected();  // as of 2.2
void close();
```

Appendix B: Syntax of JavaFBP (Java Implementation of FBP) and Component API

# Appendix C: Syntax of C#FBP (C# Implementation of FBP) and Component API

## Network Definitions

In my book, "Flow-Based Programming", I describe the syntax of the network specifications of various FBP dialects that were in existence when the book was written. C#FBP, the C# implementation of the FBP concepts, did not exist at that time, so this web page has been added describing the syntax of C#FBP network definitions.

In C#FBP we not only code components in C# but also define the networks as C# programs. The source code is on SourceForge under Subversion (SVN) for the SourceForge project Flow-Based Programming .

There is also a zip file - C#FBP zip file - on the FBP web site.

One advantage of defining the network as executable code, as compared with other approaches that merely list connections in a language-independent way, is that the network can contain additional logic. This logic then controls the way the network *is defined*, rather than the way it *runs*. Some may regard this as a defect, rather than as an asset, and both views can certainly be defended, but one of the neat things it enables us to do is to adjust multiplexing levels of various structures in the diagram using a table of values (remember the multiplexing example in Sample DrawFlow Diagram). One merely retrieves a value from a table for the degree of multiplexing in a particular structure in the diagram, and this value is then used both as the index of a loop invoking the `connect` statement, and as the index for the elements of an array-type port (see below for both of these terms).

Since the way the syntax relates to the underlying diagram may not be all that clear, a brief description is in order. At the end of this page, I have given an extremely simple C#FBP component.

Any C#FBP network definition starts as follows:

```
using System;
using FBPComponents;
using FBPLib;
namespace nnnnnnnnn
{
    public class xxxxxx : Network
    {
        public override void Define()  {
```

where `xxxxxx` is the Network name, including the usual usings, copyright statements, namespace specification, etc.

The network definition is terminated with:

```
}
        internal static void Main(String[] argv)
        {
            new xxxxx().Go();
        }
    }
```

In between the beginning and the ending statements defining the network, you specify a list of connections, using the following methods, which I will refer to as "clauses":

- `Component` - define an instance of a component (an FBP "process")
- `Connect` - define a connection
- `Initialize` - define a connection including an Initial Information Packet (IIP)
- `Port` - define a port on a process

Every component instance *must have a unique character string* identifying it, which allows other component instances or initial information packets (IIPs) to be attached to it via a connection.

The following method call:

```
Component("xxxx")
```

returns a reference to a component instance. The first reference to this particular component instance must specify the component class to be executed by that occurrence. This is done by coding

```
Component("xxxx", typeof(cccc))
```

where `cccc` is the name of the C# module to be executed.

Similarly, a port is identified by a `Port` clause, e.g. `Port("xxxx")`.

A port may be an array-type port, in which case the port clauses referring to its elements have index values, as follows:

```
Port("xxxx",n)
```

where "n" runs up monotonically from 0. Each element of the port array will be connected to a different component occurrence or IIP.

A `Connect` or `Initialize` clause may contain the relevant `Component` clauses, together with their corresponding `Port` clauses, embedded within it, as e.g.

```
    Connect(Component("Read", typeof(ReadText)),
        Port("OUT"),
        Component("Splitter1", typeof(Splitter1)),
        Port("IN"));
```

or the `Connect` and `Component` portions may be in separate statements, provided `Component` precedes any `Connect`s that reference it.

A `Connect` contains:

- "from" `Component` clause
- "from" `Port` clause
- "to" `Component` clause
- "to" `Port` clause

Optionally a `Connect` may have a fifth parameter: the connection capacity, specified as an `int`. If this is omitted, the default value is used: 1 for testing, or 10 for production (this currently has to be changed by hand in the Network.class).

If an asterisk (`*`) is specified for the "from" port, this is called an "automatic output port", and indicates a signal generated when the "from" component instance terminates (actually the port is just closed, so no packet has to be disposed of).

If an asterisk is specified for the "to" port, this is called an "automatic input port", and indicates a *delay* - the "to" component instance does not start until a signal or a close is received at this port.

If `*SUBEND` is specified as a port name on a subnet, a packet containing null is emitted at this port every time the subnet deactivates, i.e. all the contained components terminate. It doesn't have to be named in the port metadata. This null packet is emitted for all activations, including

the last one.

An `Initialize` clause contains:

- a reference to *any* object
- a `Component` clause
- a `Port` clause

as e.g.

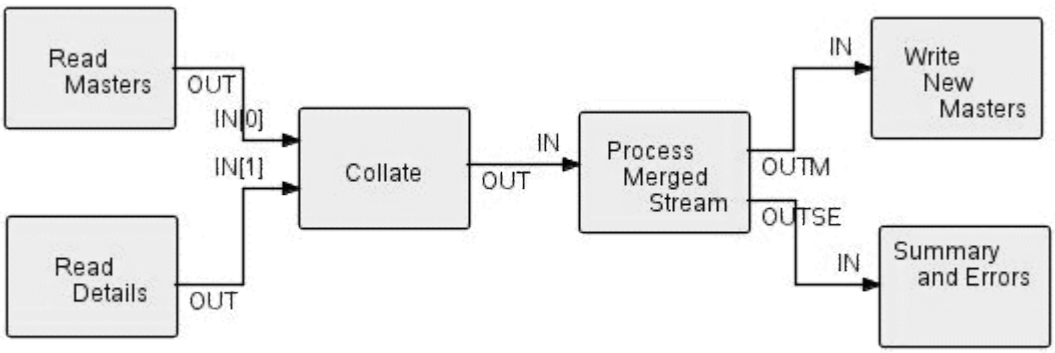
```
    Initialize(new StreamReader(
              @"c:\com\jpmorrsn\eb2engine\test\data\myXML3.txt"),
          Component("Read"),
          Port("SOURCE"));
```

However, it has been recommended that IIPs should be strings, rather than arbitrary objects, to facilitate future graphical management of networks.

One last point: any number of "from" ports can be connected to a single "to" port; only one "to" port can ever be connected to a given "from" port.

## Sample Network

Let us code up a network implementing the following picture:




First list the `Component` clauses, together with the component classes they are to execute (assuming that component classes have been written to execute the various nodes of the

diagram), e.g.:

