Fiendish Designs

A Software Engineering Odyssey

Preface

These are notes, incomplete but extensive, for a book which I hope will give a personal view of the first forty years or so of Software Engineering. Whether the book will ever see the light of day, I am not sure. These notes have come, I realise, to be a memoir of my working life in SE. I want to capture not only the evolution of the technical discipline which is software engineering, but also the climate of social practice in the industry, which has changed hugely over time. To what extent, if at all, others will find this interesting, I have very little idea.

I mention other, real people by name here and there. If anyone prefers me not to refer to them, or wishes to offer corrections on any item, they can email me (see Contact on Home Page).

Introduction

Everybody today encounters computers. There are computers inside petrol pumps, in cash tills, behind the dashboard instruments in modern cars, and in libraries, doctors' surgeries and beside the dentist's chair. A large proportion of people have personal computers in their homes and may use them at work, without having to be specialists in computing. Most people have at least some idea that computers contain software, lists of instructions which drive the computer and enable it to perform different tasks.

The term "software engineering" wasn't coined until 1968, at a NATO-funded conference, but the activity that it stands for had been carried out for at least ten years before that. To engineer something, a road, a car engine, a skyscraper or a piece of computer software is to design and build it, using the techniques and technology known at the time. I have worked in software engineering for about forty years, from 1962 to 2002. For the last 20 years of this time I specialised in formal methods of software development. My first job after graduating was with Elliott's, a British computer manufacturer, in 1962. Elliott's no longer exists as a separate company, but has been absorbed into the last remaining British mainframe manufacturer, ICL, through a succession of mergers and take-overs.

Chapter 1 Flanges and Festivities

Late in the fifties, British Thompson-Houston merged with Hollerith to become the British Tabulating Machine Company. Both these companies produced tabulators, machines that sorted punched cards and which, with a bit of persistence and patience on the part of their operators, could carry out basic statistical processes. In 1960 Powers Samas, an engineering firm, merged with them and the result was International Computers and Tabulators, ICT. I had worked as a vacation student with ICT just at that time, doing logic design and circuit board layout for a new compact military machine; if I ever knew the name of it, I have forgotten. But ICT at about that time were conscious that the word "tabulator" indicated an obsolescent technology, and so they changed their name to International Computers Limited,

ICL. By the time I had graduated, there were two other large British computer manufacturers, English Electric and Elliott's. Elliott Brothers had themselves recently absorbed part of NCR, National Cash Registers Ltd. There was also a smaller firm that nonetheless produced mainframe machines, Leo Computers. Leo Computers came about in a rather surprising way. A chain of cafés called Lyons' Corner Houses were to be found all over London. They needed to coordinate their accounts and required computer power to do so efficiently. Rather than pay someone else to supply their computers, they formed their own computer manufacturing subsidiary, Leo. No "buy in" policy for them! Leo continued producing computers for some years before they were bought by English Electric, who also took over Marconi, an electronics firm who, famously, designed and manufactured radios but also much other electronic equipment, especially for the military. Marconi was one of the firms with whom I had an interview when choosing my first job after graduating. English Electric made the KDF9 computer, a rival in size and performance to the IBM 360, which dominated the market for many years. Later still, towards the end of the sixties, English Electric-Leo-Marconi, EELM, was absorbed into ICL.

But back in 1962 Elliott's produced the 803 machine and were embarking on a more powerful version of it, with the same instruction code, the Elliott 503. In the early 1960s computers were seen as large calculators: machines that could carry out complex mathematical calculations. Most, if not all, application software was about performing large calculations for some purpose. Academic courses reflected this, and were limited to post-graduate diplomas, concentrating on numerical analysis and automata theory. Several polytechnics also provided some more practical HNC courses. Most programmers working for Elliott's were maths graduates, a few with a post-graduate diploma in computing. A minority were HNC holders.

I was one of a batch of new graduates who joined the Scientific Computing Division at Elliott's. Other graduates joined several other divisions at the same time, so there were some twenty or thirty of us, all newcomers. We rapidly got to know each other and, almost all of us being single graduates living in digs and assorted shared flats and so on, there would be a party to go to every Saturday night. All of us in the Scientific Computing Division spent a few days on a course in which we were taught to program. The language we were taught was the machine code for the Elliott 803. This machine occupied a fairly large room. The central processor was housed in several six-foot high 19 inch wide cabinets. The 803 was one of the first machines to use semiconductors, germanium transistors and diodes, for its electronics. It was consequently relatively compact. A paper tape reader and punch, and a card reader and punch were held in similar cabinets, and the operator's console was desk-shaped, and held an on-off switch, various lamps and buttons and a "number generator", which was a row of toggle switches on which one could set up a binary number. There were instructions in the machine for reading the setting on the number generator and for displaying patterns on the

lamps. One could also set up an instruction on the number generator and press a button causing the machine to obey it. This was the principal way of booting up a program, by setting up a jump to its starting address on the number generator and obeying it. A line printer could print out results of the calculations or other work that a program had performed. These printers were the first to be attached to a computer and consisted of a rotating drum and rows of hammers. Paper edged with detachable sprocket holes would be fed into the printer. The drum had lines of characters embossed on it. Each line had the same character embossed along all of its character positions. Usually, a line of characters would be printed on each rotation of the drum. The character to be printed at each character position was controlled by timing the hammer at that position to strike when the correct line of the drum was under it. These printers were large and noisy, but were the main means of printing output for most computers. Programmers could book time on the machine to run their programs, or request one of the operators to do so. I preferred to make use of the operators' services, but most of my colleagues liked to be more hands on. I caused much amusement when I went to use the machine myself for the first time after I had been working there for nearly a year. "What, you've never used the machine before, Tim?"

The first programming course covered the two main versions of the machine code and gave advice about how to program, using the accumulator and main storage. The machine code was numeric, having a couple of octal digits for the instruction and the rest for the address. Octal numbers were used rather than hexadecimal. The word size in the machine was 39 bits. This may seem bizarre today, when word sizes are invariably a whole number of bytes. Each instruction occupied 18 bits, so that two instructions fitted into each word. One bit remained, in between the two instructions. This was called the "B-line". Setting the B-line caused the contents of the address in the first half to be added to the second instruction. This was useful for working with arrays and lists. The first five bits of each instruction contained the operation code and the remaining 13 bits the address. In this way the instruction code could address a store of 8192 words. The computers were supplied with a main store of either 4096 or 8192 words. The first versions of the 803 had no backing store. Any intermediate information generated by a program would be punched out onto paper tape, ready to be read in again. Later versions had backing stores consisting of magnetic film, specially manufactured by Kodak. This was 35 mm film with sprocket holes and a magnetic coating instead of a photographic one. The film was wound onto reel-to-reel decks.

The simplest version of the machine code was " absolute"; the addresses in the instructions referred to specific, absolute locations and the program would only work if it was loaded into a specific location in store. This was so primitive and restrictive that it was only used for some very basic utilities. The relocatable version of the machine code would be loaded into the store with an offset address added to all the relevant address parts of the instructions by the loader program, so that the program would work wherever it was located in the main

store. There was also an assembly language, in which address locations were more like the identifiers in higher level programming languages and the operation codes were three-letter mnemonics, like LOA for Load and STO for Store.

The programming course put great emphasis on subroutines. This is a term that has almost fallen out of use; the equivalent construct in high level languages is the procedure or function. However, strangely there is one exception: in the various instantiations of the Star Trek episodes, Commander Data and the holographic Doctor, both containing as sophisticated software as you could imagine, have "subroutines" for ethics and other features. Subroutines in 1962 could give structure to a program, separate concerns, and save a great deal of storage space. Storage space was at a great premium in those days. The main store consisted of very small magnetic rings with activating and readout wires wrapped round them. The typical cost was about GB £1 (US $1.8 or €1.5) per word. So a great deal of effort was put into keeping one's programs compact. In my first project, I started by taking this advice a bit too literally. Having written my program, I scanned it for instances of repetitions of three or more instructions and turned them into a subroutine. Of course these little subroutines had no intrinsic meaning or purpose, and the overall structure of the program became quite amorphous. I could not get it to work and was advised to start again from the beginning. So my first lesson in software engineering was to use structuring tools like subroutines to reflect aspects of the problem rather more than to try to save storage space.

I was given a project to do shortly after the programming course. The program was typical of those written to perform calculations. It also indicated part of the different economic balance between supplier and customer. Many customers who bought computers did not program them themselves. They would often require them for just a small range of calculations. So Elliott's quite often used to write a few programs for a customer in order to secure a sale. Tube Investments used an empirical formula to calculate the size of flanges for bolting two tubes together, and the number and diameters of holes in the flange, given the diameter of the tube (figure 1).

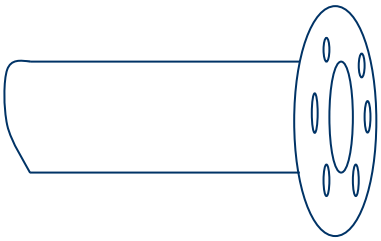

Figure 1

I was asked to write a program to perform this calculation. I threw away my first version with its too many low level subroutines, and was able to get a second version working without trouble.

Working at Elliott's was interesting, and quite fun, if exasperating at times. The location was Borehamwood, a small town alongside Elstree, on the north edge of London. The workplace included a small manufacturing plant, with the offices and computer rooms where the programmers and design engineers worked, at the front of the building. The main manufacturing plant was in Cowdenbeath in Scotland, and few of us ever needed to go there. Elliott's was emerging from an industrially class-ridden tradition into a more enlightened era, but still had some way to go by today's standard. In the year I joined them Elliott's paid men and women graduates the same rate for equal qualifications for the first time; in previous years a woman graduate with exactly the same degree class was paid slightly less than a man. In the programming areas there were several women managers, a fairly rare phenomenon in those days. Most firms had separate "executive canteens" for senior staff, but an increasing number were following American lines of a single, "democratic" facility. Some firms even banned employees from discussing or revealing their salaries to each other; it could be a sackable offence. Although Elliott's did not like their staff doing so, they did not penalise people for it. Because computing was a relatively new occupation, the demand for new graduates in the appropriate disciplines, mostly maths and engineering, was high. So new graduates were enticed with higher salaries, and existing employees of a year or two's experience discovered to their annoyance that raw recruits were engaged at a higher salary than they were earning themselves. In one case a section leader even found that new graduates in her group were earning more than she was. Elliott's was persuaded to make some adjustments.

There was an ambience of scientific enterprise and excitement about Elliott's Scientific Computing Division. One of the first ever Algol60 compilers was being designed and built. Two new machines were designed in succession, the 503, which was a more powerful version of the 803, and later the 4100 series of machines.

There were two professional institutions to which programmers could belong: the British Computer Society and the Association of Computing Machinery, which was centred in the USA although people from any nationality could belong to it, as indeed they can to the BCS. Elliott's encouraged us to join the BCS and there was an excellent subscription rate of just £1 per year for anyone who had graduated in the last three years. I joined. One got a good deal for one's £1, four copies of the quarterly BCS Journal and copies of the more frequent and less formal BCS Bulletin. Three years later I ceased to be eligible for the ultra-low student rate and, at the same time, the BCS decided to seek a Royal Charter, have its own coat of arms, and various other things. I did not like the way the society was going, trying, it seemed to me, to turn itself into an august and remote organisation and not as centred on academic excellence as it had been. In addition, they were putting their rates up, so my own subscription was going to increase from £1 to £9 a year, equivalent to half a week's salary. So, although I enjoyed membership of the BCS for the first three years after I graduated, I

left. I was to join them again many years later in the 1980s, when membership of the BCS became a qualification and normally entailed passing an exam. The BCS Journal and the Journal of the ACM were circulated round the programmers in the Scientific Computing Division. We used to read most of the papers in those journals eagerly. This would be an impossible feat nowadays. There is a plethora of journals, many of them highly specialised. Twenty years later when I was a Chief Research Engineer at Standard Telecommunications Laboratories in Harlow, I realised that my division was receiving 41 different journals and most people did not have time to read any of them.

Software lives inside a many-faceted context. Who writes it? Who uses it? What technology underlies it? How is it executed? What influences its development? In 1962 not many people wrote software, even those rare organisations that bought themselves a computer. Hardware was very expensive, hence computer time was expensive, and labour cheap by comparison. A firm would usually buy a computer just to carry out a small handful of different calculations, repeated many times with different data. Computer manufacturers like Elliott's would often write this "application software" (the phrase had not been coined then) for a prospective customer, just to obtain the sale of a computer, such were the relative economics of hardware and software. The very large machines were owned by few organisations, such as computer bureaux. Time on the bureau's machine would be hired to customers. So the bureau would usually double up as a software house, writing the application program and selling the result and the computer time spent in running it to the customer. Often the bureau would retain the program, since it could only be run on a possibly unique machine. The bureau indeed might keep the IPR of the program. My first program for flange design was such an application program for a customer. The programmers at Elliott's found themselves writing a mixture of customer's application programs and systems programs that supported the use of the machine: compilers, assemblers, device handlers, and parts of the minimal operating systems. Periodically we produced new and updated examples of these and distributed them to customers who had purchased machines. One enthusiast, a Dr. Murray at Edinburgh University, would examine each of these new products and produce one of his own, faster, more compact and sometimes with extra useful facilities. He would then send it to Elliott's Scientific Computing Division as a gift. We would check it out, find that indeed it was a superior product, thank him graciously and adopt it as the next released version. This customer became quite a legend within the division. We even joked that we need not try too hard to make a fast, efficient software product; Dr. Murray would respond with a high quality version in short order! But we did not actually adopt such a policy. As far as I know Dr. Murray never asked for any reward. I wonder where he is now.

My second project was not a programming project at all. Elliott's were producing the 503, a development of the 803 machine, but faster, with superior electronics employing more up to date semiconductor devices, and with a larger range of peripherals. Instead of the rather crude

row of toggle switches, the "number generator", the operator controlled the computer by means of an on-line IBM electric typewriter, the most up to date "golf-ball" typewriter. The golf-ball had embossed on its surface the whole character set in the chosen typeface. These golf-balls could be exchanged, so it was possible to type documents containing Greek letters and mathematical symbols using such a typewriter. They were expensive, costing about £650, about £13,000 or €20,000 in today's money.

Other new peripherals were candidates for attachment to the 503, including some rather odd suggestions, which fortunately were not adopted. One was a card reader-punch. Although paper tape, in two widths, 5 holes and 8 holes, was becoming the dominant bulk input medium, punched cards were still used to a considerable extent. These cards could be punched with holes in fixed positions along twelve rows and eighty columns. The proposed card reader-punch could both read and punch cards, so it would be possible for the computer to read a card and punch extra holes in it. We were mostly horrified at the idea and could not imagine how it could be used in any systematic way. Another proposed peripheral device was a magnetic card reader. Again, although this seemed more practical, most of the staff thought it would not be accepted and it too was dropped. It is intriguing to think that today credit, debit and other cards with magnetic stripes have for a long time been a universal part of our lives.

One peripheral which was adopted was a flat-bed plotter. This was a graph plotting device rather like those one sees in weather stations for plotting the temperature and pressure. The computer could drive the pen back and forth and the paper under it, using subroutines for drawing lines and curves. This was rather fun to write software for, and we experimented with various routines for drawing and scribing titles and even continuous script. One more advanced model even had several colours of pens that could be called into action.

Another peripheral device that was adopted was the magnetic tape deck. A development from the magnetic film complete with sprocket holes that the 803 used, the magnetic tape was much like a bulky version of the tape that is used today in video and audio cassettes, but mounted on reel-to-reel, vertical decks. A more experienced colleague, Vivian Kelly, and I were given the task of specifying the magnetic tape system. It would normally have been the role of hardware engineers to specify this system, that is, to define exactly what the system should do in detail, how it should respond to instructions from the central processor within the computer and from the operator. Elliott's had the novel idea that instead of the hardware engineers writing this specification, the programmers should do so. After all, it was the programmers who would be writing the software to drive the magnetic tape system, and therefore who were its "users". This task of specification is separate from design; specification determines what the system should do, whereas design determines how it should do it. The argument was, therefore, that the "users" of the magnetic tape system, i.e. the programmers, were in the best place to specify it. The programmers were the users because

they wrote the software that drove, i.e. used, the system. I have always thought that, although logical, this was a bold move on the part of the management. The programmers, not being electronic or mechanical engineers, had no idea what would be feasible. Nonetheless, we wrote the specification, consulting the hardware engineers at frequent intervals. This was a tedious writing task, since we were using carefully framed English; our phraseology became almost legalistic, for it was very important that what we wrote should not be ambiguous or misunderstood. I remember painstakingly writing sentences such as "when the Load Button is depressed, the Load Lamp is extinguished", and page after page of such statements. When we had finished the hardware engineers designed the electronic and mechanical system to perform the functions we had defined, and I moved on once more to some real software design, the device handler for the magnetic tape system we had just finished specifying.

This distinction between the specification of what a software or hardware system was intended to do, and the design of how it was going to achieve it, was beginning to be seen as more and more important. The idea of separating the stages of thinking required to produce a design eventually led to the notion of "separation of concerns" coined by one of the greatest computer scientists, Edsger W. Dijkstra, in 1974. I remember one of our colleagues, Bill Williams, wrote a paper for the British Computer Journal about the project he was working on, a simulator program called the "Elliott Simulator Package", or ESP. The editor of the journal queried this choice of name for the program, pointing out that ESP usually stood for Extra-Sensory Perception. Bill's riposte to the editor was that the paper described what the program did, and not how it did it! The editor relented. Elliott's approach of the user specifying the functions of a piece of software foreshadowed the user-led approach that became the vogue in the nineties, especially within Framework 5 of the European Commission's research and development programme in information technology.

Vivian Kelly and I had desks next to each other while we worked on specifying the magnetic tape system. We got on well together and had many conversations. I admired the way he spoke with warmth and affection to his wife, a delighted smile on his face, when she occasionally telephoned him at the office. Most of the men I worked with, including myself, felt embarrassed and inhibited when telephoning our loved ones within earshot of our colleagues. One day after work I found myself walking to the railway station alongside him. Vivian usually drove to and from work, so I asked him where his car was. He replied that his wife had taken it on holiday with her. Rather nervously, I asked him if he and his wife did not go on holiday together. He explained that his wife was a barrister and that the Inns of Court in London closed down for eight weeks during the summer "on account of the stench of the river Thames"[1].

[1] For the benefit of those not familiar with London, the odorous problems with the Thames were largely cured during the nineteenth century when Joseph Bazalgette and the Metropolitan Board of Works designed and built the London sewage system .

My next project, designing the magnetic tape device driver for the 503, was a natural progression from specifying the system. This was a true software project, and a systems program rather than a user application. I recall little difficulty with this task. These were, however, before the days of widespread use of interrupts in computers. A program requiring data to be read from or written to the magnetic medium would simply have to wait for the transfer to finish. But interrupt systems were being devised in the industry and theoretical work was being done to find ways of ensuring the integrity of processes that were interrupted by others. Today every personal computer user takes for granted the ability to carry on doing some word-processing, say, while some information is slowly being downloaded from the internet. The principles enabling this kind of multi-processing were being researched around the beginning of the 1960s.

For the first few months at Elliott's, because of lack of desk space, I worked in an office full of sales representatives, all men. They were, like the programmers, all graduates in engineering disciplines but had, I presume, decided to pursue a career in sales. They wore suits and ties and seemed to spend their time in the office making phone calls, projecting their personalities down the telephone and starting their persuasive tactics by enthusiastically inviting the prospective client to lunch. This was a noisy and energised environment to work in, and I found it distracting to say the least. The salesmen never ate in the canteen, but would drive to some nearby pub for lunch. They would invite me along, and our lunchtimes tended to be substantially longer than the regulation hour. One day Tony Hoare, who was in a line manager position in charge of the programmers, decided that we needed a pep-talk about extended lunchtimes, but perhaps unwisely called the meeting for two o'clock. I heard later that he started his talk with the words "most of the offenders are notable by their absence". I fear I was not a good time keeper. Security guards used to take one's name if one was late, but I soon discovered that if one arrived seriously late, forty minutes or so, they would have given up taking names down, probably assuming that one had attended to some legitimate duty in another part of the building. However, I believe I put in a fair amount of time and effort in total.

After a while I was moved out of the sales office into a room with other software staff. I have to say I was quite relieved, for although I enjoyed the company of the salesmen, I found their office was not conducive to deep analytical thought! The dress code among the programmers was more relaxed than that of the salesmen, but I probably carried this a little too far. One of my managers enquired with a polite smile "No shirt? Holes in your jersey?". Well, it was a polo neck, so any shirt would have been invisible. The Scientific Computing Division had decided to embark on designing and manufacturing a new machine. No name was given to this machine at first, but the whole enterprise was called "Project 41". It was supposed to be commercially highly confidential. We were under strict instructions not to discuss it outside Elliott's. The design of the machine instruction code was still in a state of flux. The senior

technical people thought that experiments should be done with programming the machine before setting the new instruction code in concrete, so to speak. I was therefore asked to write a simulator for the new machine, which would run on the 503. People could then write prototype systems software in the code of the new machine, test it out and measure its performance in terms of time and storage space, before the first machine cast in real hardware was produced. The development of the systems software could progress in parallel with the hardware development, so that both would be ready at the same time, shortening the time from first ideas to available product, the "time to market".

There was a lot of discussion about what would be a useful repertoire of machine code instructions for the new machine. One idea was to take the systems software already written for previous machines, the 803 and 503, and to count the usage of the different instructions. However, this study produced an unexpected result. Instead of showing which were the generally favourite and useful instructions, the count showed that there were marked differences between styles of the different programmers who had authored the programs; indeed the counting technique turned out to be quite an accurate way of determining who was the author of different pieces of software! So that approach was abandoned.

After a while the repertoire of the instruction set was mostly agreed, but then there seemed to be interminable discussions about the design of the written form of the assembly language itself. The instruction set was simply an association between binary patterns and functions that the machine's central processor would carry out. The way the programmer would write down these instructions was yet another thing to be decided. We seemed to be getting nowhere, and I thought that if we had something concrete to criticise, at least we might make progress. So I spent just on a week writing a proposal for the assembly language. The office environment was very different from today's. There were no photocopiers. If you wanted to make copies, you needed to know in advance, and have a typescript prepared on a special foil which was then used as a master for a Gestetner copying machine. Then copies could be rolled off. These copies I remember were on flimsy paper, shiny on one side, with faint print and a bit unpleasant to handle. A typewriter could be used to make up to about four carbon copies, but these had to be made at the time that the document was being typed. So I simply wrote my document by hand and passed it round the three or so others who were involved in the project. They were surprised at the progress I had made and this assembly language, with a few revisions, was adopted. For the Elliott 803 and 503 machines the company had produced a programmer's "Facts Card", a small fold-over card with all the essential instructions succinctly listed. It was small enough to fit in one's pocket, and we found these very useful. Much of my description for the new machine code eventually found its way onto the new Facts Card, something I felt rather pleased about at the time.

The simulator for the project 41 machine did not take me too long to write, although it was a large program – some 900 instructions I recall. The reason was that it was extremely simple

in structure. A central controlling piece of code would extract the instruction to be obeyed and switch on its operation code to a routine which handled that particular function. I used subroutines to deal with the actions that were common to many instructions, like extracting the contents of the address, modifying the address with an index, etc. I seem to remember the whole exercise took me about three weeks, although I was continually adjusting the program as new suggestions and ideas for the machine came forward. At the same time one of the hardware engineers, Fred Harkin, who had been at Elliott's for the same length of time as I had, was building the prototype machine. However, to make even a small modification to my simulator I had to change the code, edit the tape, re-assemble the program by booking time on the 503 and test it out with suitable test data, which I also had to prepare. The turn-round for this process would take a day. Fred just had to go to his prototype, make some physical changes and as often as not his modifications would be up and running within an hour. So, in the end, my simulator was not used very much; programmers would use Fred's prototype instead, for which there was not a lot of demand, whereas the department's 503 was used for many different purposes and one had to book time on it in advance.

A few years later, in the late 1960s, the topic of software metrics would become all the rage. More of this in due course, but it was a theory that I never found at all convincing, largely because of this early experience. My simulator comprised 900 instructions and took three weeks to complete. My next project was to result in a thirty-instruction program and took nine months to complete, for very good reasons. The general objective of software metrics was to predict the effort that developing a piece of software would take by estimating its physical characteristics, number of instructions or lines of code, or something a little more complicated. These two early projects of mine belied this possibility.

A name had to found for the new computer being developed in project 41. Great commercial secrecy had surrounded the project up to that point, because the company did not want its competitors to know anything about the new machine or indeed that we were developing one at all. There were lots of discussions about this: whether the machine should have a name rather than a number, even. Ferranti had named all their machines after classical Greek deities, Mercury, Orion, Pegasus, Atlas, Titan. So we felt that we could not go along that path. Nor could we use the names of planets or constellations, because they coincided with the names of deities. Other ranges of numbers had been taken by other manufacturers, for example 360 by IBM, KDF9 by English Electric. In the end we just called the new machine after the project, and added a couple of zeros on the end, so that it became the 4100 series. No other company was using four digits in their machine names at that point. So the 4100 machine became so named rather accidentally.

There was, throughout Elliott's, an atmosphere of exciting innovation. The programming language Algol60 had been designed just two years earlier by an international panel of distinguished experts. To be able to program in a language like Algol60, a "compiler" has to

be written. This is a program that translates programs written in the language into the machine code of the machine on which that program is to run. Compilers were considered to be some of the most complex programs anyone could be called upon to design, requiring a great deal of analytical thought. The more sophisticated the language, the more complex its compiler. Algol60 was the most advanced high level language designed up to that point. Its predecessors were Fortran and Cobol. Fortran was a language principally for translating formulas and Cobol, essentially the same in its technical sophistication, was designed for business calculations. Algol60 was a step forward in concept, treating computational procedures and functions as principal objects and being based on a mathematical theory called "Lambda Calculus". Under this theory the meaning of a computational procedure was defined by rewriting and expanding textual formulas. But, for efficiency, indeed for practical feasibility, the computer had to translate this textual rewriting process into a more direct computational one. In the room next to the one where I worked, the first ever commercial Algol60 compiler was being written for the Elliott 803. I experienced some envy not being a member of that team but, happily, there was a culture of sharing ideas among the different projects and academic style seminars would be held quite often, in which the design ideas of different groups would be explained and discussed amongst everybody. I was most impressed to hear that, for an exhibition, a machine was dismantled into parts, crated and shipped (literally, by ship) to Moscow, reassembled, switched on, and displayed the message "Algol Ready": the Algol60 compiler was still intact in the main, random access, core store of the computer and ready to go.

However, the proximity of this innovative work on the Algol60 compiler was something of an inspiration, and in any case I was given a fairly leading edge task, that of writing the kernel of the operating system for the 4100 machine. My technical manager for doing this was Tony Hoare, who had also led the team producing the Algol60 compiler. An interrupt system was being designed for the 4100 machine. This was a hardware extra that could enable an external event like a signal from a peripheral device, a magnetic tape for example, to interrupt the program currently running and divert control to another piece of software designed to receive data from the device. So a program could continue running while data is being sent to or received from a peripheral. Users of present-day personal computers probably take this for granted. As soon as you ask for a document to be printed, for example, you can continue using an application while the printing goes ahead. Until interrupt systems were invented, a program simply had to wait until peripheral activity had finished, before resuming. The kernel of the operating system is a central piece of software that directs the computer to switch between obeying different sections of code associated with interrupts from the various peripheral devices. Interrupts were organised into priority levels. All these rules and arrangements were being designed, mostly from scratch. I was working very much under Tony Hoare's direction in this effort. He was conceiving new ideas at an amazing rate.

At times it felt as if he would have new insights every night and would explain them to me in the morning first thing each day.

Tony Hoare later moved to academia, becoming Professor of computer science at Queen's University, Belfast in 1968, and subsequently joining the Computing Laboratory at Oxford University in 1977, of which he was later Director. In 1980 he was awarded the ACM Turing award and was made a Fellow of the Royal Society in 1982, one of the first computer scientists to be made an FRS. In 2000 he was knighted for his services to computer science, again one of the first computer scientists to receive such an honour. Yet his initial background was not science, but in classics; his knowledge of computer science theory and the mathematics that is necessary for its understanding were entirely self-taught. Having such a brilliant individual as my project supervisor in 1964 was a stimulating, if at times gruelling, experience.

Eventually I completed the 4100 operating system kernel after about nine months. It was a mere 30 instructions, but it took me over thirty pages of documentation to explain its purpose and how it worked. "Documentation" is a set of documents describing a piece of software, written for the human reader. Elliott's considered documentation to be very important, but concentrated on documents aimed at subsequent programmers who might want to understand the software so that they could extend it or modify it. This was especially important if the original author left the company. Over the next few years the computer industry was to consider documentation to be extremely important, not just for other programmers, but for users, managers and others. But at that stage even user manuals were much less in evidence than today, when one may see rows of books in general bookshops explaining how to use Windows, Java and so on.

At one of the many Saturday evening parties I met another new graduate working at Elliott's, Brenda Allatt. We danced for a long time to the strains of Diana Ross singing "Babylove". We got married the following year. Brenda, who later changed her name to Hazel, was working in a division that was producing the on-board flight control software for a new military aircraft, the TSR2. The quality control procedures for military projects were ponderous and thorough, with the result that equipment approved for use was always older than the current state of the art. This is, I believe, as true today as it was then. Consequently, the on-board computer of the TSR2 was of primitive and archaic design, even by the standards of 1964. The main store of the Verdan computer was not a random access core store, but a rotating magnetic drum. By the time an instruction was obeyed, the drum would have moved on several words, so to increase the speed of executing the program, each instruction would specify the address on the drum of the next instruction to be obeyed. By taking into account the execution time of each instruction and carefully placing the sequence of instructions forming the program round the drum, the programmers could ensure that the software was not slowed down by unnecessary rotations of the drum. All the programs were

written in octal on special coding pads, which were designed to facilitate this process of mapping the program's instructions to addresses on the drum. The drum was of a limited size and, to save space, programmers would consult each other's programs to share constant data and other items. A great contrast to today's extravagant attitudes to storage space! Programming this machine in octal was a task requiring meticulous attention to low-level detail and not for everyone's aptitude. Hazel's first task was to take over a program from someone who had recently left the company. After studying the program for a long time to no avail she consulted her manager. They could make no sense of the "program" that Hazel had inherited and concluded that her predecessor had no clue about what he was doing: he had simply filled the coding pads with random octal numbers for a few months before leaving the company. Much later I discovered that this individual had been promoted to data processing manager for a local authority in the west of England, and later still he became chairman of the Computer Services Association. C'est la vie.

Elliott's was situated in Borehamwood, a small town contiguous with the village of Elstree. Elstree was famous for its film studios, where many classic British films were made: The Titchfield Thunderbolt, School for Scoundrels, Man in the White Suit, The League of Gentlemen among many others. From time to time one would see film extras in the street, which could be quite startling if they were dressed as soldiers from the German SS, for example. One would often see famous film actors in the Elstree Way pub immediately across the road from Elliott's. This pub did an excellent line in roast beef sandwiches at lunchtime, freshly and generously carved, with the option of horseradish or mustard. The lady who carved the beef did so with an expression of pained reluctance, as if to say that she would like to carve even thicker slices, but commercial considerations forbade her from doing so. I can recall the taste to this day.

However, despite the fine roast beef sandwiches in the Elstree Way pub, I decided it was time for me to move on.

Chapter 2 Mighty Atlas

In the 1960s the several different colleges within London University shared both a computer science research department and a large computer. The University of London Institute of Computer Science was a free-floating department, situated in Gordon Square in the heart of Bloomsbury and largely independent of any particular college. Bloomsbury was an elegant part of London with several green squares containing shrubs and trees, and imposing seventeenth and eighteenth century buildings, which had once been private residences of the wealthy. The computer was one of the largest in the country, the London Atlas, designed in a joint effort with Manchester University and built by Ferranti at a cost of £3.7 million. In those days a new graduate would be lucky to earn £1,000, so to compare with today's 2007 money one would need to multiply that by 20, say about £75 million or €120 million. When London

University commissioned the computer, there were many questions asked about whether funds intended for other purposes had been improperly diverted to this purchase, and the university had to claw half the money back. They did this by establishing a commercial company, which bought half the machine from them. The company owned half the time on the machine and used it on a commercial basis to recover its share of the cost over a number of years. This company was called the University of London Atlas Computing Service, or ULACS. It was situated in the same building as the Institute and the computer. Although the two bodies occupied different parts of the building, 39-43 Gordon Square, they shared several facilities in addition to the computer: a staff common room, a lunchtime snack bar, a lecture room and entrance hall.

The London Atlas computer was one of three. The first to be built was the Manchester Atlas, and in Cambridge a very similar machine, with the same instruction codes but a different operating system, was called the Titan. The London Atlas was situated on two floors. The central processor and chief operator's console was in the basement, and the peripheral devices, that is magnetic tape decks, paper tape readers and punches, punched card readers and punches and so on, were on the ground floor. A closed circuit television system and communication system linked the two, so that the chief operator could observe the peripheral activity and issue instructions to the assistant operators. They would mount and dismount tapes on the tape decks, feed the paper tape and card readers, unload the paper tape and card punches, and collect the results of the separate jobs together. The Atlas, although large and of sophisticated design, was older than the Elliott 803 and 503, and used valves (thermionic vacuum tubes) rather than semiconductors for its electronics. Even on the rare occasions when the machine was not in use, it was almost never switched off. The inevitable surges produced when the machine was switched on again would blow many valves, and the time taken to detect which had blown and the cost of replacing them would outweigh the considerable savings in not having the machine consuming electricity.

I have no clear memory of my interview with ULACS, but it must have gone smoothly because I was offered a job at a substantial increase in salary. I accepted, and started work there in 1965. The working hours were 9.30 to 6.00 p.m., the later starting time reflecting the difficulty of commuting into central London. On arriving on my first morning promptly at 9.30 I found that no-one in the department I was to work with had yet come in to work. One of the managers, George Davis, was found and he bustled down into the reception hall and introduced himself. He supplied me with some literature to read until someone arrived.

ULACS was both a computer bureau and a software house; it sold time on its computer and it wrote software for clients, software that was almost always designed to run on the Atlas machine. There were two teams of programmers, the systems and the applications programmers. I joined as a senior systems programmer. The systems group maintained and improved the systems software for the Atlas, language compilers and parts of the operating

system, which formed the technical platform on which users' programs could run. We were thus working on improving the facility that made the machine an attractive proposition on which to hire time. The applications group wrote bespoke programs under contract for customers. Occasionally members of the systems group would also write a program for a customer contract, usually when the program required some more complex technical expertise. Thus the work done by the applications group would lead more directly to earnings for the company than that done by the systems group. This was always to be a source of some contention, even though selling computer time, which was ultimately supported by the systems group, accounted for 80% of the company's revenue, whereas bespoke software contracts accounted for 20%. There was always the suggestion that the systems group were distant from the company's need to make money. The two groups were much the same size, about 20 programmers.

In addition to the two programming teams, there was a team of salesmen, and computer operators who worked in shifts, 24 hours and weekends. Because the machine was shared between ULACS and the Institute of Computer Science, those in charge of the operations were academic staff. These two or three academics had taken part in the design of the central part of the operating system and were very familiar with it. There were also various management and accountancy staff, and reception staff who were hired from an agency. These reception staff, dressed in uniforms like traffic wardens or security guards, used to cause a great deal of resentment. They were rude to visitors, knew nobody's name, not even the managing director's, and were offhand to everybody. From time to time the agency were persuaded to reassign some of their staff, but the underlying problem persisted. They could not have presented a good impression to potential customers.

I had started at Elliott's at a salary of £875 and left with £1100 per year. I started at ULACS with £1450, a substantial 32% increase. I think computer programmers may never have had it so good. There was a great demand for them and firms would offer enticing salaries to attract staff from other organisations. One could expect to double one's salary every five years or so. This led to a considerable turnover rate, with programmers typically changing their job every two or three years.

There was quite a lot to learn about the technicalities of Atlas. Although the machine was built out of more primitive hardware than the Elliott machines, the operating system was very advanced for its time. Up to four main programs could run concurrently, the central processor switching between them following the rules of a scheduling program within the operating system. By choosing which programs to run together, the operators could optimise the usage of the machine's resources. The operating system was thus called "time sharing", one of the first in the world. In addition to sharing the machine's time and storage space between up to four main programs, the operating system would time-share with peripheral transfers, just like the Elliott 4100 did. To speed up the machine, there was a "look ahead" system within

the hardware of the central processor. Instead of just obeying the current instruction, the central processor, or "mill" as they called it, would extract the next three instructions and start preparing to execute the next two in addition to the current one. Of course, if control was transferred to a different sequence of instructions, by a jump or an interrupt, most of this preparation work would be discarded, but generally instructions are obeyed in sequences of some length and the look ahead mechanism speeded up the running time of the machine considerably. This technique is still used in today's computers, with look ahead mechanisms operating on anything up to the next twelve or more instructions.

The other advanced feature of the Atlas operating system was its two-level store. The main store, directly addressable by machine code instructions, was split between a fast random access core store and a secondary magnetic tape. The operating system would switch blocks of store content between these two so that the mill would in fact operate on instructions and data in the random access store, yet programs could behave and be written as if there were a vastly larger addressable storage space. It was this two-level store concept that in turn enabled the time sharing between programs that could otherwise not have fitted into the random access core store, and indeed enabled very large programs to be written and run.

The Atlas machine was one of the very first to pioneer this two-level store concept, which is used routinely now in the design of personal computers and virtually all other machines of any size; the exception being very small embedded machines dedicated to a specific task, like computers inside manufacturing machines or car engines. Those computers are often classified as "programmable logic controllers" or PLCs, but they are simple computers whose basic design is just the same as the central processors of early computers such as the 803. The two-level stores of personal computers are split between random access stores and hard discs, but when the Atlas was designed there were no magnetic discs. So the secondary storage medium was magnetic tape. When I joined ULACS, discs, in particular exchangeable disc packs, had been invented and a couple of units were attached to Atlas, as something of an experiment. Indeed, they had been introduced as a peripheral for the Elliott 503. The exchangeable disc pack performed the same rôle as a floppy disc or rewritable CD on a personal computer, but the drive was a separate unit the size of a commercial washing machine that one might find in a launderette. You could remove the disc from the unit and exchange it for another, hence the name. However, instead of a single disc like a floppy disc or CD, these discs came in packs of eight, all mounted on a single spindle and having a diameter of about 50 centimetres. All eight discs would be read and/or written by eight arms with reading and recording heads retracting in unison. The removable disc packs were also rather unwieldy and one would see operators heaving them across the room. Considerable care was needed to remove and insert them into the unit without damaging the heads or the drives, and only trained operators were allowed to handle them. I was told that these discs rotated at great speed and, given their substantial mass, if a bearing in the unit broke and a

disc broke free it could slice through a metal cabinet or two; potentially lethal! Its descendants, the floppy disc and the rewritable CDs and DVDs are astonishingly handy by comparison.

When I joined ULACS at first I shared an office with Peter Hughes, who was the Chief Systems Programmer. My first programming project was to write another device handler for the operating system, a magnetic tape handler that could read tapes written on the Ferranti Orion computer. The information was recorded in a different arrangement on the magnetic tapes by Orion, and it was going to be useful to be able to read these tapes on the Atlas. I remember struggling a little at first learning the Atlas assembly code, because it was not as sophisticated as those used on the Elliott machines. It did not use symbolic names for the instruction codes like LOA for Load and STO for Store. Instead one had to remember, or have by one's side, the numeric values of these codes, 358 or whatever. Likewise, I believe there was not the facility for using symbolic names for addresses of data to the same easy extent. But after overcoming this I wrote the required handler, feeling a little bemused about it. I had not seen any of the Orion tapes that it was supposed to handle and had no means of producing them, not having access to an Orion. So I had little opportunity to test my program. But I believe it worked without problems, because I knew at least one colleague who used it and he didn't complain!

ULACS was a very pleasant environment to work in. The whole building was colourful and elegant. At one point they even redecorated the staff common room, picking out the ceiling mouldings in gilt paint. There was a piano there which more talented staff members occasionally played during the coffee breaks. The common room was supplied with many technical journals and the day's newspapers, and it was pleasant to spend a break in an easy chair surrounded by quiet discussions and an air of studious relaxation. During the summer months it was possible to obtain a key to the fenced green area in the middle of Gordon Square where there were shrubs, trees, grass and seats: something of an oasis in central London. Within one's lunchtime it was also possible to go right into the centre with its shops and city life, and to reach the Thames embankment.

The way in which programmers submitted their programs to the Atlas will seem unfamiliar to most present day computer users. Today, we sit at our personal computers, typing the text of a program directly into the machine, and saving it in a file after frequent intervals. If the development of a program takes a few days or months, we open the file and update it by typing in new information or modifying what we have done to date, and save it again in the same or a new file. These files are held on a backing store, typically the main hard disc, but possibly another medium such as a removable CD or floppy. Even before personal computers came on the scene, a team or firm of programmers would have a large central machine with terminals, either one on every programmer's desk or in a pool of workstations which the programmers would arrange to go and use. The process of producing a program was just the

same. But Atlas and the vast majority of machines at that time were monolithic, single large machines without any user terminals. For a machine the size of Atlas, the operation was so specialised that no normal programmers would ever use the machine themselves. Because up to four programs could be run at once, there had to be a way of organising the input of programs and the collection of their results so that programmers would receive back the results of their own programs and not those of someone else's. The submission of programs to the machine was organised into "jobs" and the operators fed batches of jobs to the machine. For this reason, the type of operating system used on Atlas was called a "batch" operating system. The programmer asked the computer to do a job, that of compiling or running a program and delivering the results. To that end, one had to prepare a "job description" on paper tape or punched cards. The job description stated what software was to be used by the program, maybe a compiler for compiling the program or a package for analysing survey data. It also declared what peripherals the program would require, the names of input files, the output devices and files to be generated, the maximum storage space and central processor time required, and an identification of the job and the programmer. The job would be automatically costed, and its cost would increase with the storage space reserved and so on. If the job was predicted to be quick, that is to use a small amount of processor time, it would be scheduled sooner. It was therefore important to be fairly accurate in predicting these statistics for a program. The standard turn-around time for having a program run was one day, but priority jobs would be done in half the time, so that one could get two successive submissions per day. If there was a mistake in one's program, one would therefore have to wait at least half a day and maybe a whole day before correcting it and trying again. This was a great incentive to check one's work carefully before submitting it. All the programmers were given a budget of computer usage and would have to apply to their managers for any extension. Usually the budget was enough provided one was reasonably careful. The central processor was fast and the time used by most jobs like compiling a program would be short, far less than the time that elapsed from the operator starting the job to finishing it. The default processor time allowance was one minute.

This arrangement of giving the programmers a budget of computer usage might seem a bit draconian, until one considers the enormous cost of the machine resources. An hour's time cost £950, about the equivalent of six months' of my salary. Sometimes a client user would book the entire machine for an overnight run, at a cost of several thousand pounds.

The job descriptions would be punched out on paper tape or punched cards. The program would also be prepared on paper tape or cards. Large scale data would initially be prepared on punched cards and would then usually be copied to magnetic tape. The programs and data would initially be punched out by data preparation staff, working from coding sheets that the programmers had written. Amending a deck of cards was easy enough; one could just replace a few cards with corrected ones. To amend a paper tape, one would use a teleprinter to copy it

until the point where the correction was required, punch the replacement section and continue copying from the appropriate point. Teleprinters were much the same machines that were used for sending telegrams and produced paper tapes with five holes across and a small sprocket hole used to drive the tape through the teleprinter. The combination of five potential holes were a code for the different characters that had been punched. The paper tape readers attached to the computer would read much faster than the more mechanical teleprinters and used photo-electric cells to detect the presence of the sprocket hole. This would trigger the tape reader to detect the presence or absence of holes punched in the other five positions. Thus, if the sprocket hole became blocked, the character would be missed. This was a frequent cause of paper tape reading errors.

Teleprinters were not the most convenient machines to use, being originally designed for a somewhat different purpose than preparing programs for a computer. Making small updates to a long paper tape was particularly cumbersome, because one would have to make a complete copy of the tape, stopping at exactly the right point, typing the new section and advancing over the piece of script that was to be replaced. For making small amendments, two other devices were available: splicing tape and the uni-punch. The uni-punch was a manual instrument, of high mechanical precision, so costing a hundred pounds or so, comprising a hinged block into which one could place the tape and holes through which one could insert a small punch to make individual holes in the tape. Guide holes enabled the punch to produce holes in any of the five positions in a character. There was also a groove and a cutting knife to sever the tape. Splicing tape would be used to join tapes together and to block off unwanted characters. To cancel a specific character, it was sufficient to block the sprocket hole, because that triggered the tape reader to read the other holes comprising the character.

<photos of pieces of tape, unipunch, teleprinters and flexo-writers>

Five hole paper tape was phased out in favour of eight hole tape. Programs on eight hole tape were prepared on more sophisticated “flexo-writers”. These were much more like a conventional typewriter, and more pleasant and easy to use. The use of eight holes to represent a character enabled a greater range of characters. For example, both upper and lower case letters could now be typed out; the five hole tape, even with a case shift character, could only accommodate one case of letters and digits and a few punctuation characters. Paper tape came to be preferred to punched cards for programs, because it occupied less space and weight. One also obtained a printout from the flexo-writer of the tape that had been punched out, which was easier to examine and check that reading the small lines of print that a card punch would produce along the top edge of the punched cards. Cards continued to be used for large quantities of data because it was easy to separate them into parts corresponding to items of information, and to make amendments. For this reason for a long time

programmers writing the commercial programs that handled such large chunks of data also used cards for their programs, mainly because they were familiar with the medium.

Punched cards were made from card, thicker than paper tape. Each card had eighty columns of twelve positions in which holes could be punched. Each card represented a line of text. So one was restricted to lines at most eighty characters long. The stack of cards necessary to hold any given quantity of text would be considerably heavier than the corresponding roll of paper tape. On the other hand, amending a deck of cards was easy. There was no splicing or fiddling about with a uni-punch. One just had the replacement cards punched and threw out the ones to be replaced.

There was a story about one programmer who was transporting a deck of punched cards from abroad through customs. He was stopped by the customs officer, who asked him if the cards had any commercial value. He gleefully replied that the cards themselves had very little value, but that the holes punched in them were worth quite a lot — the holes representing valuable data and the result of much labour. The customs officer thought he was trying to be facetious and hauled him off for a long interrogation.

After data preparation staff had prepared the first version of a program, the programmers usually produced any amendments and corrections themselves. The task would involve only a relatively small amount of typing. However, handling long paper tapes and decks of cards was cumbersome and prone to misreads. The chief programmer, Jules Zell, proposed that the programmers might store their programs on magnetic tape, after the first read-in off paper tape. The operating system took care of selecting which device was the source of any input to a program, such as a compiler, through the job description. The only stumbling block was editing the program when it resided on magnetic tape; we had no editing program because all editing was done on the hard-copy medium of paper tape or cards.

So my next project was to produce the first text editing program for Atlas. It was not possible to edit a program on-line, sitting at the computer and using a terminal, because there were no terminals and all computer operations were conducted under the batch operating system. One would have to work out what amendments were needed to the program and produce instructions to carry out these amendments. These instructions would have to be interpreted by an editing program. Fortunately it was possible to submit two or more successive operations to the computer as one job, so an edit of a program could immediately be followed by compiling it and even running it if the compilation was successful.

So my task was to devise the form of the editing instructions and to produce the program that would interpret them. Normally when editing a piece of text using one of the current well known word-processing programs like Word or Wordwise, one moves a cursor to the desired place in the text in order to effect a change there like deleting or replacing a few characters. With a batch editor one doesn’t have that luxury and I had to devise a means of instructing

the computer to, so to speak, home in on a desired piece of text and then do some editing on it. Years later there were a number of line editors such as vi which did some of these things, but if there were any such editors in 1966, my colleagues and I had not heard of them.

The kind of text that people would want to edit was a computer program rather than a prose document. Computers were simply not used for holding pieces of writing then. Computer programs were always prepared on some kind of coding pad with numbered lines, so I decided that the best targets for editing were line numbers and quoted chunks of text. One could instruct an imaginary cursor to move to line number 47, or to the next occurrence of the characters "DS46". Then one could delete, insert or replace a number of lines or characters. I also supplied a global editing facility so that you could replace every subsequent occurrence of one sequence of characters by another.

Preparing these editing instructions was an elaborate and clumsy process compared with today's "What you See Is What You Get" on-screen editors. For smaller programs, even the process of editing a paper tape on a flexo-writer was less hassle. There was also a learning curve involved: one had to learn my little editing language before being able to use it, whereas the programmers already knew how to use the hand card punches and flexo-writers. So in the end my editing program was used only by a few of the more dedicated programmers who had large programs to handle and who found it definitely more convenient to store them on magnetic tape.

At that time, advances in program language design were a hot topic in computer science, and were to be so for many years to come. The advantages of high-level languages like Fortran, Cobol and Algol60 were widely recognised. These languages were designed to reflect the processes that programmers wanted to carry out, rather than to be convenient ways of expressing the machine's instructions. All three of these languages were available on the Atlas and one of my colleagues, Chris Hobson, who was recruited after me and on my recommendation, spent most of his time writing and extending a huge program in Algol60 that simulated the Atlantic Ocean for the meteorological office. Performing these simulations used hours of computer time and was a fine source of revenue for ULACS. A new language was being designed by researchers at the Institute called CPL. Although advanced, like many other languages that were to be devised over the next ten or more years, it enjoyed only limited amount of use. But this culture of developing high-level languages was to some extent stimulated by the presence on the Atlas of, I believe, the first "Compiler-compiler". Brooker and Morris's Compiler-compiler enabled one to state the syntax of a language and define processes that performed the computations represented by statements in the language. A fair amount of the work that is common to practically any compiler is thus provided by the Compiler-compiler, or CC as it was called. We all regarded this facility on the Atlas as exciting and somewhat technically avant-garde. Peter Hughes suggested to me that I might want to use it in writing my editor, for it involved a compiler for a miniature language. I

looked at the report describing CC and found it rather impenetrable. I thought it would be easier to write the input and parsing routines, for that was essentially what CC provided, myself. Later I wished I had taken Peter's advice and persevered with CC, for it would have given me useful experience. But this first contact with CC was for me the beginning of an absorption with language compilers and compiler-compilers, or parser generators as they are more often called. I think that many programmers must have been similarly fascinated, for perhaps the most well known parser generator to emerge in subsequent years was called YACC, "Yet Another Compiler-Compiler".

Being a bureau, ULACS used to run some commercial programs like payroll preparation and updates, as a regular routine, once a month or even once a week. ULACS appreciated that this customer's data was valuable and backup copies were regularly made by the operations programmers. On one occasion the update program, held on paper tape, was misread simply because of a blocked sprocket hole. Instead of updating the data as intended, the program produced blank information. This "new version" of the data was then copied back onto the original as a backup. Only then did the staff discover the error – and they had carefully destroyed the original believing that they were making a backup copy of the new data! There were red faces, and the original data had to be reconstructed laboriously from a printout at the company's expense.

Many years later the analysis of programming and computer procedures was studied and a whole discipline called "hazard analysis" was developed. It was early incidents like these that prompted the whole area of computer security and hazard analysis in the nineteen seventies.

In about 1967 I was promoted to Assistant Chief Systems Programmer, which was very gratifying. From time to time other members of the systems group would come to my office to consult my advice about some difficulty they were having with their program. I would listen and make suggestions, but most of the time the process of explaining their difficulty to me would prompt the programmers to perceive the solution themselves and they would go away satisfied. Sometimes they would depart satisfied, thanking me for my assistance, but I had not understood a word of the intricate nature of their problem. Listening and asking them to explain things was usually enough to reveal a solution.

Various other changes in staff occurred around this time, including the appointment of a new managing director, Fred Gordon. He had a thoroughly commercial background, in contrast to his predecessor, Dr. Robinson. By coincidence, his surname was the same as the address of the organisation, Gordon Square. Fred Gordon capitalised on this by starting an internal house magazine called "Gordon Square". This bore a faint resemblance in style to the house magazine of the imaginary firm of Heathco, depicted by the satirical magazine Private Eye. The Prime Minister of the time was Ted Heath and Heathco symbolised the U.K. as a business; indeed, a piece of political propaganda at the time was "U.K. plc". I have the

impression that the Gordon Square magazine was written entirely by Fred Gordon himself. Our new MD brought a more commercial slant to the firm, and he drafted many advertisements promoting the services that ULACS could offer. The content of these seemed to us technical staff to be vague and raucous, “We’re the people!” one of them actually said, and we cringed somewhat, but in retrospect maybe we were being oversensitive.

There was a good deal of freedom and an easy attitude between most of the management and the staff. Most of us rarely met any customers and the dress code was relaxed. One day early in my time at ULACS the Chief Applications Programmer, Dr. Fred Dearnley, telephoned me about a forthcoming project for a customer, which involved some advanced numerical analysis, the integration of a function over an irregular surface. He had heard that I had a degree in maths and wondered if I might do this contract. We agreed to meet in the entrance foyer, which was equipped with comfortable chairs and coffee tables, to discuss the matter. We had not met before. “How shall I recognise you” he asked. The only way I could think of describing myself was to tell him what I was wearing. I liked to wear somewhat unconventional clothes at that time. I told him that I was wearing a pair of red corduroy trousers, a black shirt and a white woollen tie. “I see”, he said urbanely. “Well, I shall be wearing a three piece, navy blue, pin-striped suit”. I felt a little unnerved.

We met and talked about the project. Although I indeed had a degree in maths, I had never formally studied any numerical analysis, and did not feel too confident about tackling this particular problem. During our conversation it became apparent that Fred would quite like to take it on himself. We ended up agreeing that this was the best way forward. I think he wanted to exhaust other possibilities before metaphorically getting up from behind his manager’s desk, rolling up his sleeves and doing a job on the shop floor. But a short time later another interesting application project came my way.

At that time in 1967 London had two airports, Heathrow and Gatwick. The government was proposing to build a third airport to cope with the increasing air traffic to and from London. There was the big question of where to site the new airport. Wherever it was sited, there would be a cost. Houses and part of their neighbourhood would have to be demolished, and other buildings would lose value and have to be sound-proofed with double or triple glazing. The Board of Trade, a government department whose role is now largely carried out by the Department of Trade and Industry, were conducting some preliminary studies into the projected effect of aircraft noise on the neighbourhood of the proposed airport. They had developed an empirical formula for a nuisance value of the noise produced by the expected landings and take-offs of aircraft arriving and leaving an airport. By superimposing these values on the actual habitations surrounding the various possible sites, they could compare them and see which site produced the least overall noise nuisance.

However, the Board of Trade wanted to verify the accuracy of their empirical formula for noise nuisance. They had therefore set up noise measurement meters around the existing Heathrow airport and had conducted a house to house survey in its neighbourhood. They wanted a program written which would interpolate the noise data produced by the meters so as to produce a noise profile that could then be compared with the survey data and the formula. This comparison would be done later, again by computer, using statistical techniques. The program the Board of Trade was asking us to bid for would need to use a great deal of interpolation, which needed to be reasonably accurate but could potentially use a large amount of computer time. The computer time used in the analysis would be biggest cost factor of the job.

The problem then, at this bidding stage, was to estimate a cost for the job. This meant estimating the computer time required to run the resulting program on the data presented to it. This in turn required having a good idea about how the program was going to work, in advance of designing it for real. Only a programmer could do this, but it was the salesmen who would bid for the job. This was a common situation and the salesmen would often come into the programming offices and ask some programmer for an estimate. The salesmen worked on commission based on the sale rather than the final profitability of the job, and so it was in their interest to land contracts, even if they might subsequently make a loss. If the loss could be blamed on the inaccurate estimate of some hapless programmer, that was all right by them. The salesmen rapidly learned that they would get the lowest and least realistic estimates by asking the least experienced of the programmers. After that the salesmen could move on to catch the next contract. Not surprisingly, after a few mishaps the salesmen were required to consult only the more senior programmers. But that directive came later and the first estimate for the airport job was £50, supplied by a rather junior programmer.

Even the salesman was suspicious of this estimate, and asked another more experienced programmer. “Nonsense,” he said, “it will cost at least £130”. News of this bid reached Fred Dearnley, the chief applications programmer. He decided that someone in the systems group should do the job, and I was approached. Fred and the financial controller, who often got involved in particular bids, described the task to me and asked me what I thought of the estimate. I reckoned on using a linear interpolation method to calculate the noise levels and worked out the amount of computer time required to do the total calculations. My time would be charged out a about £4 per hour, so the computer time was going to be the main cost factor. “It will need much more than £130”, I said. I estimated £650.

This was made the basis of the bid, which was accepted. I duly set to work and wrote the main framework of the program. After a week or so the Board of Trade wanted to talk to us about how we were proposing to do the calculations. A meeting was arranged, two rather seasoned men from the Board of Trade arrived and I told them how I was using linear interpolation. They insisted that this would not produce accurate enough results. When I

asked them what method of interpolation they wanted me to use, they said that that was for me to decide. This seemed a bit of an impasse. On the one hand they were not satisfied with the accuracy of linear interpolation, but they would not agree in advance what would be accurate enough. They would not budge on this point and I felt that we were in a difficult position. If I devised another interpolation method, they might once again object to it, and without any criteria agreed in advance, they would be able to refuse it again. The next more accurate kind of interpolation to use would normally have been a polynomial, where you would suppose that the given surrounding points were on a surface with an equation described by polynomial expression such as a quadratic. With a very large number of data points this would be much more time consuming and cost orders of magnitude more. We had already signed the contract.

I had a talk with the financial controller. He said, fairly casually, that the worst that could happen was that we came out of the contract and paid them £650. I said I would try to find another solution. I went back to my desk and thought about other ways of doing interpolation that did not go as far as polynomial calculations. One of the subjects I studied in my maths degree was projective geometry. I recalled a theorem about conic sections, where one establishes a one to one correspondence between pairs of points on the conic. Joining these points produces a family of lines. A conic section is the curve you get by slicing through a cone with a plane. Depending on where the plane is placed, the curve can be an ellipse, a parabola, or a hyperbola, all called conic sections or just conics for short. However, if the plane goes through the apex of the cone, the conic degenerates into two straight lines. The theorem works just as well with a degenerate conic as with a normal one. Using this theorem, I could get a parametrisation of the points within the area bounded by any four of the data points. This could lead to a method that was a bit more accurate than straightforward linear interpolation, and would take rather longer to compute. However, it would not take nearly as long as a polynomial method. I worked out that running a program which calculated the interpolation in this way would take up more computer time, costing about £1,250. The management approached the Board of Trade with a new bid based on this figure, and rather to my surprise they accepted it. I wrote the program and it was run with the Board of Trade's data. So our estimate for this contract moved over time and a number of hiccups from £50 to £1,250, a twenty-five-fold increase.

One of the possible sites for the third airport was Stansted in Essex. This, as anyone at all familiar with London will know, was indeed chosen and is now a thriving commercial airport. Just how much influence the elaborate study conducted by the Board of Trade had on the choice I shall probably never know.

This contract revealed a general problem that has become endemic throughout the computer industry. The problem is that of eliciting the requirements of a computing task before embarking upon it. If the two men from the Board of Trade had stipulated an interpolation

method themselves, that would have been fine for ULACS and me, but they would have the danger of finding out that the computed results were, after all the expense of performing the calculation, not accurate enough. In the event, we at ULACS chose the interpolation method, but the contract could have turned out to be a great problem if the BoT discovered that, when applied to the actual data, the method of calculation was again not accurate enough; it would have been ULACS's "fault" since we had made the choice. Neither we nor the BoT knew enough about the characteristics of the data to predict with confidence what interpolation method would be sufficient. We were lucky that the method I devised turned out to be acceptable.

In retrospect, with the benefit of forty years' hindsight, what I should have done was to try to explain that the difficulty of making the technical choice was a mutual one, and to propose an initial contract in which we applied several methods to a small but representative subset of the noise data. Then the accuracy produced by the different methods could be measured and estimates made of the cost of applying each method to the whole data. After this pilot study, the BoT could select which method to use and then we could enter into a second, bigger contract to produce the results they wanted.

In other words, we should have approached the problem progressively and incrementally, rather than in a "big bang", revolutionary way. I did not learn that lesson then, and have only relatively recently come to realise that it applies to many situations. To this day, many expensive computing disasters occur, usually when a particularly large system is being procured, and usually by a government agency. The difficulty always arises when the precise nature of the environment in which the required software is to operate is not fully known or understood.

Elliott's did not have photocopiers. Although they were not available as practical commercial machines, the process had been invented. I first came across a photocopy in my last year at university in 1961 or 1962. A local small enterprise offered a photocopying service to produce the programmes of a university society that I was running. The process was called "offset xerography". However, the results were so speckled and distorted that I did not take the offer up. By 1966 though, reasonably successful, commercial machines had become available, and ULACS had one. All photocopying machines at that stage were manufactured by Rank-Xerox, presumably because they held the patent, which had not yet expired. The Xerox part of the company produced the photocopiers, so the machines were called "Xerox machines" and the word became a verb: "I'll just go and Xerox this document". The company were temporally in the happy position of their name standing for the type of product, like Hoover for vacuum cleaners and Biro for ball-point pens. Photocopying machines were certainly something of a novelty at ULACS, and the same was true in most office environments. I remember a cartoon in which a secretary is painting her nails and an angry boss says to her "Don't just sit there – go and Xerox something!". It was some time before

they became indispensable, simply because, not having had them for generations of office life, it took time for practices to mutate to become dependent on them. If you are used to not expecting to have copies of documents, you don't start using a copying machine in a routine fashion as soon as it becomes available. The single photocopier at ULACS was often idle for long periods, despite serving the needs of over 100 people.

The staff common room, which the Institute and ULACS shared, was equipped not only with comfortable arm chairs and coffee and tea, but all the journals on computer science. In those days there weren't many of these. The British Computer Society published the Computer Journal and the Computer Bulletin, and the American ACM, the Association of Computing Machinery, published several titles, the Journal of the ACM, Communications of the ACM, and a few other specialist magazines. The ACM Journal had an Algorithms Supplement, which described new algorithms for performing particular calculations or solving well known problems. An algorithm is a step by step mechanical process, exactly what a computer program performs. These algorithms were usually published in Algol60 or Fortran or, more frequently as time went on, in pseudo-code, which is an idealised high level computer language, understandable by a human reader and easily translated by hand into a real computer language. Devising new algorithms and numerical analysis in general, which is the study of computational methods of mathematical operations, were very much a principal occupation of computer science in the 1960s. At the same time, the Institute used to hold seminars and these were often about new methods in numerical analysis. Methods of numerical integration and differentiation were a popular topic. I found these topics very interesting and often attended the seminars.

One of the ways of solving differential equations using a computer is to turn them into integral equations and use a variation of a method of Isaac Newton, doing calculations of the function at discrete intervals and interpolating between them, either linearly or using a quadratic or higher order polynomial. Second order, i.e. quadratic, and fourth order were popular approaches. Going to higher orders is not usually cost effective. The interval between calculations of the function determines the accuracy of the process. A frequent difficulty is that the function's regularity may vary a great deal from one point to another. A technique which fascinated me was a variation on a second order technique, called Runge-Kutta. The original method was devised by two German mathematicians, C. Runge and M. W. Kutta in 1901, before the age of computers. A crater on the moon is named after Runge. The variation on the Runge-Kutta method was invented by Merson, so this variable interval method was called Runge-Kutta-Merson. After each calculation, an estimate would be made of the error. If this error exceeded a certain value, the interval would be halved and the operation repeated. If the error was less than a certain smaller amount, the interval would be doubled. In this way the integration process would stride ahead with big intervals over the regular features of the function and crawl meticulously over the more difficult terrain. Since then several other

adaptive step size methods have been devised by Richardson and Fehlberg.[1] Further developments have been named Cash-Karp and Dormand-Prince. I was eager to program the Runge-Kutta-Merson algorithm and apply it to a real problem.[W]

In 1967 the Admiralty approached ULACS and asked us if we could solve a set of differential equations. Some of the constant factors in the equations would be presented as parameters, that is as data. So would the limits over which the variable was to range, and other information such as tables to be printed out and graphs of results to be plotted. The Admiralty would keep the actual data to themselves and only run it on the program when we had finished writing it. They were careful not to tell us anything about the purpose of the program, something that concerned me a little; I hoped that it was not associated with any too malicious weaponry, but I have to admit I never discovered to what use my program was put.

We arranged a meeting with the Admiralty representative, a pleasant, retiring, red-bearded man who sat back in his chair and listened while I described the principles of Runge-Kutta-Merson and its advantages. We agreed to proceed. I wrote the program in Fortran, which was a most suitable language for the job. The solving of a set of differential or integral equations required repetitious arithmetic calculations, for which Fortran was ideal. Algol60 could handle processes with a complex structure better than Fortran, but the structure of this program was straightforward, and for repetitive calculations, Fortran would be faster and therefore use less computer time, which was an important criterion, given the cost of computing. Also, the Fortran compiler was better geared to printing out results in a prescribed layout.

Because I did not have any of the customer's real data to test the program, I had to make up my own. I had no idea at all of what would be typical values, so I just invented them out of the blue. I also plucked a value for the maximum permitted error out of the air. I got the program working, but I decided that I had better consult the customer about the maximum error value. I wrote him a letter explaining the issue and suggesting that I included it as a final parameter in the data. He wrote back agreeing that this seemed a "very reasonable" approach. We had a last meeting in which I demonstrated the results of running the program, with data that I had invented off the top of my head. I explained to the customer how I had to make a guess at this, and how I had no idea of whether the data values were realistic. He assured me that they were quite realistic, indeed he was surprised that I had hit upon quite typical values of his "secret" information!

So, one more happy customer. I wonder what he used my program for. I shall most probably never know. Since graduating from university five years earlier I had come increasingly to the view that I did not want my work to be used for military purposes or for contributing to the manufacture or design of weapons. The popular perception is that one is full of ideals

[1] See Fehlberg 1969.

when young, but these fade away as one gets older and wiser or more cynical. In my case, the reverse has happened. Before university I had spent a year working at Texas Instruments in electronic semiconductor circuit design, and had unquestioningly worked on the control system for an anti-tank missile. My mentor on that project, Bhiku Unvala, discussed the ethics of such work, and was marginally willing to do it, since it was a defensive weapon rather than an offensive one. A debatable point perhaps, but such questions were quite novel to me at the time, and I regarded them as perhaps rather eccentric and quaint. Later in a vacation job with ICT I worked on the logic design of a military computer, again without too many qualms about the desirability of such work. But when I joined Elliott's I stipulated that I did not want to work on anything military. There was no problem about this. Although Elliott's had divisions doing work for the military, there was plenty of opportunity in the civil sector. Previously, while an undergraduate, I had supported CND and had helped to campaign for the abolition of capital punishment. My interest in philosophical and ethical questions has continued ever since.

The 1960s were not just a time of individual liberation, but a decade in which the public conscience was awakening, and stirred to ask ethical questions of many civic practices. The Wolfenden Report had been published in 1957 but only ten years later in 1967 was homosexuality between consenting adult males finally decriminalised. Capital punishment was formally abolished, although no-one had been executed for several years, in anticipation of its end. In the UK, it was still legal to discriminate against someone on the grounds of race or colour, and advertisements for accommodation and jobs still often bore the stipulation "no coloureds", which would seem shocking today; also, equally often, "no Irish", which would simply be perplexing now. In the mid sixties, a considerable movement was afoot to get rid of racial discrimination, and a colleague at ULACS, Gurmukh Singh, introduced me to the Camden Committee for Community Relations. We would test night clubs and other places to see if they practised discrimination, and bring them to the notice of the authorities if so. The war in Vietnam was taking place, and it prompted a lot of moral debate, including in the Gordon Square common room; there were several staff from the USA in both institutions, most of them against the war but just a few for it.

There were some interesting and eccentric individuals working at ULACS and the Institute. One of our programmers, brilliant but highly strung, would become extremely frustrated if his programs failed to work as they should. At one time he began to believe that unseen blackguards were creeping into the premises at the dead of night and altering his results. We were always advocating that programmers should document their programs, which then meant mainly writing a document explaining how the program worked. This was for the benefit of any other programmers who might take over the work, especially important in a time of high staff turnover. Another of our programmers was soon to leave and she was urged to document her recent work. She did so, but after she had left, we found that she had written

her documentation in Hebrew. In years to come, firms would have "quality systems", sets of rules which ensured that work was reviewed and signed off as being of adequate quality. Such procedures would have prevented this, admittedly humorous, caprice. One day David Powell-Evans, an urbane, senior and perhaps the most competent of the applications programmers, arrived at work carrying a climbing rope and rucksack festooned with karabiners and other equipment, in preparation for a weekend of rock climbing. After his colleagues had asked him several questions about climbing techniques, he demonstrated abseiling by doing so from the top floor down the front of the building. Passers by found this mildly intriguing.

In the ambience of the sixties, an era of new music, art and ideas, computer programmers were still ambivalent about whether they were artists or engineers, individuals practising individual skills and expressing elegance in their creations, or followers of disciplines aimed at reliability and safety. There was to be a great movement towards repeatable quality, "ego-less" programming and the maturing of software programming as an engineering discipline.

But enough of philosophy and personalities, for the present at any rate. More differential and integral calculus was to beckon me. Before digital computers came on the scene, analogue computers were used for solving mathematical problems, especially for solving equations and calculus. Amplifiers, essentially the same as you find in hi-fi and radio circuits, can multiply voltages together and the use of ohm's law can be arranged to add them. Two simple electronic components, capacitors and inductances, give the building blocks of calculus; the voltage across a capacitor is the integral of the current flowing into it over time, and across an inductance it is the rate of change or differential of the current. Engineers could patch together basic electronic units that added, multiplied, differentiated etc., so as to solve a set of differential or integral equations. These would be used to simulate mechanical and other systems. They had been used a great deal in engineering, especially the aircraft industry. But now they were becoming obsolete and engineers were turning their eyes towards digital computers, which were potentially more accurate and perhaps easier to program. It would certainly be easier to repeat calculations.