```
Component("Read Masters",typeof(Read))
Component("Read Details",typeof(Read))
Component("Collate",typeof(Collate))
Component("Process Merged Stream",typeof(Proc))
Component("Write New Masters",typeof(Write))
Component("Summary & Errors",typeof(Report))
```

Now these `Component` clauses may either be made into separate statements or they can be imbedded into the `Connect` statements that follow. Here are the connections in the diagram, *without* imbedded Component definitions:

```
Connect(Component("Read Masters"),Port("OUT"),Component("Collate"),
   Port("IN",0));                       // array port
Connect(component("Read Details"),Port("OUT"),Component("Collate"),
   Port("IN",1));                       // array port
Connect(component("Collate"),Port("OUT"),Component("Process Merged Stream"),
   Port("IN"));
Connect(Component("Process Merged Stream"),Port("OUTM"),
   Component("Write New Masters"),Port("IN"));
Connect(Component("Process Merged Stream"),Port("OUTSE"),
   Component("Summary & Errors"),Port("IN"));
```

Each item in this list is a separate C# statement.

We can now add the class designation to the first `Component` clause referencing a particular component occurrence, giving the following:

```
Connect(Component("Read Masters",typeof(Read)),Port("OUT"),
   Component("Collate",typeof(Collate)), Port("IN",0));   // array port
Connect(Component("Read Details",typeof(Read)),Port("OUT"),
   Component("Collate"),Port("IN",1));                  // array port
Connect(Component("Collate"),Port("OUT"),
   Component("Process Merged Stream",typeof(Proc)),Port("IN"));
Connect(Component("Process Merged Stream"),Port("OUTM"),
   Component("Write New Masters",typeof(Write)),Port("IN"));
Connect(Component("Process Merged Stream"),Port("OUTSE"),
   Component("Summary & Errors",typeof(Report)),Port("IN"));
```

Now "Read Masters" and "Read Details" use the same C# class, so we need some way to indicate

the name of the file that each is going to read. This is done using Initial Information Packets (IIPs). In this case they might usefully specify StreamReader objects, so we need to add two `Initialize` clauses, as follows:

```
  Initialize(new StreamReader(@"c:\mastfile"),
                        Component("Read Masters"),
                        Port("SOURCE"));
  Initialize(new StreamReader(@"c:\detlfile"),
                        Component("Read Details"),
                        Port("SOURCE"));
```

Note that, since both "Read" component occurrences use the same class code, they naturally have the same port names - of course, the ports are attached to different IIPs.

Remember that back-slashes have to be doubled in C# character strings, unless you precede the string with an @-sign.

"Write New Masters" will have to have an IIP to specify the output destination - perhaps:

```
  Initialize(new StreamWriter(@"c:\newmast"),
                        Component("Write New Masters"),
                        Port("DESTINATION"));
```

Note also that this IIP is not a *destination* for the Writer - it is an object *used* by this component occurrence so that the latter can figure out where to send data to.

Add the beginning and ending statements, and you're done! The actual sequence of `Connect` and `Initialize` statements is irrelevant.

Here is the final result:

```
using System;
using FBPComponents;
using FBPLib;
namespace nnnnnnnnn
{
    public class xxxxxx : Network
    {
    public override void Define()  {
     Connect(Component("Read Masters",typeof(Read)),Port("OUT"),
     Component("Collate",typeof(Collate)), Port("IN",0));   // array port
     Connect(Component("Read Details",typeof(Read)),Port("OUT"),
     Component("Collate"),Port("IN",1));                  // array port
```