Elliott's, another branch of the firm I had worked for previously, wanted to simulate an analogue computer on the Atlas. We had a number of meetings with them in which they showed us the kinds of "program" they would write for an analogue computer. These programs consisted of sets of very short simultaneous equations, something like the following:

$$
\begin{aligned}
x &= a + b \\
a &= \int y \, \mathrm{d}t \\
y &= x / a \\
b &= 3.4 \int x \, \mathrm{d}t
\end{aligned}
$$

On an analogue computer these equations would be programmed by plugging together an adder, a divider, a multiplier and two integrators. Any of the terminals of the units, which would represent one of the variables, could be attached to an oscilloscope and the voltage on it examined.
<block diagram of adder, divider, 2 integrators with connectors labelled with variables, and maybe an oscilloscope connected by a trailing line to one variable.>

So our task was to take the whole repertoire of these analogue computer instructions and write a compiler for them. The compiler would have to turn the instructions into an Atlas machine code program that would carry out calculations equivalent to the analogue program. One interesting and unusual feature was that these equations are much more like mathematical equations. They are declarative in that they declare what the definitions of the x, y, a, b are. It does not matter in what order they are written down, or in what order the engineer connects the analogue units together. They will become active only when they are all connected and the circuit switched on. This is totally different from a conventional computer program, where the calculations are in principle done in the order they are written down. So, having translated the equations into machine code, the compiler has to sort them. Some equations, like the one defining x, require other variables, a and b, to be calculated first. All the variables are in fact, functions of time and have values which vary as time proceeds. The results of integrals do not have to be calculated in advance, so the little program above would need to be sorted as follows:

$$a = \int y \, \mathrm{d}t$$
$$b = 3.4 \int x \, \mathrm{d}t$$
$$x = a + b$$
$$y = x / a$$

These equations then have to be calculated repetitively in a loop, with "time" being incremented in steps in each repetition. The task therefore required compiler writing skills, and numerical analysis to perform the solution of simultaneous integral equations: a task for the systems group, especially for the compiler design. Also, with possibly several hundred instructions being presented to the compiler, the task of sorting them into an appropriate order was not as simple as it might seem. It required a technique called precedence analysis. x depends on a and b, and y depends on x and a. This dependency forms a directed graph, a number of nodes (the variables) connected by lines with a direction, the dependencies. This graph can be represented in the computer and there are programming techniques for "walking" through the graph and finding its "leaves", the nodes or variables on which nothing depends, and tracking through the nodes in order of precedence. From this directed graph, the instructions could be arranged into an order of precedence.
There mustn't be any circular dependencies amongst the variables, which means that the graph must be acyclic, without cycles or loops. Otherwise the equations could not be sorted

into order, but more importantly, they could not be successfully programmed. One can find the same requirement in a spreadsheet today. If you make a set of cells contain expressions which are circularly dependent, the spreadsheet package will object.

Conventional computer programs can look insidiously like mathematical equations, but they are substantially different. It is quite in order in a program to write something like:

$$x = x + 1$$

meaning add 1 to the existing value of x and write it back to x . x is not a variable in the mathematical sense but a name that is associated with a value. The equations in analogue programs look even more like mathematics, but are still not the same, even though their order is not significant. The equations

$$x = y - 1$$
$$y = 3x$$

have a mathematical solution (0.5 and 1.5) but if you connected the units of an analogue computer following those formulas they might well oscillate wildly. (Try putting these formulas into two cells of a spreadsheet and see what happens.) So our compiler could have the additional advantage over an analogue computer; it could catch erroneous programs and report on them.

Peter Hughes and I joined forces in producing this compiler. Peter let me run the project, which was generous of him, seeing that he was my manager. Peter wrote the front end of the compiler using Brooker and Morris's Compiler-Compiler. I designed the object code that is generated on translating the instructions, the algorithms for sorting them and for doing the integration. I estimated four months for completing the project. PERT charts, Project Evaluation and Review Technique, had recently been invented, and I made a simple one of these to plan the project and estimate the time we would take. We completed the program and delivered it ten days before the deadline. I have to admit that I never repeated this, delivering a project so early, although I have completed plenty on time. In years to come software projects were to gain some notoriety for being late and over budget, although not, to my mind, in reality any worse than in other industries like civil engineering.

When we were part way through the project we were asked to go to Elliott's and give a presentation about the program and what it would be able to do. The Elliott's works was in Farnborough, not the Borehamwood location where I had worked before. When we arrived I was perturbed to see that there were a number of soldiers in uniform on the site. I had not realised that Elliott's was so hand in hand with the army. Peter and I went to the office of a manager, David Morgan. His office was large, with a conference table and a cabinet beside it. Our presentation was quite informal, with occasional writing on a whiteboard. Overhead projectors were not in common use yet. I found myself working with David Morgan more than twenty years later and got to know him very well. On this occasion he was fascinated

when we described the principles of the Compiler-Compiler. At lunchtime he opened the cabinet beside the conference table and revealed an array of bottles, gin, vodka, mixers and glasses. We had gin and tonic before lunch. I was most impressed. A cocktail cabinet in his office, supplied by his company for business entertaining! I thought: this man has arrived!

The compiler we wrote, which was called SLANG, for "simulation language", seemed to work and produce believable results. Again, supplying it with test data and checking the results was a somewhat tricky and uncertain process, but the solutions to some sample equations that we invented ourselves showed the expected pattern of values. The time came for Elliott's to use SLANG for real. They tried a sample of their own data and were satisfied with the results, and then presented the full range of values. This was to take an overnight run of the Atlas at £950 per hour. My salary at the time was £2,000 per year, so Elliott's were spending the equivalent of a couple of years of my salary on a single computer run, using the software I had designed. I was more than a little apprehensive. When running the Atlas, the operators would perform a "restart" every couple of hours, usually in order to do some minor hardware maintenance. The operating system would automatically back up the state of the machine, memory, registers and so on, to a special magnetic tape every few minutes and when the operators did a restart, the computer would resume from the last recorded state. A couple of restarts occurred during the long run of the SLANG program that took place overnight. Since the huge long tables of results of the program's calculations were being produced continuously throughout the run of the program, this meant that over the restarts, a few results were repeated. I was horrified to see the next morning that these "repeated" results were very slightly different, in the third or fourth decimal place. The repeated calculations should have given identical results. Something was wrong. I discovered that I had not initialised one of the variables in my program properly. If one fails to do this, the variable can have any random value at the start of the program, so the program can produce different results each time it is run. Usually, if there is a fault in a program, the results are haywire and it is easy to spot that something has gone wrong. It is highly unusual for a program to produce errors that are out by a fraction of a percent because of a fault like this.

We explained the difficulty to the customer. The error was easy for me to put right, but my heart was in my mouth while we waited for Elliott's reaction. They could demand a rerun free of charge. I was most relieved when they said that such a small deviation was not going to affect their subsequent analysis of the results to any significant degree. I breathed a sigh of relief.

Although we had written the SLANG compiler for a specific customer on contract, the compiler remained the property of ULACS. The reason for this was a pragmatic one: Elliott's could only use the program on an Atlas machine and the London Atlas was the only one which was regularly used for commercial hire. Even then, the SLANG compiler would have required some minor modifications before it could run on the Manchester Atlas or the

Cambridge Titan. Quite simply, there was no point in Elliott's, so to speak, removing the compiler from us, because they would not be able to use it anywhere else. Today, and for the last twenty years or more, everyone is much more commercially conscious. A customer in that situation would insist on royalties if we used the program, which they had paid for to be developed, for profit with another client. But then, in 1968, we were free to look for more users and customers for SLANG.

I was fascinated by SLANG. Here was a program, that I had designed, that in effect could solve any reasonably well behaved set of differential or integral equations that you could throw at it. I felt that it would be more satisfying if one could write fuller, more general expressions on the right hand sides of the equations, instead of the very short forms that reflected the old analogue computer elements. Then, instead of the four little equations for x , y , a , and b shown previously, one could for example write:

$$x = \int y \, \mathrm{d}t + 3.4 \int x \, \mathrm{d}t$$
$$y = x / \int y \, \mathrm{d}t$$

This would look much more like conventional mathematics and be less irksome to write out. I could see how to write a compiler to do this, using quite standard compiler techniques for analysing expressions.

I suggested this enhancement to Elliott's. They weren't particularly enthusiastic. Probably they were accustomed to the form of the analogue computer programs and so did not see much advantage in the change. However, they agreed to a further contract. The cost to them would be very small compared to the amount they were spending on computer time to actually solve their equations; it was "just" a piece of software development. Today the economics are quite the reverse. Hardware is cheap, and so computer time is very cheap, but skilled labour is expensive.

I produced the more advanced version of SLANG, again with Peter Hughes's assistance. After the success with the first version, I must have been suffering from a bit of overconfidence. The enhancements turned out to be a little more difficult to do than I had anticipated. But we delivered, a little late this time. The customer was not bothered about the two or three weeks delay, presumably because they had no immediate plans to use the "Mark II".

Together with one of the salesmen, Bill Musker, I tried to find other customers for SLANG. Bill was probably the best salesman we had. The others used not to do much more than issue an advertisement from time to time and sit at their desks waiting for the telephone to ring. Bill was much more proactive. At his instigation several potential customers came to visit ULACS and talked to me about their application. One was a doctor who was having to calculate the irradiation dosages for a cancer patient. He described how he had to make these long calculations which determined the intensity of radiation in the diseased part of the

patient's body, and to make sure that the intensity in nearby sensitive areas were low enough to be safe. I was alarmed to think that the use of the SLANG compiler could result in such a life critical procedure. But when he heard how much it would cost to solve his equations, he sorrowfully said that it would be way beyond any budget he had. I felt a mixture of relief and disappointment.

Other potential customers arrived from time to time. They all had military applications. I was saddened by this. I began to think that perhaps differential equations were not the neutral, intellectual concept that I had thought them to be, but had an inevitable, aggressive character. In the event, no further contracts arrived for the use of SLANG.

A friend of mine from my undergraduate days, Michael Digby, had started working for the computer manufacturer English Electric, which later merged with Leo and Marconi. English Electric produced a physically large mainframe computer, the KDF9, and this was the mainstay of its computer business. Mike worked on a vehicle scheduling program for them. The general vehicle scheduling problem is a classical and practically important problem in computing. If you have a fleet of vehicles which have, between them, to visit a list of locations, shops for example, assigning suitable routes to the vehicles so that they cover the least distance and therefore use the minimum amount of fuel, is a highly useful problem to solve. A computer can be programmed to solve this. The difficulty is that as you add more destinations for the vehicles to visit, the amount of computing time required to solve the problem increases disproportionately, indeed, exponentially, and soon becomes impracticably long, even with today's central processor speeds. So various techniques have over the years been devised to work round this difficulty, most of them being to find reasonably efficient but sub-optimal solutions that require less computing time. IBM had a proprietary program for doing this, but Mike had some ideas of his own. He left English Electric and set up his own company, for a long time at first working on his own. He developed a program for vehicle scheduling that out-performed the IBM product by a few percent. This was sufficient to prove attractive to very large organisations that had equally large fleets of vehicles. He won contracts to supply firms like Unilever and Whitbread.

Licensing his program to a software house was not initially what Mike planned to do, but there came a point when he thought that doing so could be to his commercial advantage. Knowing that I worked for ULACS, he approached Fred Gordon. Fred later called me to his office. "This man Digby: is he reliable?" he almost barked. Well, yes, I assured him, he operates perfectly ethically but he is not a charity. He has a business to run. Fred Gordon entirely understood this, being very much of a commercial turn of mind himself. So they reached an agreement and Mike leased his program to ULACS for customers who had vehicle applications to use. Fred and Mike decided to market his program under the name of "RouteMaster". A new fleet of double-decker buses had been introduced into London's transport system and these were called "RouteMaster". These buses were new, shiny and

popular, and in some ways the latest thing on the London scene. So the name was evocative and upbeat. The buses had very rounded lines and a large open platform on to which one could jump after running for a moving bus. Safety was less of an issue then: trains and buses now have doors electrically operated so that no-one can accidentally fall out while in motion. I sometimes think that we have become a bit too safety-conscious these days. Citizens should be allowed to take risks, and on their own heads should it be. Nonetheless, there are still a few RouteMaster buses operating today, venerable, almost vintage vehicles and reminiscent of the sixties.

Another new appearance on the London scene was the Post Office Tower, now the BT Tower. The PO Tower had a revolving restaurant near its summit, closed now for many years because of a bomb planted there many years ago by the IRA. One food critic of the time described it, damning with faint praise, as "by far the best 650 foot high revolving restaurant in London". However, its novelty value was also very high and to my delight, Mike invited me and a few others whom he wanted to thank, to lunch in the revolving restaurant. The views were magnificent and the floor in sections revolved slowly. The engines driving the revolutions vibrated slightly through the floor and each time I looked up after a conversation with my dining neighbour, a different scene presented itself. After a time I began to experience mild travel sickness, but not enough to spoil my meal. The waiters were a little pretentious, addressing one as "Monsieur" in a London accent. But the experience was unique and memorable, never to be repeated. Thank you Mike.

The Atlas was becoming rather aged and expensive to run. Technology was moving on, as it does, and the Institute had for some time been thinking that it should get itself a more up to date machine. CDC, the Computer Development Corporation, was in the business of producing large main-frame computers, as were IBM and other manufacturers. Various changes in organisation began to take place. The Institute of Computer Science became absorbed into Birkbeck College and bought a CDC 6600 machine. Many of the staff from ULACS and the Institute, including Peter Hughes, moved to the new organisation and its computer. I remained behind and was made Chief Systems Programmer. After some six months, the technical staff within ULACS and the original Institute had become considerably depleted. The intellectual environment was not what it had been and I began to get itchy feet.

Chapter 3 Workers in Control

I was disappointed that the original ULACS was, so to speak, evaporating into thin air around me. It had been an organisation of great character. One effort in the neighbouring Institute looked especially appealing. David Hendry had an idea for a more efficient way of writing compilers. Compilers had fascinated me for some years, ever since I had worked right next to the Algol60 development at Elliott's. Yet more fascinating was the notion of Brooker and

Morris's Compiler-Compiler, which was used in several projects in the Institute and in ULACS.

A compiler translates the "source" language, that is the language in which the programmer composes a program on a coding sheet, into a "target" language, which is usually the machine code of the machine on which the translated program is to run. David Hendry's idea was to divide every compiler into two sections, the front end and the back end. The front end would translate the source language into a standard intermediate code. The back end would translate the intermediate code into the target language, that is the target machine code. Then, by bolting the two together, one has a compiler for the source language producing code for the target machine. The intermediate language would be simple, rather like the code of a typical machine, so the back end would not have too much work to do and would, one hopes, be relatively straightforward to write. Now, if one has to write compilers for Algol60 and Fortran, say, to produce code on three different machines, then one has simply to write two front ends and three back ends, obtaining six compilers. If a requirement comes for a compiler for either of these languages for yet another machine, all one has to do is to write another back end.

This technique is commonplace now, but in 1968 it was new. The idea of an independent firm writing compilers for another manufacturer's computer was in any case certainly unusual at that time too. There were few independent software firms, so computer manufacturers would nearly always write their own language compilers and other systems software. And they certainly would not be writing software for their rivals' machines, so until then there had been little call for software to run on different machines, that is, to be "portable". But things were changing. Computers were becoming smaller and more affordable. The first minicomputers were beginning to arrive on the scene and independent software houses, fairly small organisations that could afford to buy such machines for themselves, were also springing up. Furthermore, the industry was coming increasingly to recognise the value of high level languages like Algol60, with their greater portability across machines and the improved ease of understanding programs.

Because of the limited storage size of minicomputers such as the Digico Micro-16, the intermediate code often had to be output from the front end onto a temporary medium – usually paper tape – and read back in by the back end. We take it for granted now that the binary representations of characters, letters of the alphabet and numerical digits especially, are universally the same in every computer, whether as stored internally or on magnetic media like floppy disks and CD ROMs. But it has not always been the case. At first almost every manufacturer defined its own way of representing a character on paper tape. Standards began to be defined, but there were still several different paper tape formats defined by rival standards institutions. Eventually the representation known as ASCII became the accepted standard, but in 1969 different machines still used various representations.

Radics wanted their compilers to be as portable across different machines as possible. Fortunately, all the paper tape codes agreed about the representation of the decimal digits 0 – 9. So the intermediate language for the compilers was coded into decimal digits and the tapes that conveyed the intermediate code from front end to back end were called decimal coded tapes. We probably would have been able to use alphabetic characters too, but we decided that a decimal code was safer just in case we encountered a really obscure paper tape code that had different alphabetic coding.

Another twist to David's idea was that the front and back ends would be written in a special language of his own invention, called BCL. There was already a language ACL, Atlas Commercial Language, which was used on Atlas for commercial applications. It was a neat straightforward language using some of the better ideas from Cobol and Fortran. The first incarnation of BCL ran on Atlas, and so it was called BCL as a kind of successor to ACL, although having a very different purpose. Since the compiler for BCL was itself written using the front end – back end technique, any compiler written in BCL could be transported to another machine simply by writing a new back end for the BCL compiler.

The first compiler for BCL was written using the Brooker and Morris Compiler-Compiler and ran on Atlas. A second version was written in BCL itself, which could be translated using the first version. From that point on the first compiler could be thrown away and the second one used. Then, by writing a back end for another machine, the whole technology could be transported. David's ambition was to start an independent software house, but he began with a small group within the Institute. We had a few discussions and I was keen to join him. There was one small difficulty: ULACS and the Institute had a mutual non-poaching policy. This was sensible enough, since the two organisations were housed in the same building. Without such a policy, there could be some chaos. So, to avoid this difficulty, I had to apply for a job elsewhere. Then, David could come to the rescue and offer me a job, with ULACS' agreement. That way, since I was likely to leave anyway, I would be kept within the fold, so to speak. This ploy was and is used frequently when different divisions within one company have a mutual no-poaching policy. So I had to go through the hoop of applying to CAP, Computer Analysts and Programmers, in answer to one of their advertisements. Programmers were in short supply, and software houses like CAP had recruitment advertisements in press almost continuously. CAP was one of the first independent software houses, and had established a high reputation for itself. They put particular emphasis on maintaining strong business ethics, especially in the area of client confidentiality.

I had an interview with the managing director and founder of CAP, Alex d'Agapeyeff. He told me how they pursued their confidentiality policy. The team working for one particular client would be separated from teams working for others, especially if they were engaged on similar projects. Compilers were becoming a frequent task, with high level languages like Fortran, Cobol and Algol60 becoming used more widely. I asked him what would happen if

there were two contracts to supply an Algol60 compiler, for example. Wouldn't it save a lot of time and be less prone to error if the two projects shared design ideas at least? He replied that, on the contrary, in that situation the two teams would not be allowed to speak to each other, to preserve confidentiality. We had quite a discussion on that topic, and he admitted that it was a question that was being debated quite strongly within his firm. CAP offered me a job at an increase in salary, and after talking to the ULACS managing director, Fred Gordon, David Hendry offered me a job in his team. Fred and I had a talk about this and he said that he would prefer to see me in the Institute rather than with a rival software house. So I joined the Institute of Computer Science, working in David's group. After a few weeks David set up his new company. All of us in his group changed employers, once again in my case, to the new company, RADICS – Research and Development In Computer Systems.

I feel I should now, thirty eight years later, apologise to Alex d'Agapeyeff for taking up his time at an interview under rather false pretences. My apology is belated, because Alex d'Agapeyeff died in 2003, having achieved considerable distinction in the commercial world of software engineering, including being President of the British Computer Society from 1970 to 1971. I did, however, find the interview an instructive experience and even that brief hour's encounter has added some more to my perspective on the evolution of software organisations and the work they carry out. CAP thrived for many years and later underwent various splits and mergers. CAP-Gemini is a successful international organisation today. Many organisations had and continue to have non-poaching agreements between their divisions, and similar agreements are routinely made between a supplier, especially a consultancy house, and its clients. Obtaining an offer of another job elsewhere was a frequent and widespread means of overcoming these barriers, especially when the demand for staff far exceeded the supply.

RADICS moved out of Gordon Square into premises of their own in Drayton House, Euston Road. This building was let to us by the Society of Friends and one term of the lease was that we should not use the building for the manufacture of arms, alcohol or tobacco. We found this mildly amusing, rather quaint perhaps, and thought that these terms were most unlikely to restrict us. About ten of us occupied the lofty and rather gloomy rooms. David had radical socialist ideas about how an industrial organisation should run itself. He wanted RADICS to operate under a system of workers' control. Every member of staff, administrative, technical and managerial, had an equal £1 share in the company. Share-holders' meetings tended to be indistinguishable from staff meetings. Policy decisions were taken democratically, everyone having an equal vote. I remember one decision that was rapid and unanimous: Christmas Eve should be a company holiday! But most of the decisions were reached only after a lengthy debate. Looking back on RADICS, I think we spent rather long periods in staff/share-holders' meetings, time not very productively spent from a commercial point of view. It was quite a large overhead and could not have improved the company's productivity.

The first contract was with Digico, who manufactured one of the first minicomputers, the Micro-16. Fortran and Algol60 compilers were required. A bi-product of this contract was that we acquired one of these computers and it stood as a general work-horse in the corridor beside the offices. It was fun working in a company that had just started up. We had to acquire everything from scratch – coffee spoons, kettle, stationery, typewriters for the secretaries, desks and chairs, you name it. I have to say that the office premises were a bit dismal, but our enthusiasm and excitement for the new enterprise was enough to compensate for many environmental disadvantages.

We numbered about twelve. One team worked on the front end for the Fortran compiler, another for that of the Algol60 compiler, and a third team for the back end for the Digico Micro-16. David Hendry was the managing director and Marshall Harris, also a director, was marketing manager, with the important task of finding further work. High level languages, like Algol60 and Fortran, were becoming more and more important in the industry's eyes. A language is "high level" if it is designed to express the solution to a typical problem, instead of being a more or less convenient way of writing instructions for a machine. With the increasing popularity of high level languages, David believed that Radics' flexible compiler technique would lead to prosperity for the firm and the workers who owned it. "Those who control the languages control the world!" he would say, with some hyperbole. David's charisma and the combined sense of new enterprise and new technology fired us all with enthusiasm.

David had done a little initial work on the Algol60 front end, and I was given the task of completing it, helped by an able assistant, Pat Whalley. When I was at Elliott's, I had witnessed the team there produce the first commercial Algol60 compiler, in the next room, so to speak, something that made me both excited and envious. Now at last I was to design and produce my own. I was both delighted and a little overawed by this prospect. Algol60 was a language which had many features that were difficult to implement. The language was "block structured", which meant that new areas of data could be defined while a program was running. The allocation of a program's working space was therefore dynamic. It could not be decided in advance by the compiler. Pieces of code could be designed as procedures or functions, which could be called from other parts of the program, much like subroutines in a low level language program. But Algol60 procedures and functions could be recursive, that is, they could call themselves or call each other mutually. For some problems this was a superb feature. The usual example is that of a factorial function:

```
integer procedure factorial(n); integer n;
factorial := if n = 0 then 1 else n * factorial(n-1)
```

In mathematics the factorial function is written **n!** For all positive values of **n** the factorial is the product of all the numbers up to and including **n** multiplied together. **0!** is defined as **1**. In

fact this example of recursion is not a particularly good one because it is perfectly easy to program an efficient factorial function without using recursion, but it illustrates the principle neatly. Compiling recursive procedures and functions is a bit tricky because each time the procedure or function is called, new working space has to be allocated for it, and de-allocated after each call is finished.

Recursion on its own is nonetheless reasonably easy to cater for in a compiler. In Algol60 the whole feature becomes much more complicated because one is allowed to jump out of the body of a procedure using a "Go To" instruction. In high level languages Go To instructions are really relics of machine code programming. In all machine codes, there are jump instructions, which alter the path of control. Instead of the computer obeying the next instruction in sequence, it obeys the instruction whose address is given in the jump instruction. Furthermore, in Algol60 and some other high level languages, labels, which are attached to instructions and can be the destination of Go To instructions, can be passed as parameters to procedures and functions. So working out just how much recursive unwinding is required when jumping out of a recursively called procedure or function is a bit of a nightmare.

Using Go To statements in Algol60 and other high level languages often led to programs being very difficult to understand and analyse by a human reader. In fact, in 1968 the renowned computer scientist Edsger Dijkstra published a letter to the Communications of the ACM with the title "Go To Statement Considered Harmful"[1]. This two page letter caused immense controversy at the time and led to a whole discipline of how to write software clearly and effectively. This discipline was called "Structured Programming". To this day the Go To statement is notable by its absence in modern programming languages such as Java.

Partly because of the difficulties of combining recursion, Go To statements and dynamic storage allocation, several different "levels" of Algol60 were defined by ECMA, the European Computer Manufacturers' Association. Each level of the language contained different features. The highest level contained all the features of Algol60. The lowest level, level 0, did not contain recursion or dynamic storage allocation. Because the Digico micro-16 was a small machine, we were only asked to provide a level 0 compiler for it, but for two other contracts, with ICL and Honeywell, we were asked to produce a level 2 compiler. Level 2 contained virtually all the principal features of Algol60, including recursion and dynamic storage allocation.

So Pat and I set about designing and writing the compiler for Algol60 level 2. I thought we could do most of the work for the level 2 compiler and then produce a new version, removing some features to obtain the level 0 version for the Micro-16. But I found that the level 0

[1]See Dijkstra 1968.

compiler was so much simpler in requirements that producing it was in reality a separate project, albeit a much easier one.

The Digico micro-16 computer had a mere 32 kilobytes of main store. A modern PC with its two-level store shared between random access memory and hard disc has, by contrast, typically 160 gigabytes of which 512 megabytes are in RAM. So today's RAM is sixteen thousand times the size and the whole main store is five million times the size of the main store of the Micro-16. Furthermore, by the time this book is published, these figures will no doubt be out of date; the factors will be even greater compared to what is available as you read this. The Elliott 803 had a main store of a similar size to that of the Digico micro-16, so I recalled the strategy used by the Elliott's Algol60 team to shoehorn the compiler into such a limited space, and followed their example.

The Algol60 language is full of opportunities for "forward referencing". As you, or the computer, read the program, there can be references to elements of the program which have not yet been fully defined. The compilation of these references cannot be completed until the definitions have been found, later on in the script of the program. This means that the compiler has to remember a great deal during the input of the program script. With a small amount of main storage, only very small programs can be compiled.

The way round this problem was to compile a program in two passes. The traditional method was for the compiler to read the program two, or maybe more, times. On the first pass the definitions would be read and stored in some codified form. On the second pass the full compilation would be carried out, with the knowledge, so to speak, of all the definitions. In this way, the compiler had much less to "remember" during the compilation process and much larger programs could be compiled.

This was a rather crude approach and had another disadvantage. Programs were still mostly prepared on paper tape. After reading the program the first time, the tape will have been dumped out of the tape reader into a tape bin. To read the program a second time, the tape has to be rewound from the bin on to a spool and fed into the tape reader again. For long programs this takes some time, during which the computer would be idle, unless it was a sophisticated time sharing system like Atlas. The Elliott's team had another idea. Instead of rereading the program, during the first pass the compiler would output a partially compiled version of the program on to secondary storage, typically paper tape. By the end of the first pass, all definitions would have been found and output on to the intermediate tape. This tape would be read back in on the second pass, in reverse direction. This has two advantages. The tape does not have to be rewound, and the definitions, which would be completed at the end of the first pass, could now be read in front of all the rest of the partially compiled program. The compiler has to remember even less than with the crude and simple two pass approach and so even larger programs can be compiled.

I followed this same approach with the Micro-16 and other Algol60 compilers that we were commissioned to produce. However, I added an extra, simple feature. The intermediate information would be written into a main store area until it was filled up. Only then would it be sent out to paper tape. This meant that programs could be compiled in one pass if they were small enough. If they were too large for one pass compilation, the compiler would detect this and automatically move into two pass mode. Our compiler could compile small programs in one pass on the 32 kilobyte Digico Micro-16 and respectable sized ones in two passes.

Our other two Algol60 projects were for Honeywell and ICL Dataskill. Honeywell manufactured their own range of computers and Dataskill was a software house wholly owned by the computer manufacturer ICL. Many of Dataskill's contracts were for writing software for their parent company, ICL. For the first of these projects, RADICS had to produce a back end for a Honeywell machine. For the later stages of development and testing we needed access to the Honeywell and ICL machines. So four of us, Mary Lee, Clive Jenkins, Pat Whalley and I had to commute to Honeywell in Hemel Hempstead, some thirty miles outside London. For the ICL Dataskill contract, the machine was the Cambridge Titan, similar in design to the Atlas, for which a back end had already been produced. The commute to Hemel Hempstead was tedious, given that we had to travel to our starting-point first, which was Highgate underground station. But the offices at Honeywell were pleasant and modern. We all sat in an open plan office and had easy access to the computer. We mainly needed this for the back end work, but the Micro-16 at RADICS was beginning to be in demand for other projects, which were approaching their final stages. The style at Honeywell was a little different from anything most of us had been used to. Managers were held in high regard and used to act accordingly. One of them once asked one of us to do some menial task for him, without realising that we were visiting consultants and not one of his own subordinates. I remember a secretary telling Clive: "You mustn't hang your coat there; that's my boss's coat stand!" But by and large we carried on without difficulty.

To carry out the same task for Dataskill was much less of a chore. The customer set up a telephone link from the Institute's premises in Gordon Square to the Cambridge Titan, and we could access it directly using a Flexowriter and the equivalent of a modem. Programs still had to be run as batch jobs just as on Atlas, but the turn round time was far more rapid. I found I could edit, recompile and run a program in half an hour, often getting eight or nine revisions done in a day, instead of just one or two, which was the limit on the London Atlas. This felt like a breakthrough in productivity. Progress was rapid and, because we were producing the same front end for the two projects, the work for both the Honeywell and ICL contracts benefited. This was a very early experience of using a computer by means of a remote terminal, which, furthermore, in this case, was 48 miles distant. It was to be some twenty years later before I used such a system again, when working at Praxis in 1987.

Algol60 compilers were complex pieces of software and several books and many papers had been written on techniques for writing them. Sample test programs had been published which would stretch the capabilities of compilers and help to distinguish those that behaved correctly and those that did not. I read many of these texts and thought long and hard about the details of solutions. Our premises were a mile or so from the embankment by the river Thames. I found that walking down to the river and letting my eyes rest on the water-borne traffic was an aid to thought, and spent extended lunchtimes working out solutions to some of the problems on these perambulations.

Then came a disaster. The other early contracts that Radics had won were completed and no new business had come our way. The commercial climate at that time had taken a dive and small computer firms started going into liquidation at an ever increasing rate. The periodical "Computer Weekly" published a lengthening list of the latest casualties every week. At one point it was actually easier for less experienced staff to find work and people began to conceal their qualifications when applying for jobs. In a time of commercial depression organisations cut back on using services, especially the relatively sophisticated services of computer consultancies. Radics had to look for a buyer to survive. Negotiations were well under way with SDL, Systems Designers Limited. They interviewed all the staff and gave us presentations about themselves and their company. In the end SDL decided not to buy Radics after all, but offered individual jobs to six of the personnel they liked the look of most and small redundancy packages to the rest. This caused some resentment, as one might imagine. Radics the company had to go rapidly into liquidation, and the six, of whom I was one, considered whether to accept their offers.

I was a little torn. My wife and I had recently bought a house, our second child had just been born, we were making ends meet on one salary and our personal finances were very tight. But none of the members of my Algol60 team had been offered a job by SDL and the two contracts for compilers were not complete. They needed about another six weeks' work and the other members of the team did not feel confident about completing them without me. So I spoke to SDL and declined their offer, asking if I could have the redundancy package, which was equivalent to about one month's pay, instead. They agreed. I then set about trying to negotiate new contracts between both Honeywell and Dataskill and us as a small group of individuals.

The next day we travelled to Honeywell in Hemel Hempstead as usual, although, with Radics terminated as a company, we would not be paid. Once arrived we had a short discussion between ourselves to agree our position, and then I went to see the Honeywell manager. This negotiation did not go well. To my surprise, Honeywell did not seem to be all that interested in preserving the results of the work. I would have thought that having invested in the project so far, a small extra sum to see the finished product would have been worth their while. But Honeywell calculated solely on the basis of their own budget that they had set aside for the

original contract. The amount they offered us would have effectively reduced us to less than half pay for the remaining few weeks of the work. I returned to my colleagues and told them the news, recommending that we reject Honeywell's terms. There was still the contract with Dataskill to renegotiate, which if successful would still leave us with the satisfaction of having produced a final version of the compiler, completed the work and delivered the product. Also, we were all eminently employable despite the temporary minor recession in the computer industry. My team were disappointed and a little dejected. There was some considerable feeling of wanting to complete the Honeywell compiler despite the miserable conditions, but after further reflection all agreed that we should terminate. I went to the manager once again to relay the news. "All right then" he said, almost with a smile. He did not show any concern that having invested in the project, they were left with nothing. I wondered just how much they really wanted the compiler. There could easily have been some internal difference of opinion about the value of equipping their machines with Algol60. If so, it had been concealed from me. I returned to my colleagues and we straight away packed up all our possessions and left. There seemed a heavy finality about this action. Lunchtime was not yet even upon us.

The next day I visited John Chilvers, the technical manager at Dataskill who had let Radics the contract. This meeting was a great contrast to that with Honeywell. John Chilvers was enthusiastic from the start and determined to find a way to complete the work. We worked out the financial package first and then talked about the logistics. Radics' premises in Drayton House were no longer available, since the contract to rent them had terminated along with Radics. John had contacts at Imperial College in London and after a few days we were able to use a basement room in a building that belonged to Imperial in Exhibition Road. A telephone line was sorted out and a link to the Cambridge Titan machine installed. Once again we were in business, this time as a group of individuals. The front end of the compiler was virtually complete; there was just some testing to be done, some last work on the back end, and testing the integration between the front and back ends. Pat Whalley, who had worked with me on the front end, had found herself another job, and left soon after Radics' liquidation. In a quiet and unassuming way, she had done splendid, sterling work on the compiler, often in spite of my own lack of lucidity in explaining some of the difficult technicalities to her. After SDL did not buy Radics out, Radics' management persuaded Pat to take SDL to the Industrial Tribunal, something of a David and Goliath situation, especially as Pat was almost the most junior member of Radics. The result was that SDL was deemed not to have done anything illegal, but we were told that the tribunal gave them some strongly worded advice.

The remainder of us progressed with completing and testing the compiler in that little white-painted basement room. It was a short walk from the Science Museum in Exhibition Road and I spent several lunchtimes visiting there, seeing again the mechanical and electronic

exhibits. There were no entrance charges to national museums then, a freedom that has since come, gone and come back again more than once. I soon finished the front end work, and oversaw the final testing and integration of the complete compiler. After some ten days there was only the final work to be done on the back end and I reckoned I was no longer needed. With their consent, I left the others to it, continuing for another ten days or so without a project leader. My absence enabled the limited budget to fund the pay of the others a bit more equitably. They subsequently delivered the compiler in working order.

Many suites of test programs were available in Algol60. John Chilvers presented some of these to us, along with the results that a compiler on another ICL machine had produced. We could run the tests on our compiler and simply compare results. I felt some satisfaction that there was only one discrepancy, and it was the ICL compiler that behaved incorrectly.

In writing the Algol60 compiler, there was one thing I would now have to do differently. There are two kinds of division operator in the language, written / and ÷. The / operator always produces a result of type real, such as 5.32 or 1.0. The ÷ operator produces an integer result, that is a whole number like 5 or 1. If the context expected an integer, I allowed the / operator to deliver an integer result. This would allow some programs to be compiled successfully with an expected meaning, whereas the Algol60 language rules would reject the program as faulty. Thus our compiler was very slightly more lenient, as it were, than one which was strictly according to the book. I felt we were giving the customer more value for their money this way. But, in due course of time, such a policy would have been regarded as erroneous. A good compiler should accept and compile correctly exactly those programs that are allowed by the language definition. It should reject programs that the definition does not allow, even if the intention of the program is obvious. The reason for this is standardisation, universality and portability. Someone who wrote a program that made use of this extra feature that I provided would not have been able to compile their program on another accurate compiler. But recognising the importance of standards was in its early years, although growing.

Algol60 was by this time, over ten years old. A few years before, in 1968, a new language, a successor to Algol60 called Algol68, had been devised, again by a committee. This had many interesting features and treated a greater variety of programming concepts as manipulable data, for example. But it never really gained popularity in the same way. It had gone up a technical cul-de-sac. Many new programming languages were being devised at that time, but the ones which took off and persisted for the next decade were more notable for their simplicity rather than their advanced intellectual features.

But, it was 1979, I was out of a job and had a family and a mortgage to support.

Chapter 4 Running through Treacle

Just as I gathered the Sunday newspapers together, where most of the professional jobs were advertised, there was a postal strike, which lasted for some weeks. I could not reply to advertisements by post. However, the strike gave me a perfect excuse for telephoning instead, and receiving perhaps a faster response. I actually set out on one or two days and cold called a few firms in person, and was received because of the postal strike. In other circumstances I would have been sent away and told to apply in writing. I telephoned Univac, Burroughs and ICL, all computer manufacturers. Univac were situated by Euston station in central London, in a tall office block covered in tinted glass. It had long been a noticeable landmark on my daily commute to work, both to Radics and to ULACS. Burroughs had offices on the Thames embankment. ICL were in Bracknell, a new town forty miles to the west of London. All three agreed to give me an interview.

When I arrived at Univac, directed to an office on an upper floor, I found that my interviewer was another mathematician I had known at Cambridge University, John Marsden, a year ahead of me at Trinity college. We had a relaxed conversation, and I was left with an optimistic assurance that the personnel department would be in touch with me.

Next I went to Burroughs' offices on the embankment: another tall modern building, with a sunny outlook over the river. The concierge directed me to an office on an upper floor once again. In the lift I encountered another man, who greeted me heartily: "Are you a Burroughs man?" My heart sank a little. This began to seem like a firm with an over-strong sense of corporate loyalty. No, I replied, not yet at any rate, I was here for an interview. My fellow traveller in the lift wished me well and left. I continued upwards and met my interviewer, a fairly young man with a slight northern accent. He was enthusiastic from the start, not so much interviewing me as trying to persuade me to accept the job offer, which he well nigh took for granted. He quickly described the company, which was American owned, something he frankly described as a disadvantage, without going into details. Burroughs were designing a new machine, a minicomputer, and the post they wanted to fill was manager of the systems software team. This team would operate from the factory where the machines were to be manufactured. The factory was in Cumbernauld, in Scotland, a new town to the north of Edinburgh and Glasgow. I was rather taken aback, because no mention had been made of this in the job advert. My job would be to lead the team producing the systems software for the new machine. This sounded a very exciting and challenging task. Their conception of the systems software sounded very pared down: an operating system and just two compilers, for Fortran and Cobol. Still, I pointed out, the OS would have to include items like an editor, because programmers would need to write and produce their programs, loaders to load code in and out of store, and device drivers for whatever devices they proposed to attach to the machine. The next shock was that they were planning on a team of just six people to do all this. I was rather amazed: even for a small machine, I was thinking in terms of twenty to

thirty or more. I said six people did not seem to be enough to me, and he asked me, well, we would welcome your advice, how many do you think would be required? I thought rapidly. I began to see that my interviewer probably did not know much about software production. But if I said thirty, he might be put right off, thinking that it was I who was being unrealistic and extravagant. Maybe it could just be done with twelve: two on the front ends of each compiler, two on the shared back end, two working on the core of the operating system, which would have to manage the interrupt system, two people working on device drivers, one more for utilities and myself managing and coordinating the whole thing and helping out in the individual programming tasks where necessary. So I said it might be done with twelve people. I was expecting him to look shocked at my doubling his estimate, but he looked unperturbed. Well, we'd certainly listen to your advice on the matter, he said.

This first interview was fairly brief. My interviewer said I should come to the factory in Cumbernauld and meet the managers there, and I should bring my family with me. He appreciated that it would be an upheaval moving up to that part of Scotland. It was important that we should see the environment and get some idea of what would be involved in moving there. We should all spend the weekend in Cumbernauld, hire a car and look around. The company would pay.

This was a completely different proposition. The only addresses for Burroughs mentioned in the advertisement were in London, in particular the one on the Thames embankment, one of the pleasanter parts of the city. A new town in Scotland would mean a change in lifestyle. I went home and talked to my wife Hazel about it. We thought about the change, being 450 miles away from our friends in London and probably losing touch with many of them, and all the other implications. But, although there were more jobs available in and around London, those in computing and engineering in general were spread over the whole country, precisely because many were associated with manufacturing facilities, which were deliberately placed in areas of low employment and cheaper land for building. We accepted the invitation for all of us to go to Cumbernauld for my interview and the weekend following.

Burroughs was perhaps my first encounter with big-company largesse. I had only flown a few times before, once to a job interview at CERN in Geneva, and the other couple of times on short holidays in Paris. A cut-price flight left from Lympne on the south coast of England and landed in Beauvais on the north coast of France. The rest of the journeys were done by coach, so the flights were alternatives to ferry crossings across the Channel. This time we left from Heathrow, flew the 400 miles to Glasgow and were accommodated in a hotel in the centre of Cumbernauld. The town centre was laid out with all the facilities, shops, pubs and so on, in a three-dimensional concrete construction with several levels. There was a green area surrounding this, laced with footpaths, and surrounding that the residential area. All the houses were of a uniform pale grey pebbledash, with small rectangular windows giving the appearance of slots. My two children were under four years old and we all slept in the same

hotel room. In the morning I was collected to go to my interview. I left my family to explore the town.

I had put on a suit and wore my best shoes for this interview. We arrived at the works and walked through the shop floor on the way to the offices. This interior was more like a heavy engineering workshop than an electronics manufacturing plant. The floor was screed concrete and the area peopled by men in blue overalls. I felt self-conscious in my suit. The blue-overalled men seemed to look at my shoes especially, as we walked across the concrete floor. We arrived in a conference room and met several other managers, including Ed Henderson, who would be my immediate manager if I accepted the job offer. The other two were American. The conference room was decorated in hideous taste, with dark orange and green vertical striped walls. They told me that the room had just been decorated and they were very proud of it. We talked again of the estimate for the number of staff required to produce the systems software for the new machine. The most vociferous American was a bit more cautious about my proposal that a minimum of twelve would be necessary, but he didn't rule it out. He left the proceedings quite soon, saying – Well I sure hope you come on board, Brian. That is my first name but everybody calls me by my second name, Tim. This man had read my application form and gone straight into first name terms without the usual preliminary negotiation. Today, indeed for a long time now, this has been the way, but in the UK in the 1970s, we were all a little more formal. His attempt at familiarity annoyed me slightly.

My escort, who had originally interviewed me at the Burroughs premises on the Thames embankment, had proposed that he showed us all the rural sights over the weekend. I was impressed by this offer, but now he seemed less enthusiastic. He mentioned a girl friend, and I realised that he would, understandably, rather spend his time with her. I assured him that I could hire a car and we would drive around ourselves, especially if he could give us some indicators as to where to go. This made him very happy! So I returned to our hotel.

Hazel had spent the day with our children in Cumbernauld. She told me how she had talked to a lot of residents there, and how all of them seemed to have said words to the effect: Oh!, you don't want to live here!. I thought maybe towns need to grow of their own accord rather than be planned and planted in the middle of somewhere where it would be useful to have a habitation. We spent the weekend driving around the beautiful Trossachs, an area where now, by strange circular circumstance, I have chosen to live. But then we were townies and the rural charms did not impress us so much.

A few days later I took the train to Bracknell and made my way to ICL's offices in Lily Hill House. I realised I had been there before. I had had an interview there on the "milk round" series of interviews in my last year as an undergraduate. On that occasion I had missed the stop on the train, not realising how close the stations were together and how short a time the

train paused at each one. That first interview had been a bit of a disaster; the person who was supposed to interview me was not available and no-one else knew that I was arriving. I was not offered a job, but I was not too concerned. I had plenty of other offers to choose from. This time, nine years later in 1971, I was seen by a senior manager, Mr. Pearson. ICL were embarking on the design of a new computer, the "new range", later to be named the 2900 series. They were also having a modern new building constructed, and all the programming teams would be moving into it when completed.

Mr. Pearson described the various teams and projects associated with the production of the systems software for the new range. There would be about twelve hundred people involved altogether. I remarked to Mr. Pearson that Burroughs were also embarking on the production of a new machine and were proposing a team of six people to achieve essentially the same task. Pearson calmly remarked that with a paired down group of that size, they might well be successful. As well as the various software construction teams, the ICL structure had a number of "technology centres", which carried out a coordinating and advisory role. With my experience and enthusiasm for compilers, the Language Systems Technology Centre seemed a good choice for me. It turned out that this group was led by John Buckle, another Trinity mathematician and contemporary of mine. I was offered a job there and then and told that I would receive a formal offer in writing as soon as the postal strike permitted.

I still had not received any communication from Univac. I was out of work, but had received two good job offers. Burroughs were pressing me for a reply to theirs, saying that if I was going to reject it, they would like to know as soon as possible so that they could interview other candidates. On the principle of striking while the iron is hot, I accepted the job with ICL.

I still wonder whether I should have pursued the possible opportunity with Univac. I took their lack of a reply as a lack of interest, but my papers could easily have been sitting in some administrator's 'pending' tray. I have since had much more experience of large corporations and have learned that their large administrations, especially personnel departments, can be detached from the units that do the real work. They can sometimes seem more of a hindrance than a support. If I had telephoned my colleague from Cambridge who had interviewed me, who seemed quite keen to employ me, things might have turned out differently. We would not have moved house; we would probably have stayed in south London and had a completely different set of friends, our children going to different schools later on. Lives close to me would have been different. I wonder this, especially since I was not happy at ICL and we moved back to London just a year later. But I did not have this foreknowledge at the time.

ICL had many offices, most of them in the Reading area. I was to work in Bracknell, at first in Lily Hill House. This building was a rather rambling early twentieth century country house

set in grounds densely planted with trees, giving a faintly claustrophobic effect. Much of this part of Berkshire had this feeling of being crowded by trees. One can rarely see a distant horizon. But ICL were fitting out a brand new purpose-built building at the other end of Bracknell, and we were to transfer to that when it was ready. The move was welcomed by everybody in the Language Systems Technology Centre, called LSCT for short. Our offices in Lily Hill House were cramped, old and poorly decorated. John Buckle, whom I knew from former times, was in charge of the group, but he was soon to move on and his number two, Peter Dove, would take over as leader. I spoke to John about my joining the LSTC and discovered to my surprise that he had not seen my job application, CV or any details. So I gave him a rapid run down on my career to date. At ULACS it would have been unheard of for even a junior manager to have staff assigned to them without complete consultation. I was slightly shocked that I had been assigned to John's group without his knowing more about my experience, but I was to learn continuously over the next twenty years how different organisations behave very differently from each other in these manners of people and organisational relations.

There were many highly skilled and gifted people working at ICL. But their reputations remained mostly confined to ICL: they produced relatively few publications and did not display their technical work very much in the forums of the professional institutions like the British Computer Society or the Association of Computing Machinery. With my experience of compilers, I was given the task of trying to unify the design of the different language compilers across the New Range machine.

For all the languages that ICL regularly provided with their computers, Cobol, Fortran, Algol60 etc., there was a separate team of programmers developing a compiler. I found a great deal of resistance among the teams to any change. They had developed their designs over the years for previous machines. The compilers worked and I can understand the reluctance to introduce even unifying changes. Transporting a design of a program to work on another machine was relatively straightforward. There were always claimed to be special reasons unique to each language why the design had to be as it was. The opportunities for unifying the designs were great. I had visions of a common intermediate language into which the front ends of every compiler could translate the source code, and then a single common code generator for the new range machine, a standard parser generator system for all the languages with the syntax expressed in a common language based on BNF, and more. These unifying approaches to design would have presented the users of the compilers, that is those programming in the various user languages, a similar feel and response to the compilers. But I soon began to realise that these were vain hopes. And the compiler teams indeed had some justification along the lines of “if it ain’t broke, don’t fix it”. Their compilers had worked well over several generations of ICL computers.

I decided to take it all a small step at a time. I thought that if the LSTC could build a parser generator and demonstrate it, this may be a way forward. A first step in building one is to produce a macro generator, a program that substitutes short pieces of computer text with longer sequences, possibly with parameters, like a form letter only more elaborate. Such a facility is also of general use in building compilers and other items of systems software. The language used for developing the systems programs for the New Range had been decided. It was "S3", a fairly simple subset of Algol68 devised by ICL. So it would be natural to program my proposed macro generator in S3. I put forward the proposal to do this as a first step at several meetings and it was agreed to go ahead.

I produced a design for the macro generator and it was implemented by a young programmer in the LSTC team. Meanwhile I had other regular duties: reviewing documents from other departments, helping to outline principles of development policy, and so forth. I began to find most of this work intensely frustrating. There seemed to be a great weight of inept and stubborn opinion to overcome in order to get anything done, despite the fact that there were many extremely capable people in the company.. People have likened ICL to a section of the UK scientific civil service, and I can see why. There was a vast amount of internal discussion and debate, with people taking stances and striking poses.

ICL was the result of several mergers, the most recent between ICT, with whom I had worked on a vacation job while at university, and English Electric Leo Marconi. Different divisions still retained some inheritance of the company cultures from which they were descended, and some isolating barriers remained. I felt I was not achieving very much and began to look elsewhere once more.

Chapter 5 The Country Club

In the Sunday Times an advertisement appeared asking for a leader of a "basic software control centre". I assumed that "basic" software meant systems software, operating systems, compilers and so on, and tentatively replied. I was invited to attend an interview at STL, Standard Telecommunications Laboratories in Harlow, a new town in Essex to the north-east of London. The letter, from the personnel officer, Martin Jenner, said I was to meet "Mr. Flowers from Antwerp". I was a little puzzled, because the job advertisement made no mention of any connection with Antwerp. However, I drove with my family to Harlow, partly to let my wife and family have a look at the town and surrounding area, and arrived at the laboratories. I met security guards in the entrance foyer and checked myself in. I asked them if they had any idea how long I would be required. "Oh, you will be hours, hours!" they said. I returned to my wife waiting in the car and suggested she returned in an hour and a half.

I was soon summoned and walked along a ground floor corridor to the personnel department. A younger man with fair hair introduced himself as Martin Jenner, and the older of the two, a

friendly bear of a man from Florida, shook hands with me “I’m Lou Flowers” he said. I had a lot of questions to ask, and learned that STL was a telecommunications research laboratory belonging to STC, Standard Telephones and Cables, a well known British firm that supplied much equipment to British Telecom, which then was part of the Post Office. What I did not know was that STC was part of an American multinational conglomerate called ITT, International Telephones and Telecommunications. STL was one of three research laboratories in ITT, the others being in Versailles to the south of Paris, and the third in Madrid. STL did much research into materials and physical devices, but not a lot in the area of computers and software.

In the earlier days of telephony, exchanges were manual, operated by many operators who would answer when you picked up your phone and manually route your call through to its destination, relaying instructions along the line to other operators if necessary. I remember the first telephone in my parents house. It had no dial, just a handle to turn which generated a calling signal to the operator. The mouthpiece was mounted on the wall too high for me to reach and speak into as a seven year old. The first automated exchanges were operated by pulses generated by dials which caused relays to switch the call to its destination. The next generation were electronic, replacing the relays by electronic switching circuits. In 1972 these electronic switching circuits had begun to be replaced by computers and software. ITT had developed two computers for this purpose, the 1600 and the more powerful 3200. The 3200 was a development of a previous STC machine, the Stantec Zebra.

Teams of programmers in many ITT companies developed software for telephone exchanges, and also for telex “store and forward” exchanges. Younger or future readers may not know what a telex is. Almost from the beginning of telephony, it has been possible to send typed messages across the public telephone network. Telegrams would be dictated to an operator who typed them out on a teleprinter in the exchange. The message would be printed on a strip in the destination exchange, glued to a sheet of paper and carried in haste by hand to the recipient, usually by a telegraph boy on a bicycle. Telexes had begun to supersede telegrams. Any firm could have a teleprinter and a telex line, just as they could have a telephone line and handset. The message could be typed out onto paper tape, the number of the receiving telex machine dialled, and the text sent via a paper tape reader. Telexes were a cheap way of sending messages internationally and had the advantage that you did not have to wait for a mutually convenient time of day. If the message arrived in the middle of the night, it would not disturb the recipient; it would be waiting ready to be collected in the morning. Telex exchanges would store these messages electronically on magnetic tapes and forward them to their destination when a route was free. These telephone and telex exchanges were often called “switches”. Fifteen years later, telexes would be overtaken by email.

In this first brief interview, I asked many questions about the software development technology that was being used. In fact, I think Lou Flowers probably learned all he needed

to know about me from the questions I asked. I began to realise that this part of the telecommunications industry was way behind the sectors I had worked in before in their software development techniques. The applications software, that which drove the telephone and telex switches, was mostly written in assembly code rather than in a high level language. The 3200 and 1600 machines were small minicomputers designed to be embedded in telephone and telex exchanges, so the best way at the time to develop software for them would have been to use a standard workhorse, a mainframe like the IBM 360, with specific compilers producing code for the target machines. A mainframe machine would bring many advantages like editors and the ability to produce test programs, analysers and so on. But the company's attitude was against buying "unnecessary" computers. There was always a target machine awaiting delivery to a customer, so at first the development teams had to use that target machine for developing the software. This policy had moved on a little bit. At STC in Cockfosters and STL there were computer centres containing a 3200 machine, as there were in other ITT production factories, some fourteen altogether situated around the world from Des Plaines in Illinois, USA to Sydney in Australia, but most of them in Europe. The mission of the 3200 Basic Software Control Centre was to keep these fourteen computer centres supplied with a simple operating system, test programs and various other facilities including an assembler for the 3200 assembler language. But the machine at the centre of these computer centres was still a typical target machine, that is a machine of the type that would be found in the centre of a telephone exchange.

The 3200 Basic Software Control Centre, or BSCC, was being moved from STC in Cockfosters to STL in Harlow. Cockfosters was on the north-east edge of London, and its underground station is noted for being the end of the Piccadilly line. Harlow was some thirty miles further out of London and not that easy to reach by public transport. Part of the team would be transferring to Harlow, others would stay in Cockfosters and move to other departments there, including its current manager. A few new people would be recruited to replace those that remained behind. STL would become the BSCC's new home and, if I got the job, I would be its new manager. I realised that this would pose interesting and challenging problems. I would have to overcome possible resentment that the promotion was not from within; the current team members would be wary about the character of their new boss; it was clear that the host organisation, STL, knew little of the BSCC's operation, providing only administrative support. I would be reporting directly to Lou Flowers, who worked in an ITT company in Antwerp.

After a surprisingly short forty minutes I was refunded my travelling expenses and discharged through the entrance hall. So much for the security guards telling me I would be hours. I had another three-quarters of an hour to kick my heels waiting for my wife and family to return. Mobile phones were twenty five years away in the future, so I could not get in touch with them. I was, however, to have two more interviews.

Lou Flowers worked in a part of the ITT organisation called the Computer Engineering Centre in Antwerp. His manager, the director of the Centre, was Gerry Jacob. My next interview was to be with him, in central London at the hotel where he was staying, the Cavendish in Jermyn Street, over dinner in the evening. I arrived at the appointed time and we went straight into the dining room. The Cavendish was one of the higher rank of fashionable London hotels. We were shown to a table and, as I surveyed the row of six waiters literally waiting on our bidding, I saw that we were the only diners in the room. I rapidly realised that Gerry liked the good things in life, was, indeed, a man who valued "good taste". After we had chosen our dishes, Gerry accepted the wine list. He said, "well, as you're eating fish and I'm eating meat, we'd better have half a bottle of red and half a bottle of white. Would you like to choose some white wine for yourself?" and handed me the wine list. I looked through the long list of white wines and felt that my choice was perhaps going to be the first test in my interview. I decided not to go for anything at the high end of the range, which would obviously be greedy, but also not to go for the cheapest ones either, for that would display nervousness, lack of an ability to fight my own corner and possibly an unsophisticated palate. I chose something about a third of the way up. I believe it was a Pouilly Fuissé.

I must say that this meal was not easy to enjoy. We were the only diners in the restaurant throughout. The line of waiters on the other side of the room unnerved me, watching us with some disdain, or so I imagined. Unlike Lou Flowers, Gerry asked me many questions, most of them about what I would do in various hypothetical management situations. How I would respond to unjustified complaints from customers, how I would handle various examples of inter-group rivalry, even how I would react to overbearing interference from my manager while I was away from my office. Gerry's questions came quite frequently, several of them while I had a forkful of food on its way to my mouth. At one point he had to give me time to catch up with eating my dinner. However, I believed I answered his questions well enough. They also indicated that the job was going to require quite a bit of diplomacy and tact, and that there were a lot of conflicts and pressures to deal with.

I received a telephone call from Lou Flowers at home in the evening. I had "passed" the second interview, and there was to be a third one with Mr. Don Combelic. "I want you to remember this name carefully" said Lou. "Don Combelic has a lot of influence". A date was arranged for us to meet over a meal once again, at the Excelsior hotel close to Heathrow airport. I was a little puzzled as to why I was being dragged along to a third interview. My wife, Hazel, reckoned that Lou Flowers and Gerry Jacob had decided they wanted to give me the job, and that they now had to convince this third man, who had some controlling say in the matter. This indeed turned out to be the case.

I walked into the entrance foyer of the Excelsior hotel. Lou Flowers was there, slightly to my surprise. I was expecting just to meet Don Combelic. "I felt I should come along too" said

Lou. “Mr. Combelic should be with us in a few minutes”. We chatted for a short while and then a tall grey haired, rather gaunt man came in. “Well, good evening gentlemen”. Don spoke in a relaxed, gravelly drawl. Lou introduced us and we made our way to the dining room. Two waiters laconically sauntered to our table pushing a trolley bearing a hinged covered dish. “Would you like the roast of this evening gentlemen? It is a honey-roasted ham”. He rolled back the silver cover and revealed a very large, steaming joint of ham glazed and studded with cloves. A section had already been carved off it and the flesh beneath was attractive and succulent. “Wow, that looks pretty good” said Don and we all agreed to have it.

This meal cum interview was in some ways a little less uncomfortable than the one I had with Gerry Jacob, and in some ways more. Gerry asked me more searching questions, but I felt much more strongly that I was on the same wavelength as him. Don’s questions were more oblique and he was more conversational, but I felt less sure of his priorities. He seemed to want to know if I could be “tough” if the circumstances demanded it. “Why did it take so long to fire him?” he asked when I recounted some previous event. I explained a bit about industrial relations legislation in Britain and he took it that my hands were tied by the rules. Afterwards I gave him and Lou a lift in my car back to the airport and then drove Lou to the nearest underground station, which then was Hounslow West. Lou was visiting STC in Cockfosters the next day, but I realised with some amazement that Don Combelic had flown from Paris to London for the sole purpose of interviewing me. With just the two of us in the car now Lou Flowers became much more forthcoming. “I thought about it and decided there was no way I was going to let Don interview you on his own” he said. Don was in what Lou called a “staff” position, and worked in the ITT laboratories in Versailles, on the outskirts of Paris. “I think he’s agreed that we can take you on” he said. It occurred to me that in all three interviews I had not mentioned that people seemed to like working for me, so I said words to that effect, adding “I’m not quite sure why”. “Well I think I know why” said Lou, and began to eulogise my character and generally extol what he thought were my good points. I was surprised and slightly embarrassed. “How does he know?” I thought. We parted and a few days later I received a letter from the personnel department at STL formally offering me the job at a salary of £4,250. I felt pleased with this offer. It was twice what I was earning some five years earlier. I accepted, although I was a bit apprehensive, since the job at ICL had turned out within a year not to be to my liking. Could the same thing happen again? We had worked hard establishing ourselves socially in Bracknell while I was at ICL, setting out to make friends and join local groups. Suppose I found that the job in Harlow was likewise not what I wanted? So we decided it would be safer to move to north London, near a train route so that I could commute against the flow by rail to Harlow. That way if I moved jobs again at least I would have all the London opportunities to choose from without moving house yet again. We also had friends in London already, from the time before I worked at ICL. We

weren't to know that I would stay at STL for the next thirteen years, the longest I was ever to work anywhere.

Because so many, widespread, companies comprised ITT, they made great use of telexes to communicate between each other. Indeed, a couple of weeks before I started at my new job, Gerry Jacob sent a few telexes to me at ICL. I was a bit alarmed, and hurried down to ICL's telex room to collect these messages from the operator before too many people saw them. I did not want my managers to think I was starting work on my new job while still on ICL's payroll and premises. Few people at ICL ever used telexes and the operator began to treat me with great respect, thinking that I must be very important to receive all these messages! Gerry's first message said that unfortunately Lou Flowers would be on holiday for the first two weeks after my arrival, and that he was afraid they had arranged a heavy travel schedule for me. I was to go to STL for the first two days, where I would make the acquaintance of Hunter Mitchell who headed the documentation group. He would arrange travel tickets for me and I should visit Gerry at the Computer Engineering Centre in Antwerp on the Wednesday, continue on to the LCT laboratories in Versailles to visit Don Combelic on the Thursday, and finally visit the ITT company STR in Zurich on the Friday, all in my first week. I realised that my passport needed renewing! Fortunately I could get a visitor's passport over the counter at a post office as a temporary measure.

My first two days at STL were interesting. A large research laboratory, set in the countryside just outside Harlow, STL had a sports and social club on the premises, including tennis courts, and employed about a thousand people. Its immediate surroundings were agricultural land. The staff at ITT headquarters in New York gave STL the nickname of "The Country Club". Hunter Mitchell explained a little of the arrangement that had been agreed between STL and the Computer Engineering Centre in Antwerp. STL were to provide my group with offices, administrative support, the services of the personnel and accounts departments, in other words all the infrastructure required to run an office within a big organisation. But I would report to the Computer Engineering Centre. So my group, the BSCC, were a bit like lodgers at STL, and they would be our hosts. Hunter himself ran a group of technical writers which likewise reported to the CEC, under the same kind of arrangement. Gerry Jacob was his manager too.

STL had a travel department which used the services of a commercial travel agent. About a year after I had been working there, both firms realised that this travel agent had a single person dedicated to all of STL's needs, and so they made the very sensible decision to place him actually in STL while still keeping him on the travel agent's payroll. This reduced telephone calls and speeded up obtaining air tickets and so on. The agent was also more than happy to arrange holiday travel for employees. Don Combelic's trip from Paris to London just to interview me, and my schedule of three destinations within my first week were typical of the casual attitude in ITT to air travel. As soon as anyone at STL went on a trip, they

would be given an expense account. Hunter guided me through the procedures for filling in claims and gave me some useful advice about the conduct expected while on company business. Staying in three or four-star hotels and eating in decent restaurants was the order of the day. There were some specific rules about supporting one's claims with receipts. The Computer Engineering Centre itself was a small office in Antwerp, but similarly hosted by a Belgian ITT company called the Bell Telephone Manufacturing Company. I was to discover that Gerry and his staff had to contend with a much fiercer bureaucracy within this company than Hunter and I did in STL.

I spoke to the STL site manager and he showed me the offices earmarked for my group, who were still at that point working at STC in Cockfosters. There were a number of rooms, equipped with new desks and chairs, and a couple of cabinets designed to hold large engineering or architectural drawings. STL did not have very much idea of what computer programming really was, except that it was some kind of engineering activity. The site manager and his assistant said that they were not sure what we would require. I explained that we did not need the drawing cabinets, but that we would need telephones in each office with outside, international lines, and a few filing cabinets. Altogether the admin at STL were extremely helpful at this early stage of setting up the new group.

I flew to Antwerp and stayed in a rather drab hotel. I was to learn that this was "scarcely adequate", but it had been booked for me by the Computer Engineering Centre. The CEC was organisationally part of ITT Europe but was on its own, occupying a floor of an office building on the edge of the red light district in Antwerp. At the beginning of a working day in the CEC one could see from the office windows sleepy sailors emerging from houses of ill-repute and young women continuing to seek custom from the last of the all-night revellers. Our software staff were always bemused by a seedy pornographic cinema across the road from the office, bearing a sign in Flemish-English "Sexy Programmers". ITT Europe itself was in a large tower in Brussels. Antwerp and Brussels are a short train ride apart, but Antwerp is in the Flemish speaking part of Belgium and Brussels in the French part. Gerry explained in more detail the relationship between the BSCC, which was my group, and STL. STL were our hosts, and so we had to conduct ourselves like well behaved guests, obeying their admin rules to the letter, not making any exceptions for ourselves compared to other staff and so on. He also explained a bit about inter-company rivalry between the European companies within ITTE, especially between the French and British companies. There were indeed two rival operating systems for the ITT 3200 machine, one produced by the BSCC currently in Cockfosters and the other by LCT, Laboratoire Centrale de Télécommunications in Versailles. All the French ITT companies used the LCT operating system, all the British, Australian, South African and USA companies used the BSCC system, and companies in other European countries varied but mostly used the BSCC software.

ITT was a company with an engineering tradition, manufacturing telecommunications equipment. But it was also a multinational conglomerate, owning many companies some of which had nothing to do with telecommunications. It owned Sheraton hotels, Maws baby bottles, a company that made bacon slicers, and many more. From time to time the USA anti-monopoly laws required ITT to sell off some of its companies. The grand total of ITT's employees world wide was a staggering eight million, more than the total populations of many significant countries such as Austria, Denmark, Norway, Finland, Israel and New Zealand. ITT's headquarters were in New York, with a European headquarters in Brussels. The managers and staff from headquarters used to spend much of their time in aeroplanes, flying from one ITT company to another, debriefing local managers and handing out decisions on funding. The European headquarters, ITTE, was cynically known as "International Talking, Travelling and Eating". I was entering this very different working world where business relationships were across national boundaries and one had to be sensitive to cultural differences.

The Computer Engineering Centre controlled the standards and practices for the manufacture of computers that formed part of the telecommunications equipment produced by ITT, in other words, the computers that were embedded inside computer controlled telephone exchanges. The company had a strong engineering tradition and had a large set of standards for engineering equipment. Great store was set by engineering quality, and there were standards for manufacture, testing and documentation of all products produced in ITT. Every product had a product number, with a structure to it to cater for different versions and variations. ITT's expertise was in electrical and electronic engineering. There was little understanding of computer software, and it was a challenging change for me to work in a company that had little tradition in my own expertise. The intangible nature of software puzzled my managers. Because it did not use any significant materials, had no weight so to speak, many managers could not understand why it was expensive and time consuming to produce. So they regarded software as being something a bit like the standards documentation for hardware products. Everything was recorded, documented, but did not add anything you could physically measure when loaded into a computer. There was a story of an aircraft manufacturer demanding to know how much extra weight the software would add to the aircraft when loaded into the on-board computer. They thought it might make a difference to the handling and fuel consumption!

Gerry told me something about the fragile relations between the BSCC, which was part of the CEC, and the French laboratory LCT in particular, who had built a rival operating system to the BSCC one, and who therefore had a suspicion of, not to say an antagonism towards, both the CEC and the BSCC. And my next visit the following day was indeed to LCT, where I would meet Don Combelic. Don was employed by ITT headquarters, and was assigned to oversee the software production in ITTE. It was perhaps an odd choice for Don to work in

LCT, which was a slightly maverick member of ITT and did not make life particularly easy for him. He could have easily worked in ITTE itself, in Brussels, which would have been a much more friendly working environment for him. But Don preferred to live and work in Paris because he had grown to like it there, although he complained often and loud about French customs. He had, however, taken on board the French attitude to food, hook, line and sinker. He did not have Gerry's gastronomic discernment, but if anything he was more fastidious. Don had learned to speak French slowly, with an accent that barely acknowledged French pronunciation, but on the other hand with complete and, to me, enviable fluency. The result was that French people would always pay attention to him and appear to respect him. When I met Gerry he was gallantly trying to learn French and was having intensive lessons. For that reason he chose to live in Brussels, where French was spoken, and commuted to Antwerp.

Don had a small enclave of staff working at his direction in LCT, all of them imported from elsewhere. This group was called the ASG, Advanced Software Group, to the slight embarrassment of some of its members. Don introduced me to the leader of the ASG, Bob Parenti, an American, who had initiated a language for the 3200 machine called ESPL1. I had not been told about this language until this moment, so I was slightly surprised. A compiler had been developed for the language and was working and used, although by only a few projects, I was to discover later. On introducing Bob Parenti, Don told me that he was "Mr. ESPL1". This was the first of many Americanisms that were to escape me in my early times in ITT, so he had to explain to me what it meant. I had scarcely met anyone from the USA until then, and there was much less American influence on British television. This language difficulty was two-way at first. I had to curb my use of specifically British English terms, after discovering which, indeed, these were. Some weeks into the job I sent a report to Lou Flowers, my boss, which contained the phrase "can be found overleaf". Lou later told me he had turned to a colleague in Antwerp and asked "What in hell's name is an overleaf?". At ICL people competed to use expressive language in their documents; anthropomorphic metaphors being a favourite. With employees of many nationalities, the stylistic imperative in ITT was clarity and simplicity.

Bob Parenti told me that ESPL1 stood for Electronic Switching PL/1. IBM had developed PL/1, a high level language, in the mid 1960s, and had promoted it energetically. It was originally called NPL, for New Programming Language, but the name was changed to avoid confusion with the National Physical Laboratory in Teddington to the south of London. PL/1 had many advanced facilities, so my ears pricked up when Bob told me of ESPL1's claimed provenance. I asked him if it had some of the more interesting features of PL/1 – recursion? Block structure? Union and Structure data types? Bob shook his head: "Nope", "Nope", "Nope". I stopped my questions to avoid embarrassment. In fact ESPL1 was a fairly simple autocode, a kind of programming language that is a step or two above machine code and

assembler languages, but at the low end of the range of high level languages. Instructions in an autocode begin to look much more like simple mathematical formulae rather than instructions to a computer. ESPL1 was a very considerable advance over the symbolic assembly language used for all the basic software and most of the telephony applications programs, but scarcely bore comparison with PL/1. In writing a few experimental programs of my own, I found it far preferable to use ESPL1 than the 3200 assembly language. Don introduced me to several other people in LCT, including John Devoil, an Englishman who worked on the ESPL1 compiler along with a small group in STC Cockfosters.