```
    Connect(Component("Collate"),Port("OUT"),
    Component("Process Merged Stream",typeof(Proc)),Port("IN"));
    Connect(Component("Process Merged Stream"),Port("OUTM"),
    Component("Write New Masters",typeof(Write)),Port("IN"));
    Connect(Component("Process Merged Stream"),Port("OUTSE"),
    Component("Summary & Errors",typeof(Report)),Port("IN"));
    Initialize(new StreamReader(@"c:\mastfile"),
                        Component("Read Masters"),
                        Port("SOURCE"));
    Initialize(new StreamReader(@"c:\detlfile"),
                        Component("Read Details"),
                        Port("SOURCE"));
    Initialize(new StreamWriter(@"c:\newmast"),
                        Component("Write New Masters"),
                        Port("DESTINATION"));

}
        internal static void Main(String[] argv)
        {
            new xxxxx().Go();
        }
    }
```

## Alternative (Simplified) Notation

In C#FBP there is a simplified notation, in addition to that shown above. In this notation `Connect` specifies two character strings, and `Initialize` specifies an object and a character string. In both cases, the second character string specifies a combination of component and port, with the two parts separated by a period. Array port indices, if required, are specified using square brackets, e.g.

```
"component.port[3]"
```

The old port notation will still be supported, but is only really needed when the port index is a variable. When debugging, it will be noted that the square bracket notation is used in trace lines, even when it was not used in the network definition.

Component names must of course not include periods or most special characters, but they may include blanks, numerals, hyphens and underscores, and they must be associated with their implementing class using a (preceding) `Component` statement.

Here is the above network using the new notation:

```
 using System;
using FBPComponents;
using FBPLib;
namespace nnnnnnnnn
{
    public class xxxxxx : Network
    {
    public override void Define()  {
   Component("Read Masters",typeof(Read));
   Component("Read Details",typeof(Read));
   Component("Collate",typeof(Collate));
   Component("Process Merged Stream",typeof(Proc));
   Component("Write New Masters",typeof(Write));
   Component("Summary & Errors",typeof(Report));
   Connect("Read Masters.OUT", "Collate.IN[0]");
   Connect("Read Details.OUT", "Collate.IN[1]");
   Connect("Collate.OUT"), "Process Merged Stream.IN");
   Connect("Process Merged Stream.OUTM", "Write New Masters.IN");
   Connect("Process Merged Stream.OUTSE", "Summary & Errors.IN");
   Initialize(new FileReader("c:\\mastfile"), "Read Masters.SOURCE");
   Initialize(new FileReader("c:\\detlfile"), "Read Details.SOURCE");
   Initialize(new FileWriter("c:\\newmast"), "Write New Masters.DESTINATION");
   }
   internal static void Main(String[] argv)
        {
            new xxxxx().Go();
        }
   }
  }
```

Here is a network example showing how variable port numbers can be used with the LoadBalance function to define an (admittedly fairly trivial) self-balancing network.

```
 using System;
using FBPComponents;
using FBPLib;
namespace nnnnnnnnn
{
    public class TestLoadBalancer : Network
    {
    public override void Define() {
          int multiplex_factor = 10;
          Component("generate", typeof(Generate));
          Component("display", typeof(WriteToConsole));
          Component("lbal", typeof(LoadBalance));
          Connect("generate.OUT", "lbal.IN");
```

```
          Initialize("100 ", Component("generate"), Port("COUNT"));
          for (int i = 0; i < multiplex_factor; i++) {
            Connect(Component("lbal"), Port("OUT", i),
                  Component("passthru" + i, typeof(Passthru)), Port("IN"));
            Connect(Component("passthru" + i), Port("OUT"), "display.IN");
          }
        }

    internal static void Main(String[] argv)
        {
            new TestLoadBalancer().Go();
        }
    }
}
```

## Sample C#FBP Component

**A more complete description of the API is given in the next section.**

This component generates a stream of 'n' IPs, where 'n' is specified in an InitializationConnection (specified by an `Initialize` clause in the foregoing). Each IP just contains an arbitrary string of characters, in order to illustrate the concept. Of course any copyright information included is up to the developer.

```
using System;
using FBPLib;
namespace FBPComponents
{
    /** Component to generate a stream of 'n' packets, where 'n' is
    * specified in an InitializationConnection.
    */
    [InPort("COUNT", description="Number of packets", type=typeof(System.String))]
    [OutPort("OUT")]
    [ComponentDescription("Generate stream of packets based on count")]
    public class Generate : Component
    {
        internal static string _copyright =
            "Copyright ....";
        OutputPort _outport;
        IInputPort _count;
        public override void Execute() /* throws Throwable */ {
            Packet ctp = _count.Receive();
            string param = ctp.Content.ToString();
            Int32 ct = Int32.Parse(param);
```

```
            Drop(ctp);
            _count.Close();
            for (int i = ct; i > 0; i--)
            {
                string s = String.Format("{0:d6}", i) + new String('a', 1000);
                Packet p = Create(s);
                _outport.Send(p);
            }
            // output.close();
            // terminate();
        }
        /*
        As of C# 2.0, Introspect has been replaced by the metadata
        public override System.Object[] Introspect()
        {
            return new Object[] {
                "generates a set of Packets under control of a counter" ,
                "OUT", "output", Type.GetType("System.String"),
                        "lines generated",
                "COUNT", "parameter", Type.GetType("System.String"),
                        "Count of number of entities to be generated"};
        }
        */
        public override void OpenPorts()
        {
            _outport = OpenOutput("OUT");

            _count = OpenInput("COUNT");

        }
    }
}
```

The scheduling rules for most FBP implementations are described in the chapter of my book called Scheduling Rules.

The previous C# implementation of FBP (C#FBP-1.5.3) presents an IIP to a component *once per activation*. This has been changed in the latest implementation (C#FBP-2.0) to *once per invocation*. In practice this will only affect "non-loopers" (components that get reactivated multiple times).

There are a few minor changes to component code for C#FBP-2.0:

- As good programming practice, we now feel that IIP ports should be closed after a Receive has been executed, in case it is attached to an upstream component (rather than an IIP), and that component mistakenly sends more than one IP - this statement has

accordingly been added to the above example.

- The Drop statement now takes the packet as a parameter, rather than being a method of Packet.
- The Send is now unconditional - it either works or crashes.
- We are adding a "long wait" state to components, specifying a timeout value in seconds. This is coded as follows:

```
double _timeout = 2;    // 2 secs
    ....
LongWaitStart(_timeout);

//   activity taking time goes here
LongWaitEnd();
```

- Typically, the timeout value is given a default value in the code, and overridden (if desired) by an IIP.
- While the component in question is executing the activity taking time, its state will be set to "long wait". If one or more components are in "long wait" state while all other components are suspended or not started, this situation is not treated as a deadlock. However, if one of the components exceeds its timeout value, an error will be reported (Complain).

and a major change - metadata, as shown in the above component. This works as follows:

- Input and output port names will be coded on components using C# 5.0 "attribute" notation. This metadata can be used to do analysis of networks without having to actually execute the components involved. Here is an example of the attributes for the "Collate" component:

```
[InPort("CTLFIELDS")]
[InPort("IN", arrayPort = true)]
[OutPort("OUT")]
 [ComponentDescription("Collates input streams at array port IN and
sends them to port OUT")]
public class Collate : Component {
```

- Input ports do not necessarily have to be connected, even though attributes are specified for them; output ports, however, must be.
- MustRun is also specified as metadata, rather than as an interface, as it was in version 1.5.3, i.e.

```
[MustRun]
```

A new service has been added for component code for C#FBP-2.2:

- `<output port name>.IsConnected` returns `bool`

To support `IsConnected`, a new metadata attribute called `optional` has been added to OutPort, e.g.

- `[OutPort("OUT",optional=true)]`

## C#FBP Component API

```
Component Metadata:
[ComponentDescription]
  parameters:
    - string
[InPort]
  parameters:
    - string
    - arrayPort (bool)
    - description (string)
    - type (typeof(...))
[OutPort]
  parameters:
    - string
    - arrayPort (bool)
    - description (string)
    - type (typeof(...))
    - optional (bool)
[MustRun]
[Priority(ThreadPriority.Highest)]   // default is ThreadPriority.Lowest
```

```
Packet class:
/**
* A Packet may either contain an Object, when type is NORMAL,
* or a String, when type is not NORMAL.  The latter case
* is used for things like open and close brackets (where the
* String will be the name of a group. e.g. accounts)
**/
Dictionary Attributes;  /* returns attributes */
Object Content;         /* returns null if type <> Normal */
```

Component class:

```
/**
* All verbs must extend this class, defining its two abstract methods:
* openPorts, and execute.
**/
Packet p = Create(Object o);
Packet p = Create(Packet.Types type, string s);
Drop(Packet p);   //  Note this change!
LongWaitStart(double interval);   // in seconds
LongWaitEnd();
/** 3 stack methods - as of JavaFBP-2.3
**/
Push (Packet p);
Packet p = Pop();  // return null if empty
int StackSize();
```

IInputPort interface:

```
Packet = Receive();
void Close();
```

OutputPort class:

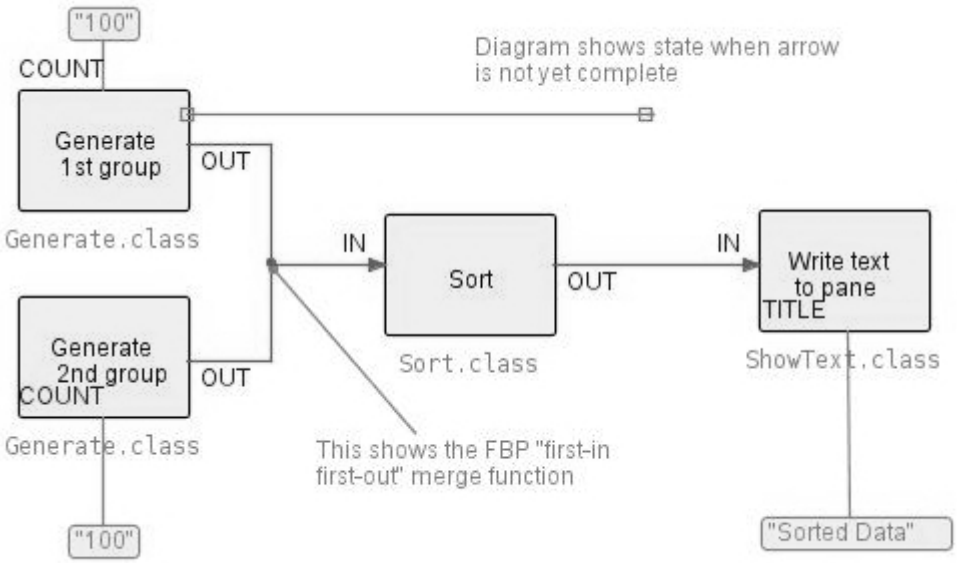
```
void Send(Packet packet);
bool IsConnected();  // as of 2.2
void Close();
```

# Appendix D: FBP Drawing Tool (DrawFBP)

Several years ago, a picture-drawing tool which supports many of the concepts of FBP was written in C++ for Windows. It can still be obtained by clicking here: DrawFBP-C++ (you may have to shift and click), and is also available on SourceForge, but it has now been superseded by a Java version, which can be obtained by clicking here: DrawFBP Installer (you may have to shift and click), and this is also available on SourceForge - look for the latest version. It can also be downloaded and executed by clicking on JWS version of DrawFBP.

DrawFBP does not attempt to generate diagrams from text - we have seen too many failed attempts at this - instead it allows the diagrammer to lay out the flow as desired, capturing the information from the diagram, including x-y coordinates of blocks and line bends, in an XML file. From this, the diagram can be rebuilt, plus it captures the relationships between processes and their connections, so it has enough information to generate the lists of connections used by FBP (or FBP-like) schedulers. The new version of DrawFBP can generate working FBP networks, prompting the user to fill in any needed information.

DrawFBP also supports step-wise decomposition, and allows files and reports to be added symbolically to a design. Here is a sample diagram, showing an arrow in process of being drawn:




A Help facility is available, based on Java SE Desktop Technologies JavaHelp System - this function does not require access to the Internet.

DrawFBP can be executed directly from the author's web site, using Java Web Start (JWS), by entering `http://www.jpaulmorrison.com/graphicsstuff/DrawFBP.jnlp` into the command line of your favorite browser. Alternatively it can be downloaded as a jar file from `http://www.jpaulmorrison.com/graphicsstuff/DrawFBP-x.x.jar`, where `x.x` is the latest release.

# Glossary of Terms used by FBP and Related Concepts

| | |
|---|---|
| 4GL | 4th Generation Language, typically generating HLL statements |
| AMPS | Advanced Modular Processing System - first version of FBP used for production work (still in use at a major Canadian company) |
| Applicative | describes a language which does all of its processing by means of operators applied to values |
| Asynchronous | independent in time, unsynchronized |
| Automatic ports | unnamed input or output ports used to respectively delay a process, or indicate termination of a process, without code needing to be added to the processes involved |
| Brackets | IPs of a special type used to demarcate groupings of IPs within IP streams |
| C#FBP | C# implementation of FBP concepts. For more information, see C#FBP . |
| Capacity | the maximum number of IPs a connection can hold at one time |
| Component | Reusable piece of code or reusable subnet |
| Composite Component | Component comprising more than one process (same as subnet) |
| Connection | Path between two processes, over which a data stream passes; connections have finite capacities (the maximum number of IPs they can hold at one time) |
| Connection Points | The point where a connection makes contact with a component |
| Control IP | an IP whose life-time corresponds exactly to the lifetime of a substream, |

| | |
|---|---|
| | which it can be said to "represent" |
| Coroutine | an earlier name for an FBP process |
| Descriptor | read-only module which can be attached to an IP describing it to generalized components |
| DFDM | Data Flow Development Manager, dialect of FBP - went on sale in Japan - sold several licences |
| DrawFBP | FBP diagramming tool, written in Java. For more information, see DrawFBP . |
| Elementary Component | Component which is not a composite component |
| FBP | Flow-Based Programming |
| FPE | Flow Programming Environment - product that was to follow DFDM. It was developed quite far theoretically, but never reached the marketplace |
| Granularity | "Grain" size - see Chap. XXII - Performance Considerations |