At Gerry Jacob's prior request I also met Peter Liou, who worked for Hunter Mitchell but was located at LCT. He was working on several projects producing documents associated with some of the electronic designs that LCT were devising. One of the LCT projects was a new version of the ITT 3200 machine, the 3202, compatible with the present one so that existing programs would still work on it, but using more up to date technology, which would make it faster and cheaper. Peter Liou was producing some of the documentation for this new upgraded 3200, following ITT's in-house documentation standards. I had had some contact with big company document standards already at ICL but ITT's were much more comprehensive and part, so to speak, of the company ethic. The ITT standards manual ran to several volumes and was continually being updated.

So this was why the BSCC was called a "control centre". The software for the 3200 was a standardised ITT product and the BSCC "controlled" it. This meant that we allocated a product number to the separate items of software and ensured that their documentation conformed to ITT standards. The documents themselves also had numbers, with suffixes indicating the version and variant, and for the software itself, a further part of the suffix indicating on what medium, paper tape, magnetic tape and so on, the software was recorded. The structure of these numbers was arranged so that the number for a piece of software and the documents describing it had a common stem, with standard segments indicating that this was the software itself, or the user manual, or the design description, and so on. The BSCC had a range of numbers at our disposal and we allocated numbers not only to our own software but also to software produced by other groups. These included one or two in STC Cockfosters and in LCT itself.

A great bone of contention for LCT was that the BSCC operating system for the 3200 machine was recognised as an ITT product and given numbers, but the system produced by LCT was not. So although LCT's operating system was distributed and used by a number of companies, mostly French ones, for producing telecommunications software, it was not recognised as an ITT product which conformed to standards. And this was not because of any particular lack of technical or even bureaucratic criteria, but on principle, because Don Combelic had instructed the BSCC not to grant it the ITT status of a standardised product. This was the source of great resentment for LCT. Yet Don Combelic worked from an office

within the LCT laboratories, establishing his own clique of programmers there and, to add insult to injury, given them the title of "advanced". It was as if the strategic officer from one side of a battle had pitched his tent with some chosen officers in the middle of the encampment of the other side, furthermore by choice, not by necessity. I never ceased to be amazed by this choice of Don's, but it indicated the impregnable character of the man; he had a persona constructed of granite.

I returned to England with my head brimming with information. I had made notes throughout my trip and prepared to compose a report on my observations. I had found the conversation with Peter Liou confusing and could not work out what his rôle was or why he appeared to be working alone at LCT. It was some time before I learned that he worked for Hunter Mitchell. I felt glad that shortly before I had set off for Antwerp, I heard from my contact at STR in Zurich that it was not convenient for me to visit him. He had told the CEC in advance, but the message had not reached me. I was very unclear about the significance of ESPL1 in the company. ITT was an advanced, complex organisation, which nonetheless used software technology ten years behind the times. ESPL1 was the only glimmer of a high level language in sight in the organisation. By contrast, ICL for its systems programming was using a subset of Algol68, believed to be the most advanced language at the time, whereas ITT was using assembly language. Who actually used ESPL1?

Until the BSCC people at Cockfosters were actually moved, I alternated my time between them and STL. The offices in Cockfosters were a miserable environment to work in. The views from the windows were dominated by the shunting yards of the Piccadilly line terminus and plenty of noise from the trains found their way indoors, the screeching of metal wheels without differentials as they were slowly pushed along curved track interconnections. The building itself was tall, narrow and cramped, with grey metal partition interior walls. If I had been working there I would have welcomed the change to the buildings and environs of STL.

Nonetheless, I still had a considerable management hurdle to overcome. I was a newcomer from outside, taking command of this group that might have had a manager promoted from its own ranks. There was bound to be some initial wariness, if not possible resentment, and I had some reassuring to do. Ten years had passed now since I had graduated, and I had spent those ten years working in software engineering. I was somewhat older than the rest of the group, with one exception, Alan Jones. He, like me, had graduated from Cambridge University ten years before, and we were the same age, within a month or two. He therefore was a potential candidate for the job I had stepped into. I needed to give him some special attention.

Over the next couple of weeks I invited each member of the BSCC to come and talk to me. I had taken over the office of my predecessor, EK, who had been moved sideways. From the

various conversations I had had with Lou Flowers, Gerry Jacob and, especially, Don Combelic, I had learned that they had not been satisfied with him. Indeed, Don referred to EK as having been "fired". So I wondered if any of the BSCC staff thought he had been unfairly dismissed and felt a defensive loyalty to their previous manager. But I found no such concerns. Indeed, when I told Alan that Don had said that EK had been fired, Alan was surprised. "Fired? Oh no! He was promoted", he said. I wondered if the Peter Principle had been at work here. In my chats with the members of the group I encouraged them to tell me what their rôle was and to talk freely about how they felt about the work they were doing and their job in general. Every single one of them complained that EK had been extremely secretive, not letting anybody know what was going on, revealing only the minimum of information to let them do their work. They all would like to know more about the context and organisational situation of their jobs, and all of them felt rather isolated. A few also complained about their salaries.

I thought, this is going to be fairly easy to deal with. All I have to do is to take every complaint they have made about EK and do the opposite, in spades if possible. He was tight-fisted with management information; I will be generous with it. He operated a closed door; my door will be open. I looked at the personnel files of all the staff and found that their salaries were quite haphazard. People with the same experience and doing very similar work were paid remarkably differing amounts. I suspect that EK or his managers had paid the staff individually as little as possible without losing them. I thought there would be an opportunity to put this right once everyone was at STL and working under a new personnel administration.

After I had spoken to everybody, I invited Alan to share my office in Cockfosters. "I'm only going to be here for half the time. I shall be working at STL two or three days a week", I said. "And it will relieve the pressure of space in the open plan office". He was grateful for this, and I thought it might make him feel slightly better just in case he was resentful at not having got my job himself, for in some ways he was the obvious choice for it, especially if it was going to be filled by promotion from within. Furthermore, sharing my office would be temporary. When we were all permanently at STL in a couple of months' time I would have my own room there and the offices for the rest of the group would be more spacious.

So, for these first few weeks I worked half the week at Cockfosters and half the week at STL. There was much to be done at STL. I had to arrange for the facilities for the group, interview and recruit a secretary, and get to know the infrastructure there. STL was quite a big site. It had its own medical department and even its own company fire brigade. The accounts department provided comprehensive support, not just payroll accounting but project accounting too. I had to divide the activities of the BSCC into different projects and set up accounts and budgets for each of them. The staff would record their time on time sheets as spent on the different projects and the accounts department would give me reports of

expenditure against budget. The total would be reported against our total budget for the year. All this had to be set up from scratch in discussions with the accounts department. Part of this was straightforward, as each year one had to make a case for the next year's expenditure under various headings. The "case" would identify and cost a number of activities. So it was sensible to match the projects and budgets for accounting with the activities in the case for that year. This was how the ITT research and development system worked: every year a R&D case had to be made for the following year's work, and the case presented in ITTE headquarters in Brussels. There was some uncertainty about this process. It was in theory possible for a piece of work to be abruptly stopped if the case for it was not accepted one year. In the following years I was to become very involved in the preparation of these all important R&D cases.

During my first two weeks my manager, Lou Flowers, was still on holiday. When I returned to STL after my trip and first visit to Cockfosters to meet the staff, I met John McEwan. He had been recruited and was doing some background reading on the 3200 machine at STL, sitting in one of the BSCC offices. We had several relaxed lunchtime discussions in the sports and social club, which was an amenable place to have a snack and a drink. With Lou Flowers being absent at this early stage, I had not been told of the names of any of my staff at Cockfosters; I had had to find them out for myself. I had heard that one or two new staff had been recruited into the group directly into STL, but I hadn't been told the names of any of these either. I thought maybe John McEwan was one of them. At one point over lunch, he said to me: "Are you my manager?". "Well, I think I must be", I replied. This was correct, but I did not have the matter confirmed until Lou returned from holiday a couple of weeks later. Looking back, I am horrified by the lack of organisation and communication that forced me to work so in the dark, but it all made for interesting times.

I invited all the BSCC staff at Cockfosters to come and view our new offices in STL. I arranged this as early as possible, for I felt that the foreboding prospect of the change of location would seem less daunting for the staff when they saw the new place with their own eyes, especially the far pleasanter environment, the better standard of offices with higher ceilings, fresher paint and more solid building. But I lost two of the staff before this visit and a third afterwards. Each of them told me they had decided to accept an offer of a post in another department at STC Cockfosters. There was something exciting about looking at these empty offices in STL that were going to be our home, and making decisions about placing of desks and so on. I made sure that everyone chose which of the three or four rooms they were going to be in. Then, as soon as they were gone, I arranged for nameplates to be put on the doors of the rooms. This had been done for me when I had arrived at ICL and I was very struck by it at the time. It had made me feel welcome and accepted. So again, a ploy if you like, but I thought it would add one more touch to smooth the path of the change of location and office for these staff. And indeed, when they arrived, there were several exclamations:

“Oh, look; they’ve put our names on the doors!”. I did not let on that I had gone to some lengths to make it happen. I thought that it would do no harm for them to think that the establishment itself had somehow recognised and named them. The BSCC staff had not had any attention given to their periodical technical training, so after a few weeks I arranged for them all to attend Data-Fair, a regular software event held in the UK, where latest developments were given an airing. One significant design method, JSP, was presented there by its author, Michael Jackson. This was to become something of a ground breaking method, but I am not sure how many of the BSCC staff appreciated its elegance at that early stage. But I think it must have had some impact on them.

Lou Flowers flew from Antwerp to visit me soon after he arrived back from holiday. Several more people were lined up for interviews to join the BSCC, most of them having responded to the same advertisement as I had. These trips between Harlow and Antwerp were the beginning of a pattern that was to continue for some years. Once a month Lou would come to see me and once a month I would fly to Antwerp to see him. That way we were in face to face contact every fortnight or so, and we would be on the telephone to each other several times a week. Lou liked to keep track of what was happening in detail and would give me specific instructions, but he left the day to day running of the group to me. I had to write a monthly report for Lou. To do this I asked for reports on their activities from most of my staff and combined them into the required form, for even monthly reports were subject to an ITT standard, with particular headings – Achievements, Problems, Red Flags, and so on. Another group, the 1600 BSCC, also reported to Lou, and he would incorporate our two reports into his own monthly report to Gerry Jacob. Gerry in turn took information from Lou’s report, composed it with reports from his other staff and sent his own monthly report up the hierarchy to his manager – and so it went on up the management pyramid. Since every report had to be completed by the end of the month, they were all always done in great haste, but somewhere along the line some slippage must have occurred. Much later it struck me that ITT top management would in this way make decisions of massive consequence, starting or halting projects, even occasionally closing companies, based on misunderstood and misrepresented accounts of the work being done by the lowliest members of the organisation several months earlier. I am not sure whether this is really true, but it seems to be an inevitable consequence of the process.

Gerry did not actually write his own monthly report. He used to delegate the task to one of his junior managers, often to Lou Flowers or, later, to me. Delegation was something of a watchword in management practice at the time. A manager who could not delegate was by definition a poor manager. Gerry therefore used to delegate as much as possible. He once advised me, with a bit of a twinkle in his eye, “Never do anything yourself!”. So a few years later I would find myself flying to Antwerp for the express purpose of taking all the contributing reports that had arrived on Gerry’s desk, including my own, and composing his

report, which he would then review, ask for alterations, and so on. I have to say, I did wonder if this was a cost-effective way of going about things, but that was part of his management style.

I had learned from Hunter Mitchell the process of filling in and submitting my expense claim to the STL accounts after a trip away. After about my second trip, Lou asked me if I could show him my claim. When I did so, he asked me why I had included receipts for a couple of items. "You only have to attach receipts if the meal costs over £5", he said. "Well", I said, "I thought I would show willing". "No", he said. "That's not the thing to do. Let me tell you a story.

> "There was this group of engineers working on site in Arizona. They were far from their main offices and were out there, isolated and working in a temporary hut with poor air conditioning. With the hot weather, they kept the windows open, but flies used to come in. So they bought a fly-paper and renewed it every month.
>
> In fact, they did not have much to do and spent a lot of the time hanging around doing nothing very much. Each month when the time came for them to write their monthly report, they had to scratch their heads to think of some thing to put in it. One day one of them had a bright idea: 'Let's write a fly-paper report', he said. So they counted the flies on the fly-paper and included a brief item in their monthly report:
>
> > Fly-paper Report
> >
> > This month's total was 119.
>
> They wondered if they would receive some castigation from their management for this piece of mischief, but they heard nothing. So the next month they counted the flies again and included another "Fly-paper Report" in their monthly narrative, and continued.
>
> After about four months they decided that perhaps the joke was wearing a bit thin, and they decided to stop including the fly-paper report. Besides, more real work was coming their way for them to report on, and the weather was getting cooler; the flies were getting fewer and they soon would not need the fly-paper any more.
>
> Very soon after they submitted this latest monthly report they received an urgent telex:
>
> > Where is this month's fly-paper report?
> >
> > Please telex the total by return.
> >
> > The Area Division Manager needs the figures for his report to World Headquarters."

“The moral of this story”, said Lou “is: never give the bureaucracy more information than they absolutely require. They’ll only start insisting on having it!”

Well, this was an interesting change. Here was my manager telling me to treat The System as an opponent, rather than an authority.

Chapter 6 Service as Usual

All but three of the group moved from Cockfosters to STL in Harlow, a distance of some 25 miles by road, but not served at all well by public transport. Most of the staff stayed in their existing homes and commuted by car. Meanwhile I was interviewing and recruiting more new people. STL had a good canteen, used by all the staff including the directors. There was no divisive separate management canteen, as there was in some large companies. Most of us used to have lunch there, and the food was good value, the main courses being subsidised. A favourite dessert was “golden sponge pudding with golden syrup sauce”. Campaigns for healthy eating were yet to arrive on the scene.

The ITT 3200 was the embedded computer in two new telephone exchange systems that were being designed. STC in Cockfosters was developing System X and LCT was developing Metaconta L. Other ITT companies tailored these systems in specific contracts for new exchanges, and all these companies needed computing facilities to develop and customise the embedded software. These computing facilities consisted of centres containing 3200 computers again, with peripheral devices, paper tape readers, punches, magnetic tape drives, line printers and so on. The BSCC and LCT provided the operating systems for these computer centres, the BSCC for those used in developing System X and LCT for the Metaconta L.

In this way, the BSCC was serving the needs of fourteen computer centres scattered in many different places in several different countries. We supplied the operating system, compiler for the symbolic assembly language, which was called SYMBAL, and a collection of test programs for testing the computer and its peripherals. One might think that having supplied this software once to a centre, there would be little need for further visits, but for two reasons frequent visits were often necessary. The first was that the 3200 machine itself was extraordinarily variable. With a computer today, if a peripheral such as a scanner is attached, there is usually just one way to attach it, a single device handler can be supplied on a disk, and with a few adjustments secured by a dialogue (the “installation wizard”) it will work. With the 3200 machine, there were an astounding number of choices to be made. A peripheral could be attached to a choice of channels, interrupt lines and priority levels, addresses for information exchange and so on. Every computer had a different arrangement and the operating system and test programs had to be prepared in advance and, usually, installed on a visit to the centre. Very often the BSCC programmer would find that the

information given did not quite reflect the reality and last minute adjustments had to be made on site and the whole lot tested out. The second reason was that most of the centres seemed to change their computer configurations with remarkable frequency.

Some of the centres were a bit more stable and needed only occasional visits. These were mostly the more distant ones, particularly in Des Plaines, Illinois and Sydney, Australia. I suspect that the distance and cost of visits concentrated the minds of the computer centres to keep their configurations in a stable state.

Another reason for personal visits was that the operating system and software itself was not all that well geared to facilitate these changes. The parameters controlling the peripheral information like channels and interrupt lines were strongly embedded into the code of the operating system. The same information was not even shared between the operating system and the test programs: it was duplicated in each. If all this peripheral-related information had been put together in tables and referred to by the software, then the tables could be altered by a dialogue program (a wizard in today's terminology), possibly even by the staff in the computer centres. About a year into my job I suggested this to members of the BSCC, and they mostly thought it was a good idea. However, we could never get it done because always the priority was to fix the next installation urgently and, more to the point, my management could not be persuaded to agree funding for what they saw as an improvement to our software that was not strictly necessary. In a way, if we had been able to make these changes, we would have been doing ourselves out of a job; amendments following reconfigurations would be much faster and simpler to make, and could even be done by the user. But I was and am sure that other work, perhaps of a more progressive kind, would have been found for us. We were rather like medical practitioners, who strive to improve the health of the population under their care. Even with the best preventative medicine, there will always be a need for doctors.

So the lives of the BSCC members were dominated by requests for software installations from our computer centre customers, and were consequently filled with visits to Antwerp, Paris, Madrid, Munich, Zurich, Cockfosters and, less often, Des Plaines and Sydney. We needed 3200 computer time ourselves to check out the modifications to software that we prepared for these installations. Being away from Cockfosters now, the group made use of a small computer centre at STL, which until then had been mainly used by another group. At first this centre could not provide all the time we needed, and Gerry Jacob suggested that we used some spare time in a similar computer centre in Zurich, which was currently under-used. Even the much travelled members of my group were a little startled by this suggestion. Today one can hire computer time in a public library or a cyber-café, and the process and nature of the transaction is essentially the same: one is paying money to use time on useful equipment owned by another party. But the idea of flying from the south-east of England to Zurich in order to do so seemed a trifle extravagant. Nonetheless, we arranged a few visits to STR in

Zurich and several of my group flew there and used their computer. I remember visiting STR myself, partly to set up this arrangement, but also on a kind of diplomatic mission to establish good working relations with the company. My management encouraged me to do quite a lot of oiling such wheels in order to keep everything running smoothly. I remember STR being situated in extremely pleasant surroundings, with views of a Swiss lake and mountains and my hosts there being very relaxed. I think they did not have much to do at that stage and seemed not to be under much pressure of work.

The bureaucratic procedures of STL required me to fill in a purchase order for this computer time, which I duly did. Then the BSCC programmers flew out taking with them the software they wanted to test. They simply carried these as rolls of punched paper tape in their hand luggage. They spent time on the Zurich computer, made any amendments necessary after the tests they carried out, and came back home bringing a possibly updated version of the software and the confidence that it now worked successfully. Then I received an invoice from STR for the computer time we had used, and I sent this to the STL accounts department, authorising it for payment.

Here the trouble began. At that time there had been quite a large number of scams in which rogue traders sent spurious invoices to big companies for services and goods that had never been provided. The fraudsters relied on companies processing such large numbers of invoices that they would not spot the mendacious nature of these demands, and these invoices were frequently paid. Most big companies were getting wise to this kind of fraud and took precautions against it. They required that invoices quoted a purchase order number originated by themselves, and verification was required that the goods had been delivered. STL demanded that goods were received through Goods Inwards, a physical door to the laboratories, and a Green Ticket would be written out, which would find its way via the originator of the order to the accounts department. In the case of a service, a visiting representative from the supplying company would normally provide this. The visitor would have to fill in a lot of paperwork on entering and leaving the building, and these pieces of paper would similarly be correlated with the purchase order. But when we used computer time in Zurich, nothing passed through the doors of the STL Goods Inwards, nor did any visiting rep come in and out. I started receiving puzzled phone calls from clerks in the accounts department telling me they could not pay the invoice I had authorised. “We haven’t received a Green Ticket!” they said.

I ended up going to see Dennis Gray, the purchasing manage, a cheery good-humoured man I had dealt with several times already. I tried to explain the nature of the transaction. At some point Dennis said to me, “I think I’ve got it, Tim. Do they ship their computer to STL so you can use it, and then you ship it back to them?” Oh, no, not quite. The computer is far too big and heavy to do that. It is much simpler for Mohammed to go to the mountain so to speak: for my guy to go to Zurich taking the software with him and use the STR machine. “Suppose you

have a cat", I said. "You decide to take your cat to the vet's to give it its annual health check. The vet has a big X-ray machine, bolted to the floor for safety and costing £50,000. You put your cat under the machine and see that, happily, all is well. You can be confident that your cat will probably be fine for the next year. So you pay the vet for the use of his expensive X-ray machine and go home happy". We ended up agreeing that I could keep a small stack of Green Tickets in my office and, when I was satisfied that we had received computer time which corresponded to that on the invoice, I could send the Green Ticket to the accounts department. But Dennis still looked a bit uncertain about the whole business.

This did indeed make me ponder exactly what we were paying for. If, as was moderately likely, no change was made to the software being tested, if all the tests were successful, what had we bought with our money? An increase in confidence that the software would work? That was a pretty intangible kind of commodity. Had the software increased in value after we had tested it, even if no change had been made to it? Perhaps it had. Our own knowledge had increased, knowledge about the reliability of the software, but it was intriguing to think that its value had increased even though the software itself was physically exactly the same as when it set out on its journey to Zurich.

Many years later, when working for Praxis, once again I came across this question of how much a piece of software is worth. The value will depend very much on how much we know about it, whether we know how to use it, and whether we can understand it well enough to maintain it, that is to find faults in it and mend them, and well enough to alter and extend it. Almost all software is going to be working in a changing context, a changing world, and so it has to be adaptable. Software can only be understandable and adaptable if it is well designed and well engineered, and if it is documented with descriptions of how it can be used and how it has been designed. Without these documents, even if it is well designed and put together, software can be virtually worthless. When members of the BSCC had tested their software in Zurich, they understood it better, knew of errors that had to be put right, and could write the documents that certified the passing of tests.

So started a routine. We would receive requests for installations of software on new or altered machines and their peripheral devices. New versions of peripherals were often being attached to the central processors, and we would have to write handlers for them, which would go into the operating system, and we would write new test programs for them. We would often receive fault reports relating to our software. We had a procedure for processing these, which was the subject of yet another ITT standard. The procedure consisted of passing several documents back and forth, Change Requests, Change Notes and so on. Sometimes a fault was reported, but the faulty behaviour could not be reproduced. This was often caused by unreliable hardware, in other words by electronics inside the computer that misbehaved from time to time. Some sites, who had less rigorous maintenance practices, were more prone to this happening than others. BTMC, the Bell Telephone Manufacturing Company, in Antwerp,

was the administrative host for the Computer Engineering Centre where Lou Flowers and Gerry Jacob, my managers, worked. In BTMC there was another systems software group, the 1600 BSCC. This group, smaller than mine, served the needs of computer centres that used the 1600 machine. There were only a few of these, and all of them were in Belgium, within BTMC. But BTMC also had a 3200 centre, and its maintenance and general administrative procedures were very haphazard. Many times they would report a fault, and we would find that they were not actually using the most recent versions of the software we had already supplied. One of the programmers in my group, Cliff Lamb, described how he saw with some despair the "software administrator" open a cupboard which was full of boxes of programs on paper tape, jumbled together and without any filing or reference system. He would just reach for the nearest one, without checking that it was the most recent issue. All computer memories had an error checking feature called parity checking. Each word of memory, usually 32 or 64 bits (4 or 8 bytes) had an extra bit, called the parity bit. This was always set so that the total number of bits in the word which were set to a 1 was even. The parity of the bits in a word, that is whether the total of 1s were even or odd, was checked on every access. If the parity was ever odd, an error had occurred, the machine had misread or miswritten information, and the machine would stop. The parity checking could be switched off, but this was most unwise as it meant that the machine could behave in an aberrant fashion, and not follow the instructions that the software in it was telling it to do. Cliff discovered that the BTMC programmers were routinely running their 3200 machine with the parity checking switched off. "It keeps on stopping of we don't", they said. Cliff was horrified, and explained to them why they needed to keep the checking switched on. Because of the unreliable machine, they had to restart it often, but at least it would not make it spuriously look as if the software was at fault. He felt a lot of sympathy for the programmers having to work with such unreliable hardware.

There were two other ITT research laboratories in Europe. As well as LCT in Versaille, there was the ITTE laboratory in Madrid. Felix Vidondo, a man of some charisma, managed the software research there. He recruited programming staff fairly regularly and a tradition had grown up that he would send a couple of his trainees to the 3200 BSCC to gain experience. So two of my staff were in fact from Spain and took part in the regular activities and duties of the group. I must say that I would not have called these individuals "trainees". They had had two or three years post-graduate experience and were doing the same job in principle as the rest of the group. The two of them, Paco Lopez and Manuel Varela, were to spend several years in my group. Indeed, Manuel married and brought up a young family in England.

My manager, Lou Flowers, frequently asked me to accompany him on a trip abroad, or to go and investigate some situation myself, at a few days' notice. Within a week or two I realised that my social life had henceforth to be confined to the weekends. Making a theatre or cinema booking midweek in advance was no longer possible. The problem of the two rival operating

systems for the 3200 machine was a continual bugbear, which engaged the attentions of many figures in ITTE and even some in the headquarters in New York. It was clearly wasteful to have two software systems doing exactly the same thing, being continually upgraded and maintained. Yet there seemed no way out of it. The issue became the subject of discussion in numerous meetings. Don Combelic took a fairly uncompromising stance, not allowing LCT to register their operating system software as ITT products and generally denouncing it as a kind of rogue artefact that shouldn't be there. Other ITTE staff recognised the de facto situation that many organisations used the LCT software as part of their essential work, but no-one could see a way out of the unsatisfactory situation of there being two rival co-existing systems. Many managers kept trying to persuade Don to relent about granting ITT product status, thinking that at least it might help LCT to become a bit more cooperative in general, But Don remained adamant for some years.

It occurs to me that this wasteful but rival situation of competing software reflects that which prevails today in personal computers. The Microsoft Windows range and Unix are rivals offering the same function, as does the Mac OS on Macintosh machines. The different versions of MS Windows to all intents and purposes compete amongst each other, for upgrading becomes difficult, often requiring hardware upgrades to support them, and then elderly but entirely functioning application software has to be replaced to be compatible with the later versions of Windows. But if one stays with an old system, new facilities, even replacement hardware, become unavailable through obsolescence. There are almost no recently produced pieces of hardware or software packages that will run even on Windows 95 any more.

Don Combelic was in a high ranking position. He had reached this partly as a result of making an extremely valuable technological contribution. Telephone exchanges used to be all electromechanical. When you dialled a number from your domestic handset, as the dial rotated it sent electric pulses down the line to the exchange. The bigger the digit, the further the dial had to rotate, and the more pulses were sent. Dialling a 1 sent one pulse, a 9 sent nine pulses and a zero sent ten. These pulses operated electromagnetic relays in the exchange which switched the connection through to the telephone belonging to the number you dialled. So telephone exchanges consisted of many relays and vast arrays of interconnecting wires. When an electromagnetic exchange was replaced by a computer controlled one, the incoming and outgoing wires were cut, the old exchange removed, the new one put in its place and the wires reconnected. This process was called "cut-over" and would have to be done as quickly as possible to minimise the suspension of the service. After that there would usually be many teething problems: would the new exchange work properly, would it handle the pattern of telephone traffic in its environment of incoming and outgoing calls?

Combelic devised a test rig called "environmental simulation". Another computer would be programmed to simulate the demands for connections produced by the telephone traffic that

was typical of the environment of the old exchange. Then the performance of the new system could be tested as exhaustively as desired before cut-over. Programming the environmental simulation could be as extensive a task as programming the exchange software, but the effort and expense was worthwhile and turned out to be an extremely effective way of enabling the technological upgrade from electromechanical to computer controlled exchanges. By the 1970s Combelic had become something of a software grandfather figure in ITT.

Computers had been used for simulation of one sort or another for many years already. At ULACS Chris Hobson had been writing an Algol60 program to simulate the Atlantic ocean for the Meteorological Office and the language Simula67 was devised in the first place for various simulation tasks. Simula67 had the first features of Object Orientation, which are the principal properties of the present-day Java language. Computers were beginning to be used to simulate financial economic trends, performance of stock markets, seismic activities, weather and much else. So using computers to simulate technological phenomena like telephony traffic was in a sense a natural course to take. This use of computers for simulation has since blossomed. In the 1980s the University of Oxford developed the ELIZA program to simulate the interaction of a psychotherapist with a client. This was sufficiently successful that trial users wanted complete privacy while they were communicating with the program, despite its relatively primitive, textual interface. Every computer game today involves simulation of visual scenes, events and a narrative.

Don Combelic had his own proposal to resolve the problem of the dual operating systems for the 3200 machine. He proposed that both should be replaced by a much superior system, which he called DPSS. He believed that this new system would show such superiority over the BSCC and LCT systems that all users would want to migrate to it. He arranged for the ASG, headed by Bob Parenti, to start developing DPSS. Don had not been able to secure any funding for this activity, mainly because many other managers were very sceptical about his plan. They thought that introducing a third rival operating system might well make matters worse, not better. So Don managed to get the work done by stealth, so to speak. I was due to have two more trainees for the BSCC from the ITTE laboratories in Madrid, but Don arranged for them to stop on the way at LCT for a “temporary period”, and assist with the DPSS development. This became something of a logistic struggle between me, my managers and Don, especially since the two trainees were funded from my BSCC budget, and indeed rather later Don referred to his having “stolen” these staff from me.

There were numerous discussions and arguments about the wisdom or otherwise of developing DPSS. I had been thinking about a different strategy to replace the two existing systems. I had worked out a series of piece by piece modifications to the two operating systems in which sections of each software would be replaced. The two replacements would be identical, so over the course of the strategy, the two systems would merge together, until they were the same. The progressive work on modification could even be carried out jointly

by the two teams. The scheme would require the cooperation of the two groups, something I knew would be very tricky. I put this idea first to Lou Flowers, then to Gerry Jacob, and finally at a meeting with both of them, their manager and Don Combelic. Don was strangely quiet during this meeting and at one point left the room. I wondered if he felt unwell or even angry. My proposal did not receive his blessing and so did not move forward.

Some few months later Don, Bob Parenti, Gerry Jacob, Lou Flowers and I met once more and yet again spent some hours discussing the problem of the rival systems. We had reached an impasse. Then Gerry suggested that we hired a consultant to consider the problem and report back to us. Here was a possible way forward. We all eagerly agreed that this seemed a good idea. Getting a fresh view on the dilemma from someone outside the company, who could take a detached look at it and see the wood for the trees could add just the insight we needed. We started trying to find a name we all knew, someone whose experience and judgement we could all respect enough to have confidence in them. Don and Bob suggested a couple of names I had not heard of. Gerry, not having a software background, was relying on me to vet any suggestions from the other two, so I demurred. I suggested Tony Hoare who had been my manager at Elliott's, where he was responsible for the first commercial Algol60 compiler, and who now in 1974 was Professor of Computing Science at Queens University Belfast. Parenti in turn demurred and, after a few moments thought said "How about John Buxton?" Apparently they had both worked together at IBM laboratories. I had known John Buxton slightly when I worked at ULACS. He had been at ULICS, the Institute of Computer Science at London University, and ULACS and ULICS had shared computing and other facilities in the same building in Gordon Square. John had worked on significant systems software projects, parts of the Atlas system and the CPL compiler[1]. I also remembered him as being a man of solid good sense and judgement, and he certainly had the right kind of technical background. I said I would be happy for John Buxton to perform the rôle. So the ambience of the meeting became more relaxed once again: we had reached at least some kind of interim agreement. I was asked to contact him and make the necessary arrangements. It seemed that I was the only one with a budget that could reasonably easily absorb a short consultancy contract, so I was to handle the contract with him too.

I had no idea where John Buxton was working at this point but, back in my office in Harlow after a few telephone calls I managed to find him. I explained the situation and that we were looking for a consultant to give us advice on a problem of technical strategy. The task should require about four days' work. It turned out that John had been on an assignment in Hungary for several years and had just returned home. He was between employments, prior to taking up a chair at the University of Warwick the next academic year. So a piece of consultancy work was, it seemed, most welcome. We arranged to meet and John came to STL. I gave him more background about the rival operating systems and Combelic's proposal for DPSS, and

[1]See Barron et al 1963.

told him a bit about the factions involved. John took it all in and I made arrangements for him to visit LCT and find out about the LCT operating system and DPSS.

John Buxton studied the documents describing the two rival systems and the work done on DPSS to date. Then we had a last meeting at LCT. This time quite a lot of the players were present: Gerry Jacob, Don Combelic, Don's assistant John Devoil, Bob Parenti, Lou Flowers, myself and, I think, several others. John Buxton ably and in a relaxed way presented his findings. His view was that the work on DPSS should never have been started. The best way to achieve technological advance is by incremental development, not by sudden revolutionary change. He strongly stressed "incremental development". But having come as far as this on the DPSS path, it would probably be best now to continue. He found that the individual programming staff involved whom he had met seemed well competent enough for the task. In a nutshell, that was the gist of his findings. It was as if we had travelled on a mountain trail and asked an experienced stranger if we were on the right track. The stranger advised us that this was not the best way to go at all, but now we were here, we might as well continue; bear round that way and we would reach our destination. I felt a small private glow of pride, because my idea of merging the two operating systems would have precisely been an "incremental development". But I knew that that idea was no longer up for grabs.

Everyone became lively and started talking about the consequences of this, effectively a cautious recommendation that DPSS should proceed. Gerry asked John, "This is probably an unfair question, but do you have any recommendation about who should head the team to develop DPSS?". The others all declared that this was indeed not a fair question, that John should not feel obliged to answer it, but John said, "Oh, I rather like unfair questions, and yes, I have come to a view about who would be the best person to lead the work". Everyone became a bit startled, I think. Here was an external consultant who was about to recommend a personnel matter, not something he was asked to do, and possibly liable to cause embarrassment. "That person" said John Buxton, "is John Devoil". John Devoil could not help looking flattered. At one point a little later, he was enlarging on some detail, and Gerry said to him, "Hey, you haven't got the job yet!". "I know, I know" said John.

After some time the meeting broke up, with everyone looking reasonably content. I mused that John Buxton had been rather clever. By saying that we should never have gone along the DPSS path, but that the best thing to do now was nonetheless to continue along it, he had at least partially satisfied all the factions. Those who had opposed DPSS had had their views confirmed, but Don and his allies were given the go-ahead to proceed with it. All round, honour had been satisfied. Furthermore, whatever the outcome if DPSS was completed, if it came to be accepted or not, John Buxton would be proved right, at least in substantial part.

In fact, what happened was that DPSS always struggled to receive ITT funding. Don continued to try to appropriate effort from my staff, and succeeded in doing so from time to

time. DPSS was eventually completed, was used to a limited extent by one company, but never came into widespread service.