| Higher-Level Language (HLL) | a language intermediate in level between Lower-Level Languages (e.g. Assembler) and 4th Generation Languages (4GLs) |
| Information Packet (IP) | an independent, structured piece of information with a well-defined lifetime (from creation to destruction) |
| Initial Information Packet (IIP) | data specified in the network definition, usually used as a parameter for a reusable component; it is converted into a "real" IP by means of a "receive" service call; it is only supported by THREADS & JavaFBP |
| JavaFBP | Java implementation of FBP concepts. For more information, see JavaFBP . |
| JFBP | Old name for Java implementation of FBP concepts - see JavaFBP. |
| Looper | a component which does not exit after each IP has been handled, but "loops" back to get another one |
| Non-looper | a component which exits after each IP has been handled, rather than "looping" back to get another one |
| OOP | Object-Oriented Programming |
| Port | The point where a connection makes contact with a process |
| Root IP | The root of a tree of IPs |

| | |
|---|---|
| Process | Asynchronously executing piece of logic - in FBP, same as "thread" |
| Stream | Sequence of IPs passing across a given connection |
| Substream-sensitivity | a characteristic of some ports of a composite component where brackets are treated as end of data |
| Thread | Same as "process" in FBP - often referred to as "lightweight" process |
| THREADS | C-based FBP implementation. The API is described in "THREADS API and Network Specification"). For the code, just click on THREADS (you may have to shift and click). |
| Tree | Complex structure of linked IPs, able to be sent and received as a single unit |
| Synchronous | Coordinated in time (at the same time) |
| WYSIWYG | "What You See Is What You Get" (describes a tool where the image shown to the developer closely matches the final result in appearance) |

# Bibliography

## Flow-Based Programming

## Bibliography


| | | |
|---|---|---|
| W.B. Ackerman | 1979 | "Data Flow Languages", Proceedings National Computer Conference, pp. 1087-1095 |
| G. Agha | 1990 | "Concurrent Object-Oriented Programming", Communications of the ACM, Vol. 33, No. 9, Sept. 1990 |
| A.V. Aho and J.D. Ullman | 1972 | "The Theory of Parsing, Translation and Compiling", Englewood Cliffs, NJ: Prentice-Hall |
| R.G. Babb II | 1984 | "Parallel Processing with Large-Grain Data Flow Techniques", Computer, July 1984 |
| J. Backus | 1978 | "Can Programming be Liberated from the von Neumann Style? A Functional Style and its Algebra of Programs", Communications of the ACM, Aug. 1978, Vol. 21, No. 8 |
| R. M. Balzer | 1971 | "PORTS - A Method for Dynamic Interprogram Communication and Job Control", The RAND Corporation, Spring Joint Computer Conference, 1971 |
| J.A. Barker | 1992 | "Future Edge: Discovering the New Paradigms of Success", William Morrow and Company, Inc., New York |
| G. Bell | 1992 | "Ultracomputers: A Teraflop before its Time", Communications of the ACM, Aug. 1992, Vol. 35, No. 8 |

| | | |
|---|---|---|
| J. Bentley | 1988 | "More Programming Pearls - Confessions of a Coder", AT&T Bell Laboratories |
| A. Black, N. Hutchison, E. Jul and H. Levy | 1986 | "Object Structure in the Emerald System", OOPSLA '86 Proceedings, Sept. 1986 |
| J. Boukens and F. Deckers | 1974 | "CHIEF, An Extensible Programming System", Machine Oriented Higher Level Languages (W.L. van der Poel and L.A. Maarsen, eds.), North Holland Publishing Company, Amsterdam |
| Ed. B.V. Bowden | 1963 | "Faster than Thought, A Symposium on Digital Computing Machines", Sir Isaac Pitman and Sons, Ltd., London, England, 1st Edition 1953 |
| F.P. Brooks | 1975 | "The Mythical Man-Month: Essays on Software Engineering", Reading, MA: Addison-Wesley |
| J. Brunner | 1975 | "The Shockwave Rider", Ballantine Books, New York |
| W.H. Burge | 1975 | "Recursive Programming Techniques", Addison-Wesley, Reading, MA |
| D. Cann | 1992 | "Retire FORTRAN: A Debate Rekindled", Communications of the ACM, Vol. 35, No. 8, Aug. 1992 |