Meanwhile, most of the business was continuing as usual. Like all except the very smallest companies, STL, being part of STC, had personnel procedures, including annual salary reviews. STC had a company wide scheme of appraisals. I used to review the performance and salaries of my own staff, and mine was reviewed in turn, but in my case no face to face interview took place. I assume this was because my managers were not actually employed by the company, but by BTMC. The personnel department at STL would send a letter to each individual, informing them of their salary increase for January each year. These letters were of standard form, starting with the words "Thank you for your contribution to STL in the last year", and were sent to everyone's manager to sign and pass on to the employee. Gerry Jacob did not feel he could sign a letter thanking me for my contribution to STL, because he did not represent STL. So I never received my annual letter. Each year I discovered what my salary increase was by examining my pay slip at the end of January.

Staff turnover was fairly average for a software team. I recruited more people fairly regularly, including the first female member of the team. Software was an industry that was in some areas a mainly male preserve, but in others the reverse. At ULACS and RADICS there was a small majority of women in the teams, but in STC there were very few. Firms which had a mainly engineering tradition would in general be dominated by men, but where the emphasis was principally that of computers and programming, women graduates in mathematics and computer science felt more welcome. Computer Science degrees, non-existent when I was an undergraduate, were now offered by many universities. Denise Brown had a first class degree from City University and showed a great deal of alertness and intelligence at her interview. I instructed the personnel department to offer her a job. She had mentioned that her husband, David Brown, a computer engineer, was also considering a job at STL in another department. Only later did I fully realise that they were applying to STL as a kind of package deal – they would either both come or neither would. I met David from time to time. He was a man of considerable ambitions and later joined Motorola UK, eventually becoming chairman, and was made president of the IEE in 2003. Several other new members joined the BSCC, including Tani Haque. When interviewing him, I was finally convinced when he mentioned in passing, right at the end of the interview, that he had sold encyclopaedias during a summer vacation when at university. He had been so successful that the firm wanted to make him area manager. "But I had to go back to finish my degree!", he said. I thought we could use some sales skills.

As well as technical staff, the group needed administrative staff. At first we engaged a secretary from an agency. Some of these were excellent, and I asked several if they were interested in a permanent position, but they always had some reason not to. Others were very temporary indeed, leaving of their own volition after two or three days. One actually left at

lunchtime, leaving a half typed sheet of paper in the typewriter, with no explanation. I could tell that the personnel department began to eye me suspiciously, wondering if I was making unwelcome advances to these mostly young women. I had always behaved with decorum, but I felt quite uncomfortable under the gaze of the personnel department for a while. Eventually one of the temporary secretaries asked if she could become permanent, and I agreed, against the advice of some of my staff. She was not the most efficient secretary, but I was beginning to feel she would be better than a continual string of temporaries. A year or two later we engaged a second "clerk", as Lou Flowers called them, for our need for typing documents, filing them and arranging travel was considerable. I also believed that it was a good, cost effective policy to use clerical staff to do as much as possible for the programmers, leaving them to spend more of their time on technical work. Other managers thought that the more they spent on secretaries' salaries, the less they could spend on technical staff, and would minimise their administrative budget. I have always thought that this was a mistaken policy, which led, among other things, to a more tedious working style for the technical people.

I continued to experiment with my policy of matching the work required from my staff to their abilities. Project leaders found it a chore to deal with project accounting. So I engaged a business studies sandwich student for a year, and got him to do the project and other accounting for the group. I encouraged everyone to give him tasks where possible, and kept an eye myself on his workload to make sure he wasn't overwhelmed. I hoped that this arrangement would relieve the programmers from the aspects of work that they found less interesting, but the experiment had only limited success and I did not repeat it. Delegation itself takes time, and it does not necessarily save effort. A few years earlier I had had difficulties persuading the programmers to fill in their time sheets on time, until one day I sat back and thought to myself, "why do I need their time sheets anyway?". I needed them to supply the accounts department with the information necessary to calculate the project accounts, which I in turn included in my monthly report to Gerry Jacob. So I decided to hand the problem to the project leaders: I explained to them the need for project accounting and delegated to them the preparation of the project information and asked them to send it to the accounts department each month and report on their project accounts to me each month. For this they needed their own and their project members' time sheets. I no longer needed to ask for anyone's timesheets, except those of the admin staff. The project leaders were quite pleased to accept this extra responsibility; it elevated their status, giving them more of a management rôle. The time-sheet problem disappeared overnight. Even I was surprised.

We started to engage a computer science sandwich student each year, following a campaign from the personnel department. STL had an institutional vocation to foster research and education, and their liaisons with universities gave a motive to provide meaningful work experience to sandwich students. I had myself spent a year at Texas Instruments between school and university, and had deliberately done vacation jobs in the electronic and computer

industries, to gain work experience. I was quite enthusiastic to engage a sandwich student for these reasons, and because I thought it could also enable the more experienced programmers to concentrate on the more advanced aspects of their work, if only to a limited extent. So Nick, an undergraduate from Nottingham Polytechnic, as it then was, joined us. He rapidly learned how to prepare software installations for the 3200 BSCC and other very useful tasks. One day we had a request for an installation at LCT in Paris and none of the full-time programmers was available to do it for some time. I asked Tani Haque, whom I had come to nominate as my deputy when I was away, if he thought Nick could do it. Tani thought he could, and so I went to consult the personnel department. I was not sure whether we were allowed to send a sandwich student abroad on business, or whether the personnel department might advise against it. But they saw no objection at all; indeed, they thought that it was a good thing to give the student a challenging task if I thought he was up to it.

So I asked Nick into my office. “Nick”, I said, “have you got a passport?” His jaw dropped visibly and I went on carefully to explain what we wanted from him and gave him advice about the hotel to stay in and how to reach LCT, the people he should speak to and so on. He was willing to go on the trip and carried it out without any apparent difficulty.

After I had been in the job for some three years, Lou Flowers, my manager, left. I got the impression that he was made redundant, for I and one or two other staff who had reported to him now simply reported directly to Gerry Jacob. I engineered a meeting to go to in Antwerp so that I could be at his office on his last day. He was rather surprised, and pleased, to see me there. It was the first time I had had the temerity to organise a trip for myself without consulting him first. With Lou’s departure, I was now in charge of the 1600 Basic Software Control Centre, as well as the 3200. The 1600 BSCC’s manager had been promoted elsewhere and so the group was temporally without a leader. Lou had been guiding it himself while working out his notice. But I could not do this effectively from STL, in a different country, never mind a different town. Lou advised me that none of the 1600 BSCC staff were competent enough for the job. One member had the technical ability for it, but not the diplomatic skills needed to interact with other groups. I was asked to find a leader for the group, possibly from the ranks of the 3200 BSCC.

I wondered how to set about this. Taking on such a position would involve the aspirant’s moving location to Belgium, managing and inspiring staff of a different nationality and working in an environment with unfamiliar traditions of employment protocols and working practices. But in the multinational ITT these features of work were normal, one might almost say run of the mill, and taken in one’s stride. I decided to take the slightly risky but simple course of announcing the vacancy to all the BSCC staff and inviting them to see me if they were interested. I reckoned that in fact any of them could handle the job. After all, they had had me as an example of how to manage a group of programmers! I also wondered how I would choose the candidate if there was competition. But in the event, only one knocked on

my door and said he was interested – Cliff Lamb. So Cliff went to live in Antwerp and led the 1600 BSCC, and proved to be a popular and successful leader of the group. Having worked for many years under the rigid approach of their former manager, the group found Cliff's style of open, fair and relaxed leadership a welcome change.

About this time, when I opened my pay slip at the end of the month I found that my salary had taken a sudden and welcome leap upwards. I deduced that I had been promoted, but once again this step function in my pay was the only indication I received. Some six months later, after Lou had been gone some time, Gerry Jacob implicitly confirmed this. Apropos my taking responsibility for some matter, he said: "You're Lou Flowers now, remember?"

Now that I reported directly to Gerry Jacob, instead of Lou, he would visit me in Harlow and I visited him equally frequently in Antwerp, sometimes elsewhere. Gerry was keen on modern management principles such as "management by objectives" and the principle of delegation. Delegation was believed to motivate staff and give them a sense of self value and confidence. Gerry liked to delegate. Every task that could be done by someone else, he would choose one of his staff, often in rotation, and give them instructions on what he wanted done. So I would often go to his office in Antwerp just for the purpose of carrying out one of these delegated missions. It certainly gave me a lot of insight into the context in which his organisation, the Computer Engineering Centre, worked. One regular task was the writing of his monthly report, which would be sent to his manager at ITT Headquarters in New York. When it was my turn to do this I would sit at a spare desk in the CEC and read through all the monthly reports of Gerry's staff, and try to compose them into a unified whole. All these reports had a fixed format: Highlights, Achievements, Problems and Red Flags. A Red Flag was a serious problem. I would consult the authors if I had difficulty in interpreting their reports. I remember one occasion when two of Gerry's staff had reported on the same event, one of them describing it as a "Problem" and the other as an "Achievement".

Chapter 7 Reorganisation and Research

The Computer Engineering Centre moved from Antwerp to Velizy, just outside Paris, next to LCT. Gerry and most of his technical staff uprooted and found new homes in France and the Centre detached itself from BTMC and became administratively part of LCT. Gerry was happy enough still to be in a French speaking country, after having spent a great deal of effort learning the language. Much of the CEC's equipment was left behind, bequeathed to the 1600 BSCC. I was to arrange payment for this to the CEC, so that they could in due course buy more. This was much more practical than physically shipping the machines from Belgium to France. But LCT refused to accept the payment; as far as they were concerned, nothing belonging to LCT had been sold, so they could not accept payment for it. I went to see the Comptroller at STL about this. He was quite amazed. "If someone wanted to give me eight thousand pounds, I'd accept it and think of a way to make it legal afterwards!", he said. I was

bemused to hear this from our respected chief accountant. We all shrugged our collective shoulders and I kept the sum in my budget, with an understanding that we could repay the CEC in various ways as opportunities presented themselves.

Advances were being made in the software world. We were in the mid-seventies. I had been in the industry for about thirteen years. In the mid fifties Noam Chomsky had carried out canonical work in linguistics on defining the grammars of human language. This was the starting point for finding ways of defining the syntax of computer languages. Writing a compiler for a programming language is far easier if its syntax can be precisely defined. Indeed, people have written "compiler compilers" which input a syntax definition of a computer language and produce the front end, that is the parser, of a compiler for it. I used Brooker and Morris's Compiler-Compiler while I was at ULACS in the late sixties, and now there are numerous such compiler-compilers. YACC, Yet Another Compiler-Compiler, is probably the most well known one today. The first generally accepted notation for defining syntax was developed by John Backus for the definition of Algol58 in 1958. This was called BNF, Backus Normal Form[1]. BNF was extended by Peter Naur for the definition of the better known Algol60, and the notation became Backus-Naur Form. Now there is an international standard for BNF developed by the British Standards Institution and adopted by ISO, the International Standards Organisation, and IEC, the International Electro-technical Commission[2].

A lot of theoretical research work followed the Algol60 development in the sixties. But the really interesting problem was the much more difficult one of defining the meaning of a computer language, rather than its form or syntax. If that could be cracked, the writing of compilers could become a much more precisely defined task and the quality and correctness of compilers could be greatly improved. As it was, compilers for the same language varied one from another, largely because there were vague areas in the definitions of all the languages, open to interpretation, no matter how hard those defining the languages tried to make them exact and complete. Nonetheless, during the early sixties more and more intricate work was done on syntax definition. Somebody likened it to a joke: a policeman at night sees a man looking under street-lamp. He goes up to him and asks him what he is looking for. "I'm looking for a coin I dropped", he replied. The policeman asks him, "Did you drop it under this lamp?" "No", replies the man, "I dropped it over there". "Then why are you looking here?", asks the policeman. "Well, I can see under this lamp, but I can't see over there" is the reply. There seemed to be a reluctance to investigate the more difficult problem of the meaning of languages, so some researchers spent time on the largely solved problem of syntax.

[1]See Backus 1960.
[2]See ISO/IEC 14977 1996(E).

However, there were some efforts being made in the late sixties into the meaning, or semantics, of computer languages. In 1963 John McCarthy outlined a theory of computation which, among other things, separated abstract syntax from the more cumbersome concrete syntax of a language[3]. Abstract syntax separated out the essential parts of a grammar, ignoring the precise form of the components, the punctuation and spelling so to speak. This eventually made defining a language's semantics a lot easier. In 1965 Peter Landin related the actions of Algol60 programs to a branch of mathematics called Lambda Calculus[4]. (Lambda Calculus is a notation in mathematical logic, devised by Alonzo Church[5]). In 1966 Christopher Strachey used an extended form of lambda calculus to help define the meaning of programs[6]. His approach subsequently became known as Denotational Semantics. In October 1969 Tony Hoare published a seminal paper in the Communications of the ACM (Association of Computing Machinery), "An Axiomatic Basis for Computer Programming"[7]. This paper laid out some principles for deducing whether or not a program was correct, that is, whether it achieved its desired result. But in so doing, the paper gave a means of defining the semantics of the individual and composite instructions that lay at the heart of most conventional programming languages of the time. Two lines of research sprang from these pieces of work. One led to being able to define the semantics of computer languages. The other led to a way of developing programs and proving them correct.

The word "semantics" just means "precise meaning". In popular use it has come perhaps to have a negative association. When someone says "That's just semantics", they mean that their disputant is responding to the literal precise meaning instead of the intention of their words. In normal discourse between people, responding to the semantics of someone's words when the intended meaning is clearly different is deliberate misunderstanding, an act of social hostility. But in most scientific disciplines the precise meaning, the semantics, of one's descriptions are very important. Precise description is much to be desired.

In 1975 Edsger Dijkstra, professor at the Technical University of Eindhoven, published a seminal paper, "Guarded Commands, Non-determinacy and the Formal Derivation of Programs"[8]. In many ways his ideas were similar to the Hoare axioms of computer programming, but there were important differences. In particular, his notation smoothly led to a way of proving that a program fulfilled a desired objective. Dijkstra wonderfully illustrated this method of proof in a small but classic book, "A Discipline of Programming", a year later[9]. He took a number of elegant examples of problems and walked through his proof method, developing the program as he went along. Some of the problems had never been solved by computer

[3]See McCarthy 1963.
[4]See Landin 1965.
[5]See Church 1941.
[6]See Strachey 1966.
[7]See Hoare 1969.
[8]See Dijkstra 1975.
[9]See Dijkstra 1976.

before, the most notable one being the convex hull in three dimensions. Suppose you are given a number of points in three dimensional space, no four of them lying in the same plane, like a scattering of stars in a galaxy. If you take a set of three of the points, they define a plane. If all the other points lie on the same side of this plane, define these as "boundary points". Another way to think of the boundary points is to imagine a large balloon enclosing all the points. Then shrink the balloon until it is in contact with all the points on the extremity of the set. Those points are the boundary points. The result will look rather like a Buckminster Fuller geodesic dome, but completely closed. The problem is to write a program to find all the boundary points. This is not an easy programming problem by any means, but Dijkstra used it to illustrate his method of developing programs, producing a proof that the program is correct as he went along.

I was fascinated by Dijkstra's method of developing programs. In some ways, the technique seemed back to front. Instead of taking a program and stepping through it proving that it achieved the desired result, you take the last statement of the program and work out what are the minimum preconditions required so that after it is obeyed, the desired result is delivered. Then you step back progressively working out the minimum preconditions until you reach the program's starting point. If then there are no preconditions at all, you have proved that the program is correct.

Dijkstra's method, his "discipline" of programming, also involves separation of concerns, breaking up a problem into sub-problems, and other techniques of simplifying a complex problem. I tried it out on some programming problems of my own and became enthusiastic and convinced by it.

I was still in a management rôle, and had not done any real programming myself for a long time. I was missing the technical challenge and, at the same time, there was one programming task I longed to do. I have mentioned compiler-compilers, which are often and perhaps more accurately called parser generators, because they generate a parser for a language. I had used Brooker and Morris's compiler-compiler years earlier at ULACS. I had also used David Hendry's language, BCL, when working for RADICS. BCL also performed the rôle of a parser-generator. The Brooker and Morris program was difficult and awkward to use, and BCL was considerably easier. It would be easier still if the input to a parser were a grammar written in BNF, the notation devised by Backus for the Algol60 definition and already the subject of a British BSI standard. I decided to write a compiler-compiler that would input standard BNF. The only computer I had access to was the 3200 machine, and the only languages available on it were the symbolic assembler language and ESPL1, the autocode developed by Bob Parenti's team. I used ESPL1 and developed a parser generator that used BNF for its input. I used it to produce a couple of small utility programs and sent details of it round a number of programmers in the company. One programmer in BTMC in Antwerp,

Rudi De Belie, used it to produce an editor, but the whole exercise was mostly for my own elucidation.

Amongst the community of software developers, as well as these rather theoretical advances, there was a lot of effort being put into "software engineering". Until the late sixties, programming was seen as a somewhat individual task, carried out by loners sitting at their desks and producing inscrutable works of art, programs that produced wonderful efficient results but which were comprehensible to no-one but their authors. There had been a lot of persuasion to move away from this attitude, requiring programmers to document their programs, to explain how they worked and generally to be "public" about them. But in the 1970s a much stronger push came to turn programming into an engineering discipline, with the writing of programs carried out by coordinated teams of people working together and to a budget. Estimating the cost of developing software and the time it would take was notoriously difficult, and software managers began to follow a variety of initiatives. Programmers were encouraged to stop thinking of their programs as their own individual pieces of cherished work. Programs would become team efforts, using peer reviews, walk-throughs and other ways of trying to ensure that the end result would be delivered to specification and on time. Cost estimating became a big effort, and the "software development life cycle" became a big topic. Too often programmers had embarked on writing the code for the program prematurely. Emphasis was now put on producing a design, which was an abstract, more general form of how the program was going to work. The very earliest of these were called flow diagrams and consisted of a diagram showing the program's flow of control, with lines, arrows, boxes and decision points. But flow diagrams tended to describe the program at a low level of detail and did not give a view of its overall design. More general and sophisticated design methods proliferated, all enthusiastically promoted by their originators. JSP, the Jackson Structured Programming method was one of the better of these. A movement called "structured programming" had emerged and was very strong. It was all started by a two-page note published by Dijkstra titled "Go To Considered Harmful"[10]. "Go To" instructions appear in the repertoire of every computer's machine code and were present in almost every high level language. Dijkstra criticised the use of these instructions, claiming that they led to unstructured and arcane software. To quote the obituary of Dijkstra by Krzysztof Apt[11], the note "led to a huge uproar" of controversy. But his view came to be accepted after a few years and, as Apt writes, "thirty years later the Go To statement shines by its absence" in the Java programming language. I recalled that coping with the Go To statement in the Algol60 compilers that I wrote when at RADICS occupied a truly disproportionate amount of effort and compiler code.

[10]See Dijkstra 1968.
[11]See Apt 2002.

So Structured Programming, of which JSP was a particular flavour, became a watchword in programming development. But structured programming was mainly about the design of programs. People began to talk of the "software development life cycle". Before writing the code for a program, one should produce the design. The design may be produced at several stages of detail. Before producing the design, one should however write a functional specification. The specification defines what the program will do, without saying anything about how it is going to do it. The design, by contrast, defines how the program is going to work. This separation of focus between the two stages was an example of the Dijkstra principle of "separation of concerns". After the code is written, a testing phase begins and can lead to repetitive revisions and repetitions of the previous stages. Many models of this process were put forward, the first and simplest one being the Waterfall Model, because the picture of it resembles a fall of water dropping from one pool to another. All agreed that it was very important to get the early stages right before embarking on the later ones. It was no use producing a brilliant design if it did not do what the specification demanded, or if the specification was wrong. More attention to the design would lead to fewer errors and less time testing and retesting, which could be extremely time consuming. In the later part of the seventies a huge amount of effort was spent studying this syndrome. It was found that attention to the early part of the life cycle would save a great amount of time detecting and mending errors later. One movement encouraged programmers to read each other's code before committing it to testing. Doing this, typical error detection rates were 600 during code reading, 300 during unit tests, 200 during system tests and 15 detected when the software was in service. The amount of effort required to put these errors right would dramatically increase as the life cycle progressed. The correction of errors after the software had been delivered required orders of magnitude more effort than during the earlier phases.

Some years later an earlier stage was added to the life cycle, that of analysing requirements. Customers for software very often could not specify exactly what they wanted, so even if a customer agreed a specification for some software, it might not reflect their real-life requirements. This mismatch has resulted in some notorious and expensive disasters, such as the system for handling emergency calls for the London's ambulances. *** more examples here, with dates. Look up on internet*** Requirements elicitation and requirements analysis were to become hot topics a decade later in the eighties. But in the nineteen seventies their significance was still not generally recognised. <More on the push to quality – look up some stuff>

Meanwhile, my own work environment was changing. ITT was selling off STC – they referred to this process as "divesting themselves" of a company. STC was to become an all-British company. STL was traditionally the laboratories of STC, so STL was going to be an ITT company no longer. I breathed a private sigh of relief, for I had always felt a little uneasy at being employed by a multinational conglomerate. A company that employs eight million

employees has an uncomfortable and possibly dangerous amount of power. Unfortunately my relief was to be short -lived; we swapped a tolerant US management style, which rewarded initiative and listened to innovative ideas, for a rigid, rule-bound British one. But it took a couple of years for the change fully to take effect.

My group, the BSCC, used a 3200 computer centre at STL continuously. After a couple of years its chief operator left, to go into the Anglican church and study to become a priest. He was replaced by a new chief operator. We began to have many problems with the service from the computer centre, and I sent frequent memos of complaint to the manager whose department included the centre. After a while, he and I had several meetings with our mutual division manager, Dave Dagwell. Dave devised a brilliant solution, at least, brilliant from his and others' point of view. He transferred responsibility for the computer centre to me, and congratulated me on this minor promotion. Now the chief operator reported to me, and I had no-one to complain to except myself. Indeed, there were a few other users of the 3200 computer centre, and now I became the recipient of their complaints. At least I had a small promotion, with a modest extra increase in salary. I had to field many complaints about the service from the centre and the performance of its chief operator. In due course we had to go through the process of giving him official warnings, the later ones in writing. According to the rules of industrial relations, after three warnings, he could be sacked. He received two, and a short while later one morning members of my group kept dropping into my office, saying, "Hey Tim, have you heard? He's had a job offer. He's going to show it to you!" So by the afternoon I was prepared. After what I suspect was a well lubricated lunch, the chief operator came into my office and placed an envelope on my desk, saying only, "Der-dum!", an imitation drumbeat. I opened the letter and read the offer of a job he had received, a position for a chief operator of a substantially bigger and more modern computer centre in another company. "This looks like a really good career opportunity for you", I said. "Congratulations! I hope you'll be happy in your new job!" I reached across my desk and shook him by the hand. He looked a little stunned, murmured "I see you're not trying to persuade me to stay", and left the room. Had he really expected me to persuade him to stay after all the complaints, the meetings and warnings? I suspect not, not seriously. But he probably did not expect me to show such brutally honest relief at his departure.

Nonetheless, I think the British industrial relations rules about giving people clear unambiguous warnings is essentially fair. It is very easy for someone to have no idea that they are under-performing if a well-meaning manager is too polite or equivocates about letting them know the true perception of their work. With clear messages the individual has some opportunity to improve.

The activities of the BSCC began to shrink and some of its staff transferred to other departments in STC, most of them working on new telephony projects. I put Tani Haque in charge of the now reduced group. At the same time, a hardware research and development

group was put under my wing. I was wary about accepting this, because it was many years since I had any experience of electronic circuit design, not since I worked at Texas Instruments and Cambridge Instruments some eighteen years previously. I talked at some length to the section leader of this group, and he was confident that the principles of electronic design had not changed, only the technology supporting it. So I agreed to take it on, but I never truly felt I could understand their work thoroughly enough to make judgements about the wisdom of their development policies. I became more confirmed in a belief I have had for a long time, that you cannot properly manage an activity unless you have worked at it yourself.

I no longer reported to Gerry Jacob or the Computer Engineering Centre. Now I was part of the STL management hierarchy. After several further organisational changes, Frank Simpson became my Division Manager. The Microprocessor Research group, headed by David Wright, moved into my department. This group developed software to drive microprocessors, computers whose central processors were on a single integrated circuit or chip. Until then central processors, which carried out the extraction and execution of instructions stored in a computer's memory, consisted of a substantial amount of electronics, transistors, chips and other components, on one or more circuit boards. Printing the entire central processor on a single chip was a very recent development and STL, being a leading research laboratory, were keen to explore the potential of these new microprocessors. David Wright, an extraordinarily energetic character, led this research group. Its work was mainly software, intimately bound up with the hardware but also comprising operating system and basic utility functions. I was more than happy to have this group reporting to me. They talked the same language as I did. We had to make many policy decisions, because manufacturers of microprocessors began to proliferate. The company had to choose which manufacturer's products to concentrate its efforts on. TI, Intel and others were contenders. Microprocessors were beginning to be used in telephony. They would need support for the development of their application software, just like the 3200 had. David Wright's team had produced an autocode language for microprocessors, PLM. Other smaller languages for various purposes were in use. We had to set up ground rules and sort out questions about who uses these different languages, how can they be controlled, who should produce documents on style, usage and so on. Many small committees were set up to do these things and an STL language management group was set up. All this was a small reflection, a microcosm of what was happening in the industry at large.

More staff moved into the division and a software research team was formed, starting with Bernie Cohen. With all these groups, my department was beginning to become overweight, so to speak, and more reorganisations followed. Bernie and I were made Chief Research Engineers. There were just a few people with this title in STL. We were no longer line managers but were more like free floating gurus, with the same rank as Division Manager. I

got a company car as a result, and was attached to the software research group. I thought this was ideal for me. I did not relish the prospect of continuing up the STL management ladder, for the higher ranks concentrated too much on the financial and not enough on the technical side of the work for my taste.

Organisations and their management began to look upon computer programming as part of "engineering". Engineering products should not have faults, should work properly and be genuinely useful. A drive for "software quality" became widespread. Many of the problems with software arose because it was easy to build. It did not require components to be manufactured and assembled together. It "only" required a programmer sitting at a desk writing code, which was then punched onto paper tape or cards and fed into a computer. The result was that pieces of software could be built which were extremely large and complex. This inherent complexity was the source of errors. Furthermore, because of this great complexity, even exhaustive testing could fail to uncover some errors. One such error in a Fortran program caused the destruction of the NASA Mariner 1 mission to Venus in 1962. A transcription error substituted a comma for a full stop, causing the navigation software to miscalculate, causing the rocket to lose control. It had be destroyed over the Atlantic ocean. There is a long list of other disasters resulting from software engineering errors. Many programmers and managers flocked to resonating halls to listen to conferences on software quality. Programming languages, methods for developing software, and "architectures", ways of putting the large number of components of a piece of software together, were intensively discussed. The fact that massive programs consisted of many components, all with different versions and variations for slightly different customer requirements was a particular problem itself. It was called "configuration", and configuration control became a discipline, with rules and computer support tools. These rules and support tools would try to prevent mismatching versions and variants of components being put together.

The USA Department of Defense and their bigger contractors and associates such as TRW and the Jet Propulsion Laboratory were among the largest ever customers for software. They could afford to do extensive research studies into causes and sources of errors, for they had the most to lose from them and the most to gain from learning how to avoid them. They carried out extensive statistical studies into the sources of errors, what part of the life cycle produced the most, which were the most expensive to repair, and so on. Removing an error early in the development process saved a great deal of effort, and therefore money, down the line. One of the first practices adopted was "code reading". When a programmer had written a piece of code, another programmer had to read and understand it, looking for mistakes, before the code was submitted for testing. A typical proportion was 600 errors detected during code reading, 300 during unit testing of that individual piece of software, 200 when it was consolidated with other pieces of code into a system and finally 15 when the software was in service. The latter, 15, were the most costly to repair, for in effect the software had to

be recalled, like a car with a part found defective after many examples of the model had been delivered. The 600 found during code reading were the cheapest to repair, because not much work had to be redone and recombined.

Code reading was just one form of "peer review". Producing a piece of software was divided into a number of stages. One started with a statement of general requirements that the software had to meet. Then someone produced a specification, which described what the software would do, the functions it would carry out. Then came a design document, which described how the software was going to work, the layout of its internal and external data and the step by step processes, the algorithms, which it would use to perform the necessary calculations and manipulations of the data. Finally would come the code, the actual instructions which were fed to the computer and be automatically turned into a program by a compiler. The writer of one document would be the customer for the next one to be produced. So the writer of the requirements would be the customer for the specification, the writer and hence supplier of the specification would be the customer for the design document, and the supplier of the design document would be the customer for the code. This way of looking at the process was known as the contractual model. Each player in the process had a contract to fulfil, even if the same person was writing two or more successive documents. This simple model of the life cycle, with each document being a predecessor of the next in the chain, was one of the first models of the life cycle and was called the Waterfall Model. Many more were to follow. People soon realised that the waterfall model was too simple; it did not take into account the many revisions and backtracking that took place as mistakes and misconceptions came to light during the process. Often, even the requirements would be revised as the end customer realised that their needs were not quite as they had at first perceived. The "V" model was a variation of the waterfall model and showed the typical backtracking up the waterfall and down again. Numerous lifecycle models have subsequently been enthusiastically put forward by their protagonists, and are so to this day, Prototyping and Agile Computing being among the more recent ones.

Different styles of design reviews were advocated by different gurus. In a design review, which was on the whole a good practice borrowed from other engineering disciplines, the designer would explain the design to an audience of other designers, who would question and probe to in order to uncover any errors or shortcomings. There was an emphasis that this review was in no way a check on the competence of the designer and would have no effect on his or her pay or promotion prospects. For this reason the reviewers would be "peers" of the presenter. Design reviews were an example of "walkthroughs", where the writer of any of the documents in the life cycle would walk through it and explain it to an audience.

ITT had established user groups for many of its products. This was a practice being adopted by many engineering suppliers and manufacturers. The supplier would gather representatives of its customers together into a kind of club and consult them about the acceptability of its

products. Doing this was considered to be a commercial advantage, spiking complaints before they became too annoying and also being in control of any customer antipathy. But it was also another way of improving quality of the products, ensuring they were fit for purpose. STL set up a user group for the internal and external customers of the microprocessor group's software. This was one more mechanism typical in the industry for improving quality.

The studies done by the DoD and their contractors showed that more errors occurred during the design phase than during the coding phase. They also showed that using high level programming languages, as opposed to machine or assembly code, reduced the cost but not necessarily the error rate. The reductions in cost occurred in some of the later phases, including testing and maintenance. Some organisations found that it was difficult to motivate staff to do software maintenance work. This is not altogether surprising; experience in maintenance was not and is still not regarded as particularly valuable on someone's CV. The activity does not impart design experience and involves little creativity, even though ingenuity may often be required. Design has always been the more glamorous part of the process. Yet studies showed that maintenance costs formed 60% of the total. Some organisations seemed to employ large numbers of less qualified staff instead of smaller numbers of graduates. This seemed to have given rise to difficulties, both technical and organisational. Less qualified staff seemed to take much longer to learn a new programming language. It was as if they had to unlearn the language they previously learned and the whole relearning process would take several months.

The DoD used many different computer architectures, software development methods and programming languages. This increased the amount of effort they needed to maintain their systems, not least because many people had to relearn and adapt all the time. They decided to try and standardise on one new all purpose high level programming language. A document listing the requirements for this new language was published in 1975. It was called "Strawman", after the fairground game in which a straw man is set up and knocked down by throwing balls at it. The Strawman document was intended to be criticised and "knocked down", following which it would be replaced by a more resilient version, a "Woodenman". Strawman was published in 1975, and Woodenman and its successor Tinman in 1976. Proposals were then invited to design a language meeting the Tinman requirements. The new language would not be imposed on existing projects, but only introduced in new ones. The DoD believed that the reluctance of programmers to change languages would be a critical factor. The style of the language, quality of compilers and other tools, and user experiences would all be critical. Seventeen designs for the language were submitted and these were reduced to a short list of four. These four language designs were given code names, the colours Green proposed by CII Honeywell Bull, Blue proposed by Softech, Red by Intermetrics and Yellow by SRI International. The short list was reduced further to two, Red

and Green, in 1978. The contenders, Intermetrics and CII Honeywell Bull, were given a further year to refine their proposed language designs.

This initiative on the part of the DoD caused great excitement in the software community. The DoD was a huge procurer of software projects, some of them requiring enormous amounts of effort, like 300 person years, to complete. The DoD was very influential and the technical features of the new language was going to have a lasting effect on the shape of software and programming for some years to come. The proposal chosen was Green and the final language called Ada after Ada Augusta Lovelace, the assistant of Charles Babbage who in the nineteenth century invented the mechanical calculating machines now in the London Science Museum. Ada Lovelace is considered to be the world's first ever computer programmer, for she devised programs for Babbage's calculating engines. The requirements were revised one more time as Steelman and the final version of the Ada language produced in 1980. However, in 1978 the process of procuring and designing the language was not yet complete but well on its way.

With systems getting larger and larger, more difficulties came along, arising more from human limitations rather than technological ones. Software could often be written to meet the wrong requirements. Specifications were frequently open to interpretation by the programmers and analysts, who did not know the details of the total design. Top-down, hierarchical design could be the way to defeat the problems of size and complexity. The industry began to look for methods of stating requirements that were unambiguous and that supported top-down design. Projects needed management tools providing automatic analysis and giving information on progress and other features. But there were many contenders. In one software management conference in 1978 a speaker from the Royal Aircraft Establishment gave a list of 24 different software and system development methods. Project audits became popular for a time. An independent, specially trained, auditor would be requested to audit a project, examining it following specific guidelines, check-lists, procedures and standards for measurements. There were some apprehensions: the results could be misused and full time auditors could lose touch with the advancing technology they were supposed to assess. There was also a strong danger of equating progress with aspects of a project that were simply easy to measure, like lines of code, expenditure, or milestones that did not properly reflect a project's advancement. The measurement of progress needed to be based on the planned structure of the work, the amount of work remaining to be done and the cost to completion. Even today, some projects and organisations have yet to learn these lessons.

Software is useless if it does not work to a certain degree of correctness, and in some situations, like on board a spacecraft where it is remote from human intervention, it needs to function well-nigh perfectly. But other properties of software began to acquire importance. Examples were its portability, that is, whether it could be transported to new hardware or a

new physical type of computer without too much rewriting; maintainability, the ease with which it could be corrected and upgraded as requirements evolved – this could depend on many things, the documentation, the clarity of structure, and the simplicity of design; and usability or user friendliness, the phrase coined in later years.