| N. Carriero and D. Gelernter | 1989 | "Linda in Context", Communications of the ACM, Vol. 32, No. 4, April 1989 |
| I.A. Clark | 1976 | "STREMA: A Graphic Language for Relational Applications", IBM UK, Technical Report, UKSC 0084, October 1976 |
| M.E. Conway | 1963 | "Design of a separable transition-diagram compiler", Communications of the ACM, Vol. 6, No. 7, July 1963 |
| B.J. Cox | 1987 | "Object Oriented Programming - An Evolutionary Approach", Addison-Wesley Publishing Company |
| E.W. Dijkstra | 1972 | "The humble programmer", Communications of the ACM, Vol. 15, No. 10, Oct. 1972 |
| 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 |
| N.P. Edwards | 1977 | "On the Architectural Requirements of an Engineered System", IBM Research Report, RC 6688 (#28797), T.J. Watson Research Center, Yorktown Heights, NY, 8/18/77 |

| | | |
|---|---|---|
| P.R. Ewing | 1988 | "Bibliyna Simfoniya, 988-1988, Yuvileyne Vidannya", Prisvyachene Tisyacholittyu Khristiyanstva, GLINT Canada, Toronto |
| D.P. Friedman and D.S. Wise | 1976 | "CONS should not evaluate its arguments", Automata, Languages and Programming, Edinburgh University Press, Edinburgh |
| J. Gall | 1978 | "Systemantics: How systems work and especially how they fail", Pocket Books, Simon & Schuster |
| G.R. Gao | 1991 | "A Code Mapping Scheme for Dataflow Software Pipelining", Kluwer Academic Publishers, Boston/Dordrecht/London |
| P.T. Gaughan and S. Yalamanchili | 1993 | "Adaptive Routing Protocols for Hypercube Interconnection Networks", Computer, 0018-9162/93/0500-0012, IEEE |
| D. Gelernter and N. Carriero | 1992 | "Coordination Languages and their Significance", Communications of the ACM, Vol. 35, No. 2, February 1992 |
| M. Hammer, W.G. Howe, V.J. Kruskal and I. Wladawsky | 1977 | "A very high level programming language for data processing applications", Communications of the ACM, Vol. 20, No. 11, November 1977 |
| J. Hendler | 1986 | "Enhancement for Multiple-Inheritance", SIGPLAN Notices V21, #10, October 1986 |
| C.A.R. Hoare | 1978 | "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, August 1978 |
| IBM | | "Messaging and Queueing Technical Reference", SC33-0850, IBM Corp. |
| IBM | | "VM/System Product CMS Pipelines", Program No. 5785-RAC |
| IBM Japan | 1989 | "Data Flow Programming Manager (DFDM)", Product Number 5799-DJB, Form No. N: GH18-0399-0 |
| K. Jackson and Gp. Capt. H.R. Simpson | 1975 | "MASCOT - A Modular Approach to Software Construction, Operation and Test", RRE Technical Note, No. 778, Royal Radar Establishment, Ministry of Defence, Malvern, Worcs., UK, 1975 |
| M. Jackson | 1975 | "Principles of Program Design", Academic Press, London, New York, San Francisco |
| T. Capers Jones | 1992 | "CASE's Missing Elements", IEEE Spectrum, June 1992 |
| K.M. Kahn and M.S. | 1988 | "Language Design and Open Systems", The Ecology of |

| | | |
|---|---|---|
| Miller | | Computation, B.A. Huberman (ed.), Elsevier Science Publishers B.V. (North-Holland) |
| K.M. Kahn | 1989 | "Objects - A Fresh Look", Proceedings of the Third European Conference on Object Oriented Programming, Cambridge University Press, July 1989 |
| K.M. Kahn and V.A. Saraswat | 1990 | "Complete Visualizations of Concurrent Programs and their Executions", TH0330-1/90/0000/0007, 1990 IEEE |
| R.P. Kar | 1989 | "Data-Flow Multitasking", Dr. Dobb's Journal, Nov. 1989 |
| R.C. Kendall | 1977 | "Management Perspectives on Programs, Programming and Productivity", IBM Report 1977 |
| R. Kendall | 1988 | "Manufactured Programming", Computerworld Extra, June 20, 1988 |
| W. Kim and F.H. Lochovsky | 1989 | "Object-Oriented Concepts, Databases, and Applications", ACM Press, Addison-Wesley |
| L. Krishtalka | 1989 | "Dinosaur Plots and other Intrigues in Natural History", Avon Books, New York |
| T.S. Kuhn | 1970 | "The Structure of Scientific Revolutions", Chicago, University of Chicago Press |
| K. Kuse, M. Sassa, I. Nakata | 1986 | "Modelling and Analysis of Concurrent Processes Connected by Streams", Journal of Information Processing, Vol. 9, No. 3 |
| W. Lalonde, J. Pugh | 1991 | "Subclassing ≠ subtyping ≠ Is-a", Journal of Object-Oriented Programming, January 1991 |