ITT and the telephony industry in general shared the DoD interest in programming languages. In some ways, the available high level programming languages were unnecessarily general for many purposes. Specialised, "Problem Oriented" languages could have advantages. They would be smaller and hence easier to learn, and would avoid features that were not going to be used. People who were not programming specialists might be able to use them. On the other hand, use of problem oriented languages could lead to a proliferation of languages just at the time when most of the industry was try to reduce the number and investment in them. Along with other organisations which produce computer programs, ITT devised its own guide to programming style. The overriding criterion was visual clarity, but other motives were portability, the ability to transfer the program to other machines. The program should not make assumptions about the architecture of the machine, how many bits are in each word of store, for example. Procedures implementing related functions should be grouped together. Major data items of complex structure should be accessed by a few specific procedures grouped together, rather than spread throughout the program. This assists later modifications and extensions to the program, and foreshadowed much later ideas of object-orientation. New programming languages were being developed and becoming available. The older languages had a fixed collection of data types: variables could be of a limited choice of types like integers, character strings, Boolean or truth values (True or False), arrays of these and so on. More modern languages gave the programmer a much greater, unlimited choice, and there were consequent advantages. Arrays, that is linear or multi-dimensional collections of values of a simpler type, could be dynamic, that is their size need not be determined in advance. Different types of data could be grouped together ad lib into records. All these facilities made life easier for the programmer and could enable clearer programming, more closely related to the concepts of the application. Researchers drew out some principles of language design which made programming languages easier to learn and use. The earlier high level languages were designed as convenient ways of driving the machine. As time went on, the impetus was to design languages that expressed application problems well. Hand in hand with this, computer architectures were considered that would facilitate the implementation of advanced languages, but less progress has been made on this front over the years.

One of the most advanced programming languages of the time, Algol68, was coming up for its tenth anniversary. I went to a conference on Algol68 in 1978, but it was becoming clear that the language was falling out of use, for no very obvious reason. ICL still used an in-house subset, S3, for its systems programming, but there were few users outside academia. I was a little disappointed about this; I liked the language, but maybe its lack of popularity was

due to some difficulty in learning and implementing it. On the other hand, Ada, yet to be finally defined, was to be much more difficult both to learn and to implement. In fact, the acceptance and popularity of languages and methods over the years seems to have been almost arbitrary at times.

So Algol68 was not for the telecommunications industry. The CCITT is the international standardisation authority for telecommunications. It is an influential and respected organisation, probably because, without strict standardisation, telephone calls and other communications between different countries and telephone systems would be impossible. The CCITT observed the DoD initiative to standardise on a single high level language, with its series of requirements documents Strawman, Woodenman, Tinman, Ironman and so on, and decided that they should conduct a similar study themselves. CCITT set up a high level language committee to choose or design a language for telecommunications. The committee considered simply using Ada, the forthcoming DoD language, but decided this was a little too elaborate for telecoms purposes. So the committee designed a new high level telecommunications programming language, CHILL. By mid-1978 the definition was in an advanced state and the US research laboratory of ITT prepared to write a compiler for it. There was even discussed the possibility of my division at STL being involved in writing the compiler. A growing number of telecommunications organisations started to want CHILL compilers. STC in the UK wanted one for developing code for the 8086 microprocessor. Philips expected to produce a compiler by October 1978. Donn Combelic arranged for me to join the CHILL implementers' forum, a CCITT committee. This committee was next meeting in London in December 1978, and then scheduled to meet every quarter, February in Rome, June in Geneva, and September in Melbourne. At the London meeting the different participants shared brief news about their progress in producing compilers, putting on courses and so on. Somewhat to my surprise, no-one discussed compiler techniques or difficulties with implementing any features of the language. However, the act of implementing compilers for CHILL revealed ambiguities and some incompleteness in the language definition, so most of the discussions centred on resolving and agreeing details of the definition. At least this made sure that the compilers produced by different organisations were consistent.

In the computer industry at large, people were beginning to give some thought to methods of designing software, the stage in the development process that would naturally precede the writing of the programs, as well as to the languages in which they wrote the programs. The earliest form of design was a flow diagram. These were not conducive to well structured programs, as they allowed unrestrained use of jumps, or transfers of control. For ten years now, since 1968, this was recognised as poor practice and so the next step in design techniques was to devise a form of flow diagram that would tend to well structured programming. The most well known form of these structured flow diagrams are Nassi-Shneiderman diagrams, but these did not gain great popularity, although they still have their

devotees, are the subject of a standard (DIN 66261) and have a body of support tools. The main problem with them was that they did not lend themselves to abstraction and the other principle of well structured programs, separation of concerns.

At the beginning of 1978 at STL we too began to think more seriously about methods – the ways we developed software and systems, looking at several pieces of academic and not so academic research. For that reason, our efforts could perhaps genuinely be called "methodology". Was a development method the same thing as devising a language in which to express different steps of the life cycle, requirements, specifications and designs, languages which could perhaps be processed by computer in the same way that programming languages are compiled by computer? Much of our discussions conflated the concepts within methods with the languages they used. We made a case for internal ITT funding research into SW, "investigating techniques which will move the programmer's task nearer to a formulation of what the computer is required to do, rather than how it is to do it. Each step in this direction will reduce the costly, error prone activity of programmers reinventing ways of realising specifications of problems. This overall objective is to be attacked on three fronts: specification, program design and program implementation". A traditional design technique used in the telecommunications industry was Finite State Machines and State Transition Diagrams, loosely based on a theoretical computer science concept called Finite Automata. Other techniques were being developed in academia, Petri Nets, a diagrammatic notation with a strict mathematical definition invented by the computer scientist Carl Petri, and other approaches based on mathematical logic. There were also some more pragmatic approaches being developed elsewhere, PSL/PSA, Gamma, and SADT, "Structured Analysis and Design Technique". PSL/PSA, the Problem Statement Language – Problem Statement Analyser, had been started at the University of Michigan under Professor Daniel Teichrow as early as 1968, and continued to be developed until the early eighties. It was developed with the help of many industrial sponsors. The Problem Statement Language was used to define a system's requirements, using techniques from relational databases. The Problem Statement Analyser consisted of a set of tools for generating reports and checking the integrity of the database created from the PSL description. PSL/PSA is still alive and well today, with its enthusiastic devotees, a web site and a series of conferences. We invited the proponents of a further development of SADT, SAFP/2, to come and give us some extensive presentations. Several ITT companies were to take part in this software research activity. In 1978 we held the first of several seminars, to which we invited known people from other companies as well as software programmers and managers from ITT companies. There was a tacit, almost subversive, conspiracy amongst the researchers in different, even rival, companies to share our technical ideas with each other, as a way of helping to persuade all our managements to encourage and fund the activities we believed were worth advancing. We could then use a subtext when talking to our managers, "look, company X is investigating technique Y, we'd

better not lag behind". Participation in international standards committees, research conferences and professional bodies like BCS special interest groups was a great help in promoting this unofficial collaboration. In writing and presenting the case for our software research activity we included a "scenario", like a story, of how software might be developed in the future (the future then being the 1980s). This use of scenarios presaged the European Community initiative Framework Six, twenty five years later, which used a number of scenarios of computers in everyday life to give a direction to the aims of current research in IT. One can then list the kind of technologies needed to make the scenarios possible. Our software research case was to be part of a coordinated collection of activities from different ITT companies, so we had quite a few meetings with people from ITTLS in Madrid, SEL in Munich and others.

We began to concentrate on development methods even more. SADT and its derivatives seemed to prompt people into dashing off to do the design of a system before working out fully what the specification was, what the system was intended to do. Much of telecommunications software involved parallel processing, using a computer to service demands and data input that was arriving in parallel from different sources. Analysing and designing parallel or concurrent systems was a much less well understood art. Petri Nets could handle concurrency, and so could CSP, Communicating Sequential Processes, a specification formalism devised by Professor C. A. R. Hoare, then at Queen's University, Belfast[12]. CSP has since thrived, is the basis for the concurrent programming language OCCAM, celebrated its twenty-fifth anniversary with a conference in 2004[13] and continues to stimulate research and development.

I went to a conference on "Fuzzy Reasoning". It explored the relationship between mathematics, linguistics and computer applications. The classical syllogism:

> All men are mortal.
>
> Socrates is a man.
>
> Therefore Socrates is mortal.

uses the deductive rules of classical logic. One can contrast it with fuzzy reasoning:

> Healthy men live long.
>
> Socrates is very healthy.
>
> Therefore Socrates will live a very long time.

The terms "long", "healthy", "very" are not cut and dried. The truth of statements like "Socrates is very healthy" is not absolutely decidable or determinable, but can be slightly true, very true, debatable, etc. Different rules of deduction have to be used for these "fuzzy"

[12]See Hoare 1978.
[13]See Abdallah et al., 2005.

statements. These new rules of logic were called “fuzzy reasoning”. Fuzzy reasoning could be used in certain kinds of control systems in chemical plants for example, where temperature readings could start getting “a bit hot” and so on. The conference suggested that there were applications in system modelling, artificial intelligence, medical diagnosis, social sciences and medical information systems. This was a conference full of interesting ideas, but I did not think they were particularly applicable to telecommunications[14].

The National Computing Centre was already over ten years old, having been founded by the UK government in June 1966. Its aim was to encourage the growth of computer usage in the UK, simplify the work involved in using computers and ensure that the necessary education and training were available. Since its foundation it has produced copious useful booklets, guides and studies. In 1978 the NCC was planning a booklet on program design methods. We tried to be as up to date as possible in program development methodology by meeting with many organisations, the NCC, the SRC – Science Research Council, now the EPSRC, Engineering and Physical Sciences Research Council – Software Sciences, the Digital Equipment Corporation and others. You could say we were thirsty for knowledge and understanding.

We invited two members of the SRC to a meeting and told them about our research programme, both in software and computer architectures. Another group at STL was investigating novel microprocessor based architectures. We also visited Software Sciences in Macclesfield and heard about their Gamma method. They had been tutoring the data processing department of Barclays Bank in its use. Barclays bank had a team of eighty programmers who were programming the first cash point or ATM facility. Cash points were in their infancy but coming into use as the nineteen eighties approached. We decided to provide some funds to assist the development of Gamma, along with Barclays Bank, and made a case for ITT funding planned cover three years from 1978 to 1980. Cases for ITT funding had to be made for each calendar year afresh, which hampered the planning of longer term projects.

Digital electronic technology was also advancing. Software resided in new computers constructed from this new, miniaturised technology. Because computers were becoming smaller and cheaper, it was possible to have several intercommunicating with each other. With the advent of microprocessors, this was even more the case. We were, indeed, starting a collaborative project on a new microprocessor with the Intel Corporation. This computer was expected to be ready to take new systems software on board in the early eighties. We had to restrict information on the project to protect the commercial security of Intel and we were all charged not to talk about it. The fewer people who knew about it the better! To enable

[14]See Mamdani & Gaines, 1981.

computers to communicate with each other, specific communication software has to be written. The ways of doing this are far more problematic than with simple, linear programs that sit inside one computer and are executed one instruction at a time. Protocols are needed to make sure that information is sent and received at the right time, to ensure proper synchronisation and to resolve possible conflicts if more than one computing process is trying to communicate with another. "Semaphores" were one of the first techniques for doing this and were invented by Edsger Dijkstra[15]. They were a technique for ensuring that critical sections of code could not be interrupted, and also provided a mechanism for forcing different processes to wait and allowing others to resume. Another computer scientist from the Netherlands, Per Brinch Hansen, devised a method called Monitors, which could be implemented using semaphores. Monitors could ensure mutual exclusion, so that for example two different processes could not work on the same piece of data at the same time, with resulting confusion. With an airline booking system, you can imagine that two or more travel agents might be trying to allocate the same seat to two or more customers at the same time. Used judiciously, monitors could prevent this. Monitors were incorporated into some programming languages that were specially designed for concurrent programming. Modula-2, based on the earlier language Pascal, was a notable one of these. But these were low level mechanisms, programming techniques, that had been around for some ten years. The theoretical understanding which would lead to more general rules and specification languages for parallel programming were just beginning. Petri Nets[16] and Hoare's Communicating Sequential Processes[17] were two such theories that had recently been published, and were soon to be followed by a "Calculus of Communicating Systems" by Robin Milner at Edinburgh University's Department of Computer Science[18].

But the advancement of computer hardware did not just affect the nature of the programs we had to write. They could also affect our working environment itself. This book has been prepared on a word processor, that is an application program on a personal computer. Until the late seventies, ordinary typescripts were not held on computer files but were prepared on typewriters, usually by typists and secretaries. In 1978 the first word processors began to arrive. These were purpose-designed computers, used for nothing else, a desk-top substitute for a typewriter. There was a lot of resistance from typists and secretaries to using these. A firm came to visit us and demonstrated a word processing product called Wordwright. Typists and secretaries did not want to learn "how to use a computer". Our organisation did not adopt this new way of working just yet, but some others did, to advantage. We now take it for granted that any document can be altered, paragraphs inserted and moved about, with minimum effort. For a few more years we stayed with the less efficient typescript which

[15]See Dijkstra 1968.
[16]See Carl Petri 1973 and 1980.
[17]See Hoare 1978.
[18]See Milner 1980.

required retyping for major revisions or messy, literal “cut and paste” jobs with scissors and glue.

DEC, the Digital Equipment Corporation, manufactured a successful minicomputer, the PDP11. They gave us a presentation on a new extended version being developed at the time, the VAX11/780. Minicomputers were much smaller than the big mainframes like the IBM 360 and 370, or the old English Electric KDF9 or CDC or Burroughs machines, which occupied the whole of a large room. Minicomputers would take up a couple of racks, tall bookcase sized metal frames holding layers of circuit boards. So they were bigger than personal computers, which were based on microprocessors and were designed for individual use. The VAX series was to become successful and to have a substantial product lifetime. The VAX11/780 was to have quite a number of innovative ideas. It would have a virtual memory, that is a two or more level store that the operating system would present as a single large storage area, which I first encountered on Atlas and which is present in every present day personal computer. It would have a cache memory holding the last or most likely used 128 memory contents, as do today’s computers. Its instruction set was to be oriented to high level languages, having loop and case instructions (case statements are multiple conditional branch instructions). But I noted that there was no direct support for things like parameter passing or the more complex data types. I also had my doubts about whether a compiler could make use of these special loop and case instructions in practice. Every high level language has subtle differences that can easily be incompatible with the simple interpretation of such facilities that a machine’s architecture might offer. The VAX11/780 operating system would allow concurrent, multiprogramming, interactive and batch work. Its memory had error correcting codes that would correct all single bit and detect all double bit errors. Altogether, fairly impressive and full of good ideas for the time. I think it gave our microprocessor group much food for thought.

Alongside all this assessment of new research into computer technology and trying to push it forward, some of our day-to-day operational problems still persisted. They did not want to go away. Jacques Newey kept producing document after document giving details of his proposal for transporting the ITT 3200 software development platform from the 3200 to the IBM 370. This would have been by far the better, more efficient way to develop systems software for the 3200, something I had recognised soon after I started at STL some six years earlier. When I look back on it, part of me wants to say “At last!”, and part of me marvels at Jacques’ persistence in the face of management inertia. He didn’t take “no” for an answer and after being steadily bombarded with his ever more detailed documents and papers, describing proposed conversion utilities, cross-assemblers and other features, management began to believe that it was actually happening. The result was – it did. By September 1978 SDSS, the name given to the new cross-development system, was beginning to be used, particularly by

the Spanish ITT company SESA and their research laboratory ITTLS. But simultaneously DPSS, the replacement operational system for the BSCC and LCT software, continued to be developed and was well in hand by 1978. The main user seemed to be BTMC in Belgium. They were providing the most copious feedback to the development team and continued to test and validate it throughout that year. I mused to myself that the original concern over DPSS had been that instead of superseding its two previous rival development systems for 3200 software, the BSCC and LCT systems, DPSS might simply become a third. Now, with SDSS, we had four contending systems! Professor John Buxton's original recommendations had been two-part. A recommendation of principle, that development should be incremental, not big bang, and a pragmatic one, and secondly, that since we had started DPSS we should finish it. While appearing to accept his advice, the company did not seem able to follow it, so as a consultant he was quite safe. No-one disagreed with his advice, and no-one could hold him to account since they did not do what he said!

However, the two pieces of development, DPSS and SDSS, had another beneficial effect. Programmers and managers from various teams in Europe all took some part in them, and previous rivals were often working together on technical policy committees to reach agreement over the schedules, timescales, resources and technical details of these projects. The old rivalries were sinking into the mists of the past. I thought there was a moral here: don't ever make too hostile an enemy, for you may find yourself working alongside them a couple of years later.

STL, being a research laboratory, frequently filed patents. The lab had a patents department, whose rôle was to draft patent applications and guide them through the filing process. Every department was urged to file patents. Filing patents was seen as a measure of success. But the question of whether one can patent computer software has always been uncertain. Only "products" can be patented, but these can include processes, for example, like those found in the chemical industry. A pure computer program is a document and an algorithm, a step by step process for performing a calculation or driving a piece of apparatus. It might be possible to copyright a program, treating it as a document. But that would not be particularly useful, for a program can easily be radically changed in form while keeping its design intact. A specific physical device containing a computer and some software could be patented, for that would be a product. These days the attitude to patenting software has relaxed a bit, especially for software that has a particularly recognisable user interface. As part of the drive within the company to accumulate patents, we were urged to keep laboratory notebooks, dating all entries so that evidence would be to hand in case of disputes over the origin of ideas. Most of us followed this advice, although we never produced software patents. Since then I have noticed that workers in many research and academic establishments keep their day to day notes in day books, instead of on pieces of paper or in folders. It does add some weight in an argument if you can say to someone, "On 30th July at 2.45 p.m. we agreed that…". And

without my notebooks I would not have been able to recall the flavour and details of all this work!

With the hardware technology changing under our feet all the time, ITT and STC continually needed to train and retrain their staff. Not only did existing programmers need to learn about the new languages and methods of design, but there was a general shift towards more use of computers in telephony. Electromechanical and electronic apparatus were slowly being replaced by computer-driven technology. This meant that fewer hardware engineers and more software engineers were needed. So hardware engineers had to retrain as software engineers. Not only do old dogs not learn new tricks, but even relatively young hardware and software engineers were often reluctant to retrain. Engineers may have established a reputation for themselves, having expertise built on several years experience in technique X. If they now have to learn technique Y, they are on the same level as younger staff, paid less, who are a few years behind them. The seasoned engineers would then feel insecure. We had to work hard to allay these feelings.

Part of the rôle of the research groups was to watch the future, to keep an eye on the way hardware and software were advancing so that the company could keep abreast or, preferably, ahead of its rivals. To do that the development staff had to be prepared for new techniques too, so the research groups were frequently making recommendations about training, designing curricula and devising courses. We worked with several other institutions to discuss and jointly create courses. Lancaster University were, like us, interested in producing a course in the programming language Pascal. It had overtaken Algol60 in popularity and was regarded as being more practical. We also worked with The University of Essex, who were not far from us geographically. They laid on custom designed courses for us, which were presented like fairly prestigious industrial events, with a formal dinner and various goodies for the “delegates”, rather than “students”. The government had started to encourage and require universities to work with industry, and so there was a certain eagerness in academia to get together with us as soon as we said the word. Harold Wilson’s government coined the phrase “UK plc”, and if universities worked with industry, they were justifying their existences more strongly. We also had discussions about course construction with InfoTech, a software training and consultancy company that is still thriving. However, we also had to work quite hard to persuade our own management to let us go along this path. STL was a research laboratory of the company and STL management had a slightly ivory tower view of their own establishment. For STL’s researchers to get involved in training was, to them, rather undignified, a waste of intellectual talent. One or two of our proposed training projects were actually stopped by management intervention, at one time by the managing director, who happened to see a set of course notes of mine in the photocopying room, ready for

duplication and distribution. He cancelled the duplication order (without letting me know). Training is not STL's rôle, he said.

Prophets are rarely recognised in their own country. A research group in a large company can be greeted with cynicism and suspicion by management and other development groups. Recognition outside the company can bring about greater internal respect for the research group. It was to our advantage to make ourselves known to other bodies, and we made a few approaches to enter contracts with other firms. Of course, they could not be direct rivals, but software was used in a great variety of applications, many of them not pursued by STC or ITT. We met Hawker Siddeley Dynamics in Stevenage, who were bidding for a contract with the European Space Agency. The contract was to write the on-board control software for an unmanned space vehicle.

Hawker Siddeley gave us a rundown on the potential contract. The ESA required very restrictive Quality Assurance measures, which led to their tending to insist on methods and procedures of development that were not up to date in their thinking. For example, they required a development process of design, code, test and recode. We favoured performing verification while implementing, which meant that the development preserved correctness whilst it was in progress. The more old fashioned code–and–test process meant that one was always trying to remove errors after they were made. It is well nigh impossible to catch them all. We believed our approach was much more reliable, and therefore more suitable for critical applications. Once a space craft has been launched, repairing the on board software is very difficult, if not impossible. In 1978 the ESA also required a great deal of information from its contractors: the CV of everyone engaged on the contract, corporate details and track record of the company, corporate experience as subcontractors (of which we had little), the organisational structure of the company. The development of the on board software was to be hosted on a Modular One, a minicomputer manufactured by a British company, Modular Technology Ltd. Hawker Siddeley told us how a proposal for a similar study ran to 50 pages including 25 on the planning and scheduling of the work and 23 on the technical content.

The ESA were opposed to using high level languages for on board software because of variations between compilers and of occasional faults, which were outside their and their subcontractors' control. There is a real dilemma here. If one uses a low level language, one is effectively translating from the low level design to the code by hand, which is much more error prone and subject to variations in design approach. But the ESA were correct in their assessment, at the time, of variations and lack of reliability of commercially available compilers.

The ESA also had a low limit on the labour rate they were willing to pay for software development work. Hawker Siddeley Dynamics admitted they were hard pressed to meet

these labour rates and found it only worthwhile bidding for these contracts if they had staff who would otherwise not be engaged on any fee earning work. This was not really the case for us. After a lot of discussion within our division I proposed that, regretfully, we should withdraw from the bid. I thought it was important that we should tell the ESA the reason. I believed that they were not ensuring the best quality, which they surely needed for such critical work, if they adopted this financially strapped policy. I telephoned their representative in Amsterdam and made this point to him. His only reply was:- "Well, we usually get several bids". I believe that, since those days, the ESA have moved on a long way, and use software technology that is state of the art.

To increase our exposure to the technical world outside, we tried to publish academic papers and attended an increasing number of conferences. It all helped to make ourselves known elsewhere and in turn improved the regard in which our own management held us. IFIP, the International Federation of Information Processing had a number of specialist groups and a prestigious periodic conference. The Computer Journal of the British Computer Society and various publications of the ACM, the US Association of Computing machinery, were all outlets for publications. These multiplied rapidly as the years went by until one day I realised that the department was receiving thirty seven journals. No-one had time to read many of them, and so we pruned the selection.

This account may suggest that my time was filled with excitement and innovation. In fact, I have found that every job consists of boring activities for at least 50% and, with luck, 50% of interest and engagement. Much of my time was taken up with necessary but pedestrian tasks of administration and non-technical meetings, planning and negotiating office space, budgets, furniture, purchase of equipment such as mains voltage stabilisers, and comparing their sources of supply. The more senior of us spent much time discovering what other technically respected organisations, including competitors like Bell Labs, were doing, the programming languages and development policies they were pursuing. We needed to keep up with the best in the field, but also to try and predict in what directions various aspects of technology were likely to advance, so that we did not waste time and resources by going the wrong way.

A huge amount of my time was spent at this period on planning and estimating budgets for possible projects and activities, most of which we knew would never happen, but were just ideas up for discussion. For example, how much would it cost to develop a "computerised document control system" and train everyone to use it? ITT was very proud of its quality control system. They set up small groups of people called "product control centres" and gave them responsibility for controlling the quality of a set of products. The BSCC was one of these. An ITT standard prescribed how these centres should work, producing documents such as a product register, a register of sites which produced the products, and change proposal register, a change note register and so on, and so on. There were rules about what documents could or should be written when, and the whole system cried out to be implemented on a

computer. We made an outline design of a computerised system and made estimates for the cost of building it and bringing it into service, and did the same for several other proposals which never came into being. At the same time I was still involved in the nitty-gritty issues of the 3200 computer centre, configuring new peripherals to be shared between several machines, the repair of sticky hammers on the line printer, maintenance contracts with suppliers etc.

The seventies were coming to an end and the eighties approached. We continued to keep track of the latest advances and at the same time to spread the principles that we had learned to the development groups within our own company. Other research laboratories in other companies, the GEC Hurst Laboratory and Plessey Laboratories, were doing likewise, pursuing their own research agendas. But we all communicated, despite being ostensibly rival organisations. There was an almost subversive agreement amongst us, that sharing our knowledge would speed up progress in the industry and accelerate technological advance. So I believe that our experiences were fairly typical of the computer and telecommunications industry, and not just in the UK.

We held seminars where we presented some of the latest thinking, applying it to practical telecommunications software design, and inviting occasional speakers from outside. We tried to make these attractive to visiting names in academic computer science, both to promote our own reputation among them and to attract interest in the seminars. We invited academics and known researchers in other companies like ICL, BT, Plessey and GEC to the seminars. STL and STC had rules about company confidentiality. One had to go through elaborate and time-consuming procedures to obtain permission to publish papers and articles. The company had a certain paranoia about giving away our “secrets” to rivals. But, probably through omission rather than deliberate policy, there were no such strictures on seminars held on company premises, even when outsiders were invited. When publishing papers I had, in any case, hit upon the tactic of writing on the authorisation form “This paper is mathematical in nature and hence has no commercial value”. I was surprised at how well this worked.

One seminar went down particularly well and several external invited visitors came along to it. We decided to hold the event in a bigger, better accommodated lecture theatre in another STC building in Harlow. There was a local canteen in the building, and also a visitors’ dining room. My manager decided that it would be too expensive to provide a visitor’s lunch for everyone attending, so he authorised only enough to accommodate the external visitors, speakers and organisers, who were so to speak the hosts. On the day he decided to come to the seminar himself, and come lunchtime, he realised that he had excluded himself from the visitor’s lunch by his earlier ruling. He and I were talking together in the mêlée as it divided, the guests, speakers and organisers to the left and hoi polloi to the right. He said rather mournfully to me, “Oh well, I suppose I’ll have to try the canteen”. It was on the tip of my tongue to say I was sure there was room for an extra one on the visitors’ dining room, but I

held back. I felt that to do so would have been sycophantic. But it was an embarrassing moment.

While I had been much influenced by Dijkstra's seminal *Discipline of Programming*[19] and his pre- and post-conditions approach, he had not shown how this could be applied to the programming technique of decision tables. Decision tables use tables of conditions to determine which of a choice of actions to take, and do not fall into the usual ambit of programming structures prescribed by the structured programming methods advocated by Dijkstra. Decision tables were used a lot in telecommunications programming because of the many features and choice of actions that occur in telephony. Imagine the recorded instructions you hear when listening to a recorded voice when dialling many numbers these days. The system takes different actions depending on whether you press or say "1", "2", etc. In one seminar I showed how Dikstra's pre- and post-conditions could be extended to include well-structured decision table techniques.

Mike Gifkins at STC IDEC proposed producing one-page sheets, called "Software Technology Summaries" for internal consumption. We liked the idea of this and with Mike produced several over the next couple of years. The first was on state machines and grammars: The simplest kind of computing device consists of a machine that has a finite number of states and moves from one to another depending on the input it receives. In the 1930's, before any practical computers had been built, Alan Turing showed that any calculation that a computer could perform could be performed by a finite state machine (his was a very particular type using an indefinitely long tape, but other researchers showed that this was equivalent to other kinds of state machine). In fact, many of the earlier telephony designs were based on state machines, mainly because the early telephone exchanges were electromechanical and could best be described in this way. Other computer scientists such as John Backus had showed an equivalence between state machines and grammars, expressed in the form defined by Chomsky in the 1950's. It was traditional amongst telephony design engineers to define the simple telephony protocols using state machines. A protocol is really just a simple language. I was always puzzled by the fact that the telephony engineers used state machines to define a language, when it would have been much more natural, to my mind, to use an equivalent grammar. It was as if they were defining the shape of a key by describing the lock that it would open. So one of the first Software Technology Summaries, STS's, was on state machines and grammars. We produced more on the Contractual Methodology, the approach to system design mentioned earlier in this chapter, and other topics.

[19] See Dijkstra 1976

Chapter 8 The Search for Formality

One of the turning points in my whole career was in January 1979. Don Combelic urged Bernie Cohen and me to attend a Winter School on Abstract Software Specifications, organised by Professor Dines Bjørner of the Technical University of Denmark. Don had met Dines Bjørner on the CCITT CHILL committee and had been impressed. This was a two week event in the depth of a Danish winter in Copenhagen. I went prepared for the cold, taking my walking boots which I wore to travel between my hotel and the university, a bus ride and trek through snow every morning and evening. The river in the centre of Copenhagen was frozen. Dines had gathered together most of the big academic names in what became known as formal semantics, the mathematical modelling of the semantics or meaning of programming languages. We had challenging and mentally exhilarating lectures from Cliff Jones, Steve Zilles, Joe Stoy, David Park, Peter Lucas, Gordon Plotkin, Peter Mosses, Ole-Jan Dahl, Barbara Liskov, Peter Lauer, Rod Burstall and Dines Bjørner himself. The lectures showed how mathematics could be used not only to model the syntax of computer languages, which the earlier work on grammars and automata had done. Set theory and logic, normally regarded as part of pure mathematics, could be used to define the semantics of a language. Mathematical theories could be put together to build specifications of what a program was designed to do, using the same ideas from set theory and logic. If you have the mathematical tools to define the meaning of a program, then you can write this meaning down before you have written the program itself. That way you have a precise statement of what you want the program to do. This is a functional specification, more usually just called a specification, of the program. Mathematical logic can then be used to show that the program satisfies the specification. Some advanced mathematics was needed to model certain aspects of programming languages, topics such as domain theory and category theory that I had not come across, even in my maths degree course. All this extended and generalised the work of Dijkstra that had so engaged me to date.

Several of the lecturers, Dines Bjørner, Cliff Jones, Peter Lucas, had worked for the IBM laboratories in Vienna and had produced a formal semantics definition of the PL/1 programming language, which IBM used extensively. To do this they produced a special "meta-language" called VDL, the Vienna Definition Language. A variation and development of this could be used to specify programs, that is to write their specifications. Cliff Jones and Dines Bjørner together devised a way of developing programs in which one wrote the specification, and produced the program proving that it satisfied the specification as one wrote it, using mathematical logic. This approach was called the Vienna Development Method, and the specification language, based on VDL, came in due course to be known as VDM-SL[1].

[1] See ISO/IEC 13817-1, 1996

We had a free weekend in the middle of this course. On, I think, the Saturday, Bernie and I sat in his hotel room discussing how we might spend the time. I had seen people skating on an ice rink in the town and I had wondered about doing that. While we were deciding, Bernie produced a bottle of duty free vodka he had bought on the way out. We finished the vodka and didn't go skating.

I think the Copenhagen Winter School stimulated both of us to go to more events and try to learn more. In January 1980 I attended the annual POPL – Principles of Programming Languages – conference in Las Vegas. This may seem an odd choice of venue for an intellectual conference on computing, but I learned that it was off season in Las Vegas and the hotels offered extremely advantageous rates for conferences. There were 130 papers submitted to this conference and 25 were accepted. The criterion for selecting them was not to be survey or tutorial material, but original work. The most interesting from my point of view were papers by Leslie Lamport and Amir Pnueli. Lamport's paper was on modal logic, that is logic that takes time into account by using symbolised representations of 'before', 'after', 'never', 'sometime' and so on. This is useful for reasoning about concurrent programs in which one is interested in continuous behaviour rather than what is true before and after execution. Examples might be operating systems or control processes which may in principle continue indefinitely. He also talked about modelling non-deterministic programs which might depend on indeterminate external events or timings. Pnueli talked about another temporal logic system using a different notation and presented an axiomatic rules of reasoning about systems. I thought this might well be relevant for telecommunication systems where there is a lot of parallel activities and unpredictable traffic properties.

Las Vegas was like nowhere else I have ever experienced. I stayed in the same hotel as the conference venue and on entering I could not find the reception for some time. The whole of the entrance foyer was a casino. One had to walk through it to find the hotel reception hidden at the back. There were fruit machines everywhere, on the walls lining the baggage carousels in the airport and even in the bus shelters at the side of the road. I determined to spend my time in Las Vegas without gambling either on machines or tables.

This was my first ever visit to the USA. To get a bit more value for money out of the trip I had arranged to visit the ITT location in Des Plaines, which is near Chicago, after the POPL conference. In January in Las Vegas the temperature was a pleasant 70°F, 21°C. People were walking around outside in their shirtsleeves. In Chicago, on the shore of Lake Michigan, it was -4°F, -20°C, the coldest I have ever experienced. The change in temperature in the course of a short flight was dramatic. I stayed with a colleague who I knew from the CHILL committee who lived on the outskirts of Chicago and worked for Bell Laboratories. On an afternoon off we both visited the Chicago museum of modern art, a veritable treasure house and remarkably free of visitors for such a fine collection. There was not much to discuss with my colleagues in ITT Des Plaines, although they were interested in our Software Technology

Summaries and asked to be on the mailing list. I was a little shocked by the physical state of their premises. Their canteen had no crockery, only disposable plates and paper cups, which littered the tables and the floor.

In March of 1980 we invited Jean-Raymond Abrial to give us a seminar on his abstract language, Z[2]. Abrial was an independent French researcher who had developed an approach to modelling computer programs that was in principle very similar to VDM. He had recently joined the Programming Research Group at Oxford University, and this was to be the beginning of an enjoyable collaboration between us and the PRG. Z and the VDM language both used set theory and logic, topics in pure mathematics, to model the actions of a computer program. The syntax and appearance of the two languages were different, but there were more important differences in approach. Z used traditional mathematical functions to model programming functions and procedures, whereas VDM used the functions of domain theory. To be more accurate, VDM used a type of reflexive domain developed by the computer scientist Dana Scott to model programming functions. VDM had evolved from the endeavour to define the semantics of programming languages. That meant that it had to be able to model any construct that one could write within a typical high level language, regardless of how likely anyone was to do so. In sophisticated high level languages one may define a data type recursively. In particular, a data type could include functions from that same data type to another. However, in the late nineteenth century the mathematician Georg Cantor proved that no mathematical set can include its own function space. So, in theory, it is impossible to represent all possible computer data types as traditional mathematical sets.

VDM overcomes this problem by using reflexive domains, and in particular Scott domains[3], which are an elaboration of sets, to model data. Domains, unlike sets, can include their own function space, because only computable functions are included. Computable functions, the kind that can be programmed on a computer, are finitary; that is they can be defined with a finite amount of information. If one restricts the function space to computable functions, then a domain can indeed include its own function space. Dana Scott had worked on this problem from the early seventies[4] and spent important periods with the PRG at Oxford University.

Z uses traditional set theory to model data types. However, Z was devised to model programs, rather than to define the semantics of programming languages, and so one could argue that it does not need to use domains as its foundation. But, debatably, it might not be able to model programs containing certain kinds of recursive data types.

Abrial's seminar was the beginning of a collaboration between our software research group at STL and the Oxford PRG. The PRG were keen to demonstrate Z by applying it to a real life programming project and to transfer the technique to ITT. The U.K. government were urging

[2] See ISO/IEC 13568, 2002.
[3] See Scott 1980.
[4] See Scott 1971.

academic departments to demonstrate the practical value of their research work by applying it to industrial problems and working collaboratively with industry. So we looked for a project to act as a test-bed for Z. We thought it would be most unlikely to find a commercial telecommunications project willing to take part in an experiment of this kind, so we sought a non-critical task that would nonetheless be useful for the firm and that could serve as a demonstrator.