| B.M. Leavenworth | 1977 | "Non-Procedural Data Processing", The Computer Journal Vol. 20, No. 1, 6-9, February 1977 |
| B. Liskov, M. Herlihy, L. Gilbert | 1986 | "Limitations of Synchronous Communication with Static Process Structure in Languages for Distributed Computing", Communications of the ACM, 1986, pp. 150-159 |
| J.L. Martin | 1993 | "Travels with Gopher", Computer, May 1993, pp. 84-87 |
| R. Milner | 1993 | "Elements of Interaction", Turing Award Lecture, reprinted in Communications of the ACM, Vol. 36, No. 1, Jan. 1993 |
| J. Moad | 1993 | "How to Break the Distributed Logjam", Datamation, May 15, 1993 |

| | | |
|---|---|---|
| E. Morenoff and J.B. McLean | 1967 | "Inter-program Communications, Program String Structures and Buffer Files", Rome Air Force Base, New York, Spring Joint Computer Conference, 1967 |
| J.P. Morrison | 1971 | "Data Responsive Modular, Interleaved Task Programming System", IBM Technical Disclosure Bulletin, Vol. 13, No. 8, 2425-2426, January 1971 |
| J.P. Morrison | 1978 | "Data Stream Linkage Mechanism", IBM Systems Journal Vol. 17, No. 4, 1978 |
| W. Needham | 1965 | "A Book of Country Things", Recorded by Barrows Mussey, The Steven Greene Press, Lexington, MA |
| O. Nierstrasz, S. Gibbs, D. Tsichritzis | 1992 | "Component-Oriented Software Development", Communications of the ACM, Vol. 35, No. 9, Sept. 1992. |
| D. Olson | 1993 | "Exploiting Chaos: Cashing in on the Realities of Software Development", van Nostrand Reinhold, New York |
| R. Orfali and D. Harkey | 1991 | "Client-Server Programming with OS/2 Extended Edition", van Nostrand Reinhold, New York |
| D.L. Parnas | 1972 | "On the criteria to be used in decomposing systems into modules", Communications of the ACM, Vol. 5, No. 12, Dec. 1972, pp. 1053-8 |
| R.F. Rashid | 1988 | "From RIG to Accent to Mach: The Evolution of a Network Operating System", The Ecology of Computation, B.A. Huberman (ed.), Elsevier Science Publishers B.V. (North-Holland) |
| E. Rietman and M.F. Flynn | 1993 | "FAT-Eating Logic Bombs and the Vampire Worm", Analog Science Fiction and Fact, Feb. 1993 |
| N. Shu | 1985 | "FORMAL, A forms oriented visual directed application development system", Computer 18, No. 8, 38-49, Aug. 1985 |
| W.P. Stevens | 1981 | "Using Structured Design: How to make Programs Simple, Changeable, Flexible and Reusable", John Wiley and Sons |
| 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 |

| | | |
|---|---|---|
| W.P. Stevens | 1991 | "Software Design: Concepts and Methods", Prentice Hall International |
| 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 |
| R.E. Strom, D.F. Bacon, A.P. Goldberg, A. Lowry, D.M. Yellin, S.A. Yemini | 1991 | "Hermes: A Language for Distributed Computing", Prentice Hall |
| Y. Suzuki, S. Miyamoto, H. Matsumaru | 1985 | "Data Flow Structure for Maintainable Software in Railway Electric Substation Control Systems", CH2207-9/85/0000-0219, IEEE |
| T. Swan | 1991 | "Learning C++", SAMS, Prentice Hall |
| E.D. Tribble, M.S. Miller, K. Kahn, D. Bobrow and C. Abbott | 1987 | "Channels: A Generalization of Streams", Concurrent Prolog Vol. 1, MIT Press |
| D. Tsichritzis, E. Fiume, S. Gibbs and O. Nierstrasz | 1987 | "KNOs: KNowledge Acquisition, Dissemination and Manipulation Objects", ACM Transactions on Office Information Systems, Vol. 5, No. 4, pp. 96-112 |
| L.G. Valiant | 1990 | "A Bridging Model for Parallel Computation", Communications of the ACM, Aug. 1990, Vol. 33, No. 8 |
| J-D. Warnier | 1974 | "Logical Construction of Programs", 3rd edition, van Nostrand Reinhold, NY |
| G.M. Weinberg | 1975 | "An Introduction to General Systems Thinking", John Wiley and Sons, Inc., New York |
| B.L. Whorf | 1956 | "Language, Thought and Reality", Technology Press, MIT |
| J. Winkler | 1992 | "Objectivism: 'Class' Considered Harmful", Letter in Technical Correspondence, Communications of the ACM, Vol. 35, No. 8, Aug. 1992 |
| R.J. Wirfs-Brock and R.E. Johnson | 1990 | "Surveying Current Research in Object-Oriented Design", Communications of the ACM, Sept. 1990, Vol. 33, No. 9 |
| 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 |

# Bibliography