STL, one of the leading research laboratories of ITT, had some 3,000 visitors every year. If one expected a visitor, one had to fill in a form, and send it to Gladys, the visitors' administrator. Gladys would generate more forms to be sent to the security at reception, another to the visitors' dining room if one had ordered lunch, book hotel accommodation, send forms to the canteen for coffee if required, initiate the production of a visitor's lapel badge, and several other things. This was quite an involved administrative process, all done manually and on a fairly large scale. If the process was computerised it would be easier to amend arrangements, to trace progress and to avoid the hiccups which sometimes inevitably occurred. We invited Carrol Morgan and Bernard Sufrin from the PRG for a day and put forward to them the idea that they could specify this system in Z and develop an implementation on a desk-top computer. When a piece of software is developed, one starts with a requirement, which can only be accurately determined by discussing it with the customers, those who are going to use the software. From the requirement, which, if everything is done properly, should be written as a document that forms part of the project development history, the principal items of data and functions to be performed on them can be determined, and then the specification can be written. The specification is taken back to the customer and its details played back to them to see if it truly reflects their needs. In this case the customer was Gladys, the visitors' administrator. The statement of requirements and the specification would typically go through several iterations before all parties were satisfied with the results. Bernie Cohen and I were keen that the people from PRG should carry out all this part of the process in order to reach the specification in Z. It represented the proper way of doing things according to the latest thinking in good quality software engineering, and we were particularly interested to see whether two academic researchers could successfully replay the implications of a specification written in a language based on pure mathematics to our user Gladys, who had no technical or scientific background. One misgiving many people had about formal methods was that the technical documents would be unreadable by all except a few specialists, and this would render them impractical. If the PRG could produce a formal software development while working with someone with no technical background, it would give the lie to this common pessimistic doubt. Carrol Morgan and Bernard Sufrin enthusiastically took on the challenge and after a few weeks came back to tell us the result. They had had several meetings with Gladys and walked through the resulting Z specification with us. All seemed well and they proceeded to produce the program as described by the

specification. They chose Pascal, a popular high level programming language that was widely available, to implement the specification, and all went according to plan. The project and resulting system was called "CAVIAR", an acronym for Computer Aided Visitor Information And Retrieval.

From this exercise we and the PRG had a demonstration of the efficacy and utility of Z, a formal specification language for software. This demonstrator was useful for both parties. It gave us evidence to persuade our own management to go ahead with using formal methods in further real projects and it gave the PRG evidence that their researches were of practical industrial use.

In June the same year Tony Hoare, who was head of the Oxford University PRG, was seeking to have Jean-Raymond Abrial's fellowship from the Science Research Council renewed. Abrial was French, and this was before there was the freedom to work anywhere in the European Community. So Tony Hoare needed affirmation of the value of his being in the U.K. We sent a letter with a director's signature attesting to his useful work in technology transfer to ITT, which Tony Hoare could show to the SRC. So this academic-industrial collaboration was of mutual benefit.

Z and VDM could specify and model systems that are sequential, which means performing one action after another and having a beginning and an end. To model a process that is reactive and continuous, interacting with its environment or with a human user, and having no particular endpoint to its computation, was more difficult. Even more difficult is modelling parallel computation, in which two or more computers interact with each other. Telecommunication systems were typically reactive and the new microprocessors, which were coming on the scene, were bringing with them the possibility of many small computers acting in parallel to combine together to make a more powerful computation engine. So we were looking at other researches into ways of modelling reactive and parallel systems. There were several contrasting pieces of academic work being done into this, all with their enthusiastic protagonists. It was not easy to determine which approach would be best for our purposes and for some time we kept track of most of them. Robin Milner from Edinburgh University visited us and gave a presentation on CCS – Calculus of Communicating Systems[5]. We were all impressed with this. It could model parallel computations, with shared data, and communication of data between processes. It could also display the structure of a system of parallel processes, allowing one to analyse a system and express the model at different levels of detail, and to prove properties of a system such as absence of deadlock. Today's personal computers are reactive systems that can do some operations in parallel. How often they seize up and have to be restarted! The system has reached a deadlock. A little later Bernie Cohen and I attended a conference on the semantics of concurrent computation, the first of many over the forthcoming years. Several of the approaches used temporal or

[5] See Milner 1980

modal logic, in which one could make statements about the state of a system over time: something is true now, or will be true sometime, or will be true henceforth, or a combination of these. These systems have rules of deduction so that desired temporal properties can be demonstrated. Another formalism for modelling concurrent systems was developed by Carl Petri at the University of Bonn, which has become generally known as Petri Nets[6].

Although we did not fix upon any one specification or modelling technique, we embarked on a collaboration with Robin Milner and Edinburgh University. The idea was to apply the theory to a number of practical applications provided by us at STL which were of relevance to concurrency problems in telecommunications. It took nearly a year from the first inception of the idea to get started. We had to find a project and, more critically, to make the case for funding the collaboration. Robin had to find a suitable candidate for doing the work. In mid-1980 he proposed Mike Shields for the post. At the time Mike was working in the computer laboratory at the University of Newcastle.

I had encountered Mike the previous year at a conference on the semantics of concurrent computation[7] held in Evian, the small French town famous for its water. Mike had particularly struck me when he gave his paper at the Evian conference, because he said right at the beginning that what interested him was the mathematics of the problem. A gathering of notable computer scientists were there, Tony Hoare, Leslie Lamport, Glynn Winskel and others. Evian is on the shores of Lake Leman, the French name for Lake Geneva. A much travelled colleague from ITT recommended to Bernie and me that we stay in a particular small hotel in the town, which he declared was the best hotel he had encountered in all his travels in Europe. I must say that this was a delightful place, every room different and furnished with antique oak pieces. The hotel had a small open air swimming pool and one sunny morning we both had breakfast by the side of it. There was no breakfast menu – you just asked for whatever you wanted. In the bar all the way round the walls on a high shelf was the largest collection of Scotch Whisky I have ever seen. The prices were quite modest.

So it was that Mike Shields spent the time at STL busily reading up on telecommunications projects and principles. We sent him on an introductory course that most of us had been through, to give him context: Telecommunication Switching Planning, run by a charismatic septuagenarian British employee of ITT who had an apartment in both London and New York. Mike returned from this course energised and enthusiastic about the forthcoming work. We had thought of several possible applications for Mike to work on. There are concurrent program facilities in both the languages CHILL and Ada. There were three software R&D projects in ITT that were possible candidates. But beyond the confines of ITT, the CCITT had defined a standard language for expressing the design of telecommunication systems, based on finite state machines, SDL. We decided to ask Mike to work on the semantics of

[6] See Petri 1973 and 1980.
[7] See Khan 1979.

SDL – System Design Language. It was widely used throughout the industry, not just in ITT, and was the subject of an international standard. Improving the definition of this language would deliver an industry-wide quality upgrade, and benefit not just our own company.

I wrote the ITT case for funding Mike Shields over the next three years. It was accepted. It was going to be far easier for us if this effort could be funded from a single year's budget. Otherwise we would have to re-justify the case every year though its lifetime. If partway through, the case was declined, we would be in difficulties with our contract with Edinburgh University. So, rather to their surprise, I asked the University if we could pay for the whole three years of Mike's costs in advance. They agreed without difficulty! So began a fruitful and stimulating collaboration.

Mike worked on the semantics of SDL and of other protocols that commonly occurred in telecommunications. In mid-1981 I proposed to STL that we made a case for ITT funding to enable Mike's work on the SDL semantics to be presented as a contribution to CCITT Working Party XI-3-1. Since he was funded to do the work already, presenting it to CCITT would incur very little extra cost: a trip to CCITT in Geneva to make the presentation and a few days extra for preparation. It would reflect well on STL, I argued, and be a constructive contribution to CCITT. Mike used several different theoretical approaches to explore the SDL semantics: vector firing sequences and later, event structures that formed partially ordered sets. Numbers are an example of a fully ordered set: for any two different numbers, one is greater than the other. In a partially ordered set, one member may be greater than another, or neither may be the greater. A family is an example, where one member may be an ancestor of another. He developed this quite considerably, calling the subject "non-sequential behaviour", and it led to further ground-breaking work in years to come.

One of STC's most important customers was British Telecom, or the British Post Office as it was then. Post and telephone services were provided by the same public corporation. Not until the next year, in 1981, did the two services split into the Post Office and BT. System X, the first public digital telephone exchange system in the UK, was being developed and would go live in the first public exchange in Woodbridge, Suffolk, in a year's time. Charles Jackson of the BPO/BT Research Laboratories in Ipswich called together a group of researchers from the main suppliers to BT and founded the "Advanced Software Techniques Group". Bernie Cohen and I attended from STL, and others came from GEC Research Laboratories, Plessey and elsewhere. Charles set the aims of the group and did a very good job of conducting discussions by consensus while still subtly keeping to his own agenda and focus, which were forward looking and yet practical. This group came to be a forum in which the software researchers in rival companies got to know each other, gave each other informal presentations and discussed their work and technological ideas without interference from their own

management. And, of course, the meetings almost always took place on the neutral ground of our mutual customer, BT. The group became a collaboration almost by subversion.

At the inaugural meeting we set the aims of the group. These were to be wider than just the software involved in System X, and should consider technologies in the longer term. We were to think about telecommunications system design and how we believed software should be developed in three to five years time. We should also think about development environments, that is the sets of software tools needed to develop software using good engineering practices. In other words we should consider what we are going to build, how we are going to build it, and what environments we need to support those building methods.

I have a note from that inaugural meeting that shows how Charles Jackson was typically willing to be creatively eccentric. Successful engineering design requires aestheticism and complexity. Thus, he said, our designers need to be "good poets".

A telecommunication system has complex requirements: it involves combinatorial interactions of behavioural facilities. Multiple users share resources. There are asynchronous demands on the system from its operating environment. It has to respond in real time. It consists of concurrent components, which must therefore be free from deadlock and have other necessary properties. The sequences of its actions are, to a considerable degree, arbitrary. It must operate continuously: there must be no cessation of service. Enhancements and updates to the system have to be made on-line, while it is in operation. Being able to prove non-termination, i.e. that the system never stops, would help to ensure these properties. There are requirements on performance. These come under the headings of security, privacy, accuracy, integrity, availability and ability to be enhanced. The software would have to have properties that support these requirements. It should be well documented, well structured, accurately specified, with a low level of parallelism. Unavoidably, it would be large, with the problems that that entails.

So this forum began to explore the details of these principles and ideas. Over the next few years the ASTG came to endorse and encourage the use of formal methods, as well as considering several different design methods. The software that operates telephony stays embedded in the equipment for a very long time compared with many other computer applications. One of the most important and costly parts of the development of this software is the maintenance phase. This is the work done on the software after the original installation, testing and putting into service. The maintenance does not just consist of correction of faults. Over its lifetime the software will require numerous enhancements and extensions to take account of all the extra features and enlargements that the exchanges and provided services will require. Think of all the frequent BT updates that arrive on one's doormat, offering new options and possibilities. Each of these will involve alterations and additions to the operating software. The detailed phases of the continued development of the software over its lifetime,

its evolution, is called the Software Life-Cycle. The detailed phases of activities in the life-cycle came to be known as the Software Process. Professor M. M. Lehman of Imperial College put great emphasis on the study of the software process, and coined the term "Software Science" to denote it[8]. For example, even the simplest correction of an error in software consists of changing its design from that which was originally delivered. This is quite different from the equivalent correction of a hardware error. A piece of hardware will wear out, and degrade from its original design. Correcting an error will consist of restoring it back to its original design. By contrast, all software maintenance consists of some measure of redesign, in principle a much more radical change. Lehman's approach was to identify, observe and measure different activities of the life-cycle almost as if they were natural phenomena, and derive various laws relating to them.

The ASTG invited Professor Lehman to a meeting to talk about his ideas on software evolution. We were trying to define the education and training needs for software engineers for a report to the Post Office. For that we needed to know the skills that software engineers use. Those skills depend on the roles played by the engineer in the life-cycle. Those roles relate to the life-cycle activities, so we needed to have a clear idea of the nature of the life-cycle and its processes: the nature of maintenance, programming and even management. This led in time to the ASTG concentrating on software processes. Producing a computer program is a series of transformations of models, from a model of the application domain (telephony, engine control, railway signalling etc.) to an operational system. Testing a program to ensure that it is correct mixes two different concerns. Verifying a program compares it with its functional specification. Validating it compares its results with the real-world application domain. A process support environment needs to provide facilities that enable and assist the building of the various models that each stage of the life-cycle produces, the transformations between them, the verification and validation activities, and finally the planning and control of the process itself.

The ASTG continued until 1983, when BT was approaching privatisation. This resulted in changes in the relationship between BT and its suppliers. They wanted to foster joint enterprises rather than projects funded entirely by themselves. Environments, that is coordinated collections of software support tools, which assisted the software process, began to be the hot topic. With a standard environment, a large customer such as BT could become less dependent on individual suppliers and more easily switch between them. The driving influences on advancing technology were thus not just technical but also economic and political.

[8] See Lehman & Belady, 1985.

In the late 1970s the US Department of Defense became concerned about the large number of high level programming languages that were being used in defence applications. They numbered some thirty or so, and resulted in a large training overhead as well as hampering the transporting of programs from one application area to another. So the DoD started a process of commissioning a design for an advanced language that would supersede all the others and become the standard for defence applications throughout the USA. This eventually resulted in the Ada programming language in 1980. Meanwhile, however, the CCITT was embarking on the same kind of exercise for telecommunications. This effort resulted in the CCITT High Level Language, CHILL. In fact the effort to define CHILL predated Ada, although the two pieces of work overlapped in time to some extent. There was, of course, discussion within CCITT as to whether to abandon the CHILL work and adopt Ada as the telecoms standard language, but they preferred to go their own way. With CHILL the telecommunications industry would have control over its own language development.

By the beginning of 1979 a first definition of the CHILL language had already been written and a number of firms and research institutions were beginning to write compilers. The CCITT gathered the writers of these compilers together with other interested parties into a Working Party to make sure the compilers were all compatible and interpreted the language in the same way. The committee structure of the CCITT was large and complex. A collection of subcommittees formed Group 9, of which WP3 was concerned with SPC, Stored Program Control languages. SPC was the telecommunications industry's term for software and computers embedded in, and controlling exchanges. WP3 consisted in turn of several committees, one to work on the semantics of SDL, a Software Design Language, another on the CHILL Implementers Forum, and a few other committees. Each of these bottom level committees appointed a rapporteur who reported up the hierarchy by submitting papers to a plenary session. Don Combelic was on the CHILL IF mainly as an observer, although one of the other ITT laboratories had started to implement a compiler and also participated in the forum. Don asked me to join the CHILL IF alongside him. I think he wanted someone to explain the more technical issues to him so that he could make a better judgement about any strategic consequences of the decisions that were taken. I was one of not very many people in ITT that he knew with substantial compiler experience. Don made representations to Frank Simpson, my Division Manager, to authorise me to join. The first meeting I attended was in London, which eased Frank's agreement; Don mentioned that subsequent meetings were hosted by participating organisations and that later in the year there would be a meeting in Melbourne. Frank agreed, although he implied that he would have reservations about the cost of a trip to Australia.

These meetings took place quite frequently, every three months. Before the London meeting there were two preliminary meetings. One was amongst the interested ITT parties, in preparation for the next two CHILL IF. Several ITT companies in Europe and the USA were

involved in various CCITT committees and this first pre-meeting took place in ITTLS in Madrid. I was concerned about to what extent I was representing ITT, or STL, or neither. I wanted to know what my liabilities were, what were the expectations upon me. The answer was specific but quite subtle. One participates on these committees in one's own right, and as a representative of one's own company, in my case STL. However, one's participation should be compatible with ITT interests. This ITT committee, which I was attending there and then in Madrid (for the first time), organised channels of communication from the ITT companies to the CCITT committees. The leadership of this ITT committee used to be with the UK. Nowadays it was shared between the companies of ITT Europe. There was a nomination procedure for formal ITT representatives. All this had been thought about at length. The procedures were laid down in a booklet written in 1974, some five years earlier.

CHILL was not the only issue that this ITT committee was considering prior to the next few CCITT meetings. It also discussed SDL. One criticism was that SDL had little structure and could not hide details or produce an abstract view of a design at higher levels. There were shades of Dijkstra's separation of concerns here. Don proposed that the semantics of SDL should be defined so that any designs expressed in the language had a clear meaning. This was in time to lead to Mike Shields' work at STL, already mentioned, which would later feed into the CCITT definition of SDL.

A second pre-meeting was called by the BPO for the British participants. The Post Office wanted the UK to present an agreed view of any difficulties or questions. GEC Hirst Laboratories hosted this meeting. The discussions were right down at the technical details of language syntax and facilities. Some features, we considered, were not worked out enough, others should be amalgamated or dropped.

The CHILL Implementers' Forum then met in February and May, in London and Florence. There were about ten implementations of the language in various states of progress and in various countries: Italy, Germany, the Netherlands, the U.K., Denmark, Norway (a pan-Scandinavian cooperation), the U.S.A., Japan, France. In some cases a telecoms company was implementing a compiler, in other cases an administration, that is a national telecoms and/or post office service provider, or an administration was subcontracting to a software house or forming a collaboration amongst its national suppliers. The British Post Office was planning to subcontract an implementation to a software house.

Every compiler potentially has a host and a target computer. Most people who use a computer language are accustomed to these being the same. One translates a Basic or Java program using a compiler on a PC and then runs the translated program on the same machine. But they need not be the same, especially when one is producing software for an embedded machine, that is a computer that is part of a larger engineering device like a manufacturing plant or a telephone exchange. The software will typically be prepared on a general purpose machine

and the translated code will run on an embedded microprocessor. The different implementations had various different host and target machines. A popular target microprocessor at the time was the Intel 8086, but there were others.

The choice of a computer language, indeed any computer technology, easily ossifies. In the case of CHILL, administrations were to some degree forced to accept the technology that their suppliers had to offer. The suppliers, on the other hand, to keep competitive needed to anticipate the technology they thought the administrations, their prime customers, would require. This kind of positive feedback effect had a disadvantage of being potentially in disregard of the absolute technical merits of a particular technology, but also had an advantage of tending to establish widely accepted standardisation.

The Implementers' Forum observed the progress of the several compiler implementations and sorted through many language ambiguities and difficulties that were revealed by the task of trying to implement it. The members of the forum would bring to the meeting proposals for changes to the language and if accepted (most were after discussion), these would be incorporated into the "Blue Document", the on-going language definition. The Blue Document was to be recast by the end of the study period into a new Brown Document.

Unusually for a computer language, CHILL had two alternative concrete syntaxes. The concrete syntax is the surface form of the language; underlying both concrete syntaxes was the same abstract syntax, which embodied the structure of the language. The first stage of any traditional compiler is throw away the concrete syntax of a program in order to reveal its abstract syntax. With CHILL, one concrete syntax resembled PL/1 and the other resembled Pascal, both established programming languages but without the concurrency and many other features of CHILL. We agreed that any one compiler should accept programs written in just one syntax, not in a mixture of the two. In fact it was mainly NTT, the Japanese member, who required the PL/1-like syntax. All the others were going for the Pascal-like option.

The Technical University of Denmark, a member of the forum, was producing a formal definition of the semantics of the language. To date, formal definitions, that is mathematical models of the meaning of languages, had been constructed only after a language had been implemented and in use for some time, as, so to speak, an afterthought. This was, I believe, the first time that a formal definition was produced as the language itself was being defined. This work would help to contribute to a total language definition, syntax and semantics, that would form part of the standard. At the time I felt that this activity was innovative and would assume considerable importance over the next few years. I am not sure that this has been recognised, but since then an increasing number of formal language definitions have become parts of ISO and BSI standards for computer languages.

CHILL was quite a sophisticated language. I'll briefly describe some of its features. Move rapidly on a few paragraphs if you want to skip these more technical details. There were

powerful facilities for defining data types. The discrete types that could be defined were integer, character, Boolean, enumerated sets and ranges of these. Powersets, that is sets of elements of a discrete type, could be defined as a type. Composite types could be defined as arrays, strings or structures (like Pascal records or records in databases) of other types. There were also reference types for handling references to variables. Finally there were types related to the concurrency facilities. Events and buffers could be used to synchronise processes. Instances of processes, or tasks, were types. Processes could proceed concurrently and would be identified by a variable of process instance type. Types could be read-only and types could have scoped access in a recursive context. The types in CHILL were called "modes".

There were the usual conditional, loop and procedure call statements, and a range of exception conditions such as array bound overflow. There were also Assert statements, as in Ada, that could give rise to related exceptions.

There were several facilities for programming concurrent processes. A process instance could be created on obeying a START instruction and executed concurrently with other processes. Two further synchronisation modes or types were available, events and buffers. A process could wait for an event by means of a DELAY statement, or resume execution with a CONTINUE Statement. Buffers and Signals could be used for communicating information between processes. Finally, Critical Regions could be defined for providing mutual exclusion on access to common resources. When a critical procedure is called, it cannot be interrupted or suspended. All other critical procedures are locked out from execution until the called critical procedure exits. Language restrictions help to ensure these rules are followed, for example critical procedures cannot call each other.

These concurrency features of CHILL were partly based on the customary practices in telecoms software at the time, and partly on well respected computer science work. This summary is brief and a simplification of the actual features, many of which will have changed over the years.

Although CHILL is not a widely known language amongst general software engineers these days, it still has a substantial user base in telecoms. There are today large teams numbering hundreds each, who are continuing to write large scale software in CHILL for telecommunications applications.

The final CHILL IF in 1979 was held in September in Melbourne. My Division Manager, Frank Simpson, was reluctant to authorise the expense of my attending. I explained this to Don Combelic, who telephoned Frank there and then. Don was a powerful personality. When Frank called me back to his office after the phone call, he positively instructed me to go to the meeting. The BPO held another meeting of the UK representatives in preparation for the Melbourne meeting. Their main recommendation was for a language control committee to be

set up afterwards in order to ensure the continuing compatibility of the various implementations and to review its competitiveness with Ada. There should also be a mechanism for certifying compilers.

So I went to the Melbourne meeting of the CHILL Implementers' Forum. An aunt and a cousin of mine lived in Sydney, and I combined the trip with a visit to STC Sydney, who were one of the ITT customers of the 3200 BSCC, and stayed with my aunt. I had never visited my Australian relations on their home ground before, so this was an added benefit for me.

The CHILL IF meeting in Melbourne was spent mainly in completing final details. All the implementations were in various degrees of progress, several of them advanced. Indeed, the ITT implementation was complete, as was the Japanese compiler, produced by NTT, which had been in use for some months. The Danish implementation had been suspended while they carried on with the formal semantic definition. A team, headed by Dines Bjørner, was writing this in the VDM language, the development of the VDL, Vienna Definition Language, that Dines and his colleagues had used to define PL/1 at the IBM Vienna Laboratories a few years earlier. The formal semantics exercise had already uncovered nineteen inconsistencies and errors in the language description, so it was worth the effort for that reason alone. The meeting organised the production of a useful introductory manual.

The meeting in Melbourne was, essentially, the end of the main effort of the Implementers' Forum and the end of my involvement in CHILL. ITT decided to conduct a trial of the language by reprogramming part of an existing system in CHILL. They would then obtain feedback to assess the advantages of using the language in a typical telecoms application. They wanted particularly to find out about how readable the code was and how easy it was to maintain. They were also interested in whether using the language improved productivity, and how accurate the implementation was. ITT also planned to study the efficacy of available tools for processing the CHILL language and to establish requirements for new ones. CHILL used more sophisticated data types than existing languages such as Post Office CORAL, and it was important not just to translate existing software into CHILL; the design had to be re-expressed in terms of abstract discrete structures, which could then be expressed as CHILL data types. So a measure of deconstructing back to high level design, i.e. reverse engineering, and reconstructing in the light of more advanced data structures was necessary in order to make an effective comparison.

About a year after the final Implementers' Forum in Melbourne, in October 1980 a further ad hoc meeting took place at DataTechnik in Denmark. Implementers from seven of the participating countries took part and described the state of development of their various compilers and the experiences they had had in using the language. The Technical University of Denmark gave a comprehensive account of their work on the formal definition. They had

had to make their own extensions to the VDM language to be able to model the concurrency features. The formal modelling of concurrency was to become a compelling issue in program language semantics over the next few years. The CCITT continued to coordinate work on CHILL compilers and other support tools for the next few years. A user manual was produced in 1982.

Meanwhile, at STL we continued to be aware that the US Department of Defense was sponsoring work on the Ada Language. In the nineteenth century Charles Babbage had built his mechanical computing engines, parts of which are to be seen in the Science Museum in London. His assistant was a mathematician called Ada Augusta Lovelace, and she is commonly regarded as the world's first programmer. The Ada language is named after her. The DoD wanted to fix on one language for defence applications in order to reduce redundant effort. They decided that this language should have all the advantages of those currently in use, and be technically in advance of them. This meant inventing a new language. So Ada had an ancestry of previous languages: Coral66, Simula67, PL/1, Pascal, Jovial, RTL/2 and others. In 1975 the DoD set up a working party to define, not the language itself, but a set of requirements which the language definition should meet. This set of requirements was to be open to public scientific scrutiny and criticism, and so it was called the Strawman document, a straw man set up to be knocked down as in a traditional country fair. This document went through many iterations, each with a new name to indicate its increasing rigidity and stability. Woodenman and Tinman were defined in 1976 and then the DoD invited proposals to define the language. Seventeen proposals were submitted, of which four were short-listed. These four were named Green (submitted by CII Honeywell Bull), Blue (by Softech), Red (by Intermetrics) and Yellow (by SRI International). In 1978 the shortlist was reduced to Red and Green. Intermetrics and CII Honeywell Bull were given one year to refine their designs and resubmit. Green was chosen in 1979, and so Green became Ada, by fiat so to speak. The requirements were meanwhile refined again to become the Steelman document and Ada was finally revised again in 1980, after which it became defined as an ANSI standard, ANSI being the American National Standards Institute. A reference manual was produced in 1983.

With all this activity on producing a standard programming language across the Atlantic, the British DTI (which was then the DoI, Department of Industry) felt that the UK had better not be left behind. They believed there was a strong chance that Ada would become widely used in many applications, not just military ones, and so UK industry had better get up to speed in this new language. To use the language for projects one needs not just a compiler but also other language support tools, loaders to load the compiled code into a target computer, linkers to link together separately compiled programs, debugging tools and many other software tools to assist the programmer. The collection of all these came to be known as an environment; it was the technical support environment in which the programmer was working and producing a program. So the DoI planned to sponsor the building of a UK support

environment for Ada, APSE as it was called – Ada Program Support Environment. They would issue invitations to tender for this work.

During 1979 several of us learned about Ada and held invited in house seminars on the language. In April 1980 we had a meeting with Ferranti with a view to collaborating. Ferranti had already applied to the DTI to be invited to tender. In May we had a visit from ICL. Their interest was increasing and they saw Ada as a possible systems programming language, that is a language in which they might write operating systems software for their own ranges of computers. ICL had a product called CADES – Computer Aided Design and Evaluation of Software, which had many similarities in objectives to the proposed APSE. The APSE was to be oriented to databases, organised large volumes of data, and ICL had a lot of database experience. There was a strong competitor for the DoI tender: Logica had formed a consortium with several other companies, which they had strategically called the "Ada Consortium". In mid-1980 we met with CAP, Ferranti and Scicon, who were apart of BP, and initiated our own consortium. To vie with the competition we called ourselves the "Augusta Consortium", Augusta being Ada Lovelace's second given name. In July we were agreeing about what questions we needed to ask at the bidders' conference, which the DoI was shortly to hold at RSRE. RSRE, the Royal Signals Research Establishment, later became DERA and subsequently the more independent Qinetic. RSRE was at that time part of the Ministry of Defence. We spent much time working out a work-plan for the tender and a strategy to follow at the bidders' conference. By then SWURCC, the Southwest Universities Regional Computer Centre had joined Augusta. There was a lot of work to be done in establishing a working consortium. Memos of understanding were drafted, and we needed to choose a prime contractor. There was some competition for this prestigious rôle, but soon CAP was chosen. Should we allow subcontracts to further parties to provide specialist expertise if necessary? We decided to keep flexible. The APSE bidders' conference was held in late July 1980. The MoD and the DoI were to share the funding of the work. RSRE were acting as the MoD customer. Formal invitations would be issued in two to three weeks. The winning bid would be chosen on value for money rather than just least cost. They would take particular account of the technical strength of the bidding team, targets and the work plan. Our consortium expanded again in September to include Imperial College.

The weeks went past and we had not been invited to tender. On enquiring we discovered that the contract had already been let to our rivals, the Ada Consortium. We held another meeting and sent a strongly worded complaint to the DoI, pointing out that letting the contract to one consortium without considering other contenders was non-competitive, and the government could be open to a charge of favouritism. The DoI became quite agitated at this – it was clear that they had made a serious mistake in protocol – and by way of compensation they offered us a subsidiary study into the software development methods that could be used with Ada programming. This was a substantially smaller piece of work than the main APSE, but we

accepted it. So, in November 1980, as part of the Augusta Consortium, Mel Jackson and I from STL embarked on the Ada Methodology Study.

The other members of Augusta were CAP, Ferranti, Imperial College, Scicon and SWURCC, the South West Universities Regional Computing Centre. CAP were the prime contractors, but that didn't mean they were necessarily the project team leader. Our various managers were present at the first meeting to initiate the project. We decided to split the leadership rôle into two parts: administrative and technical. CAP would take on the administrative rôle and I would be the team leader. One of the CAP members felt that the technical leader rôle would be diminished by not also having responsibility for administrative matters like time recording and expense claims, but I declared that I was quite happy with the arrangement. "It means that I get to do all the interesting work and CAP take care of all the boring bits", I said. Peter Weston, the manager from CAP, seemed to be amused by this characterisation of mine and thereafter referred to "the boring bits". I began to regret my spontaneous coinage. However, the arrangement remained and it went very well as far as I was concerned. The customer, the DoI, in consultation with us set up a steering committee to act as intermediary and general policy aides. This group of eleven came from industry and scientific civil service: British Steel, ICI, INMOS, British Telecom, British Aerospace, the Central Electricity Generating Board (CEGB), which has now been privatised and replaced by many different energy supply companies, Easams, the Atomic Energy Research Establishment (AERE), the National Physical Laboratory (NPL), and RSRE. Mike Pickett, the manager from CAP, acted as liaison between the project team and the steering committee. He would report on our progress and represent our position to them and come back with messages from them from time to time. Mike's skilful handling ensured that our relationship with the customer had a smooth ride to the end of the project.

Ada was and still is a sophisticated and complex language. There was a danger that if pressed into use prematurely, its strengths and weaknesses would not be properly understood. In order to capitalise on the language, our study tried to relate the language features to the development process and identify the methods of working which would produce the most benefits. We carried out a literature review of twenty-one development methods. These addressed various stages of the development life cycle: requirements analysis, specification and design. We did not look at any methods that catered for the maintenance phase, for no better reason than in those days, maintenance was not seen as so crucial. There was a clear understanding then, in 1981, that computer systems inevitably evolve and maintenance was important, but methods to support evolution were not much to the fore.

No single method is equally applicable to all applications, situations and stages of the life cycle. While the literature study revealed a lot of information, printed matter necessarily does not tell the whole story, so we visited potential users and developers of Ada-implemented systems, twenty-four organisations in total. These visits gave us more insights into the use of

methods and notified us of a further fifteen, on which we reported in outline in the study. Then we selected six of the methods and applied them to a couple of example problems, developing Ada designs and partial implementations in each case. We found that the more sophisticated featrures of Ada, such as packages, generics and overloading, could be used to beneficial effect. On the other hand, they could also be misused or simply ignored through lack of understanding their purpose. Amongst our fairly wide-ranging conclusions, we strongly recommended that a first imperative before using any technical method was to establish a defined procedure for recording and communicating the outputs of the life-cycle which helps to follow a disciplined approach. This presaged the future industrial standards for quality processes, BS5750 and ISO 9000, and the influential work on the Capability Maturity Model initiated by the Software Engineering Institute at Carnegie Mellon University.

We published the findings in September 1981 and presented them to an invited audience at the National Physical laboratory in December that year.

It is rare in industry that one is working on just one task at a time. During 1979 and 1980 I was also engaged in several other avenues of enquiry, besides CHILL and Ada. In some of these, other projects merely asked me for advice and involved me in meetings, using me as a kind of internal consultant. In others I was more heavily involved. Two of the more intense efforts were our adoption of VDM as a software development method, which we were to propagate through the company, and a more temporary flirtation with a software design system called Gamma, the brainchild of Dr. Mike Falla from Software Sciences Ltd.

Bernie Cohen and I had been impressed by the descriptions of VDM which Cliff Jones and Dines Bjørner had given at the Winter School in Abstract Software Specifications in Copenhagen in January 1979. Cliff had given courses in VDM when he was at IBM and had written a book, Software Development, A Rigorous Approach, based on these courses. We ordered ten copies of his book and distributed them amongst some senior technical software staff at STL. In June 1980 we and two of the STL Division Managers visited Cliff, who was now at the Oxford University Programming Research Group, headed by Professor Tony Hoare, to discuss his giving us courses in-house. He could offer a two-day seminar for managers and a longer in-depth course based on his original three-week courses at IBM. For starters we took up his offer of the two day managers' course and he duly presented this to us in July of that year. Cliff's industrial background was at IBM, where there was a "dry" tradition. No alcoholic drinks were allowed on IBM premises or could be consumed on company expenses. This was far from the case at STL, part of ITTE whose nickname was "International Travelling, Talking and Eating". Cliff was, I think, a bit taken aback when during his course we broke for lunch and a canteen staff member pushed open the lecture

room door and wheeled in a trolley of clanking drinks! This was to be the beginning of a long association with Cliff and VDM.

After several meetings and in-house courses with Cliff we began to deploy formal methods in STC. One of the difficulties in getting VDM and other formal methods accepted was that they all used certain aspects of pure mathematics, namely set theory and symbolic logic. Graduate engineers have mostly been taught applied mathematics. Differential and integral calculus are the topics that underpin the traditional engineering topics like electronics and mechanics. Although the kind of pure mathematics that lie at the heart of formal methods is mostly very elementary, such as a first year maths undergraduate would be taught, it is just that little bit more abstract and the symbols used that little bit more unfamiliar. Most practising engineers shied away from this unfamiliar ground at first. I decided to try to overcome this by putting together a course in set theory and logic, "discrete mathematics" as it is called, and give it to volunteers. STL had recently introduced flexible working hours, so that two hours were reserved for lunch between midday and 2 pm. In general meetings would not be held during this time and staff could take as long or short a lunch break as they wished, and accumulate the hours worked. This enabled people to work longer at times and less time at other times, even taking a whole or a half day off if they had accumulated the hours. Quite a few organisations were beginning to institute this innovative scheme, which is now quite commonplace. So I gave the discrete maths course during lunch breaks once a week. A number of people attended, including my manager Frank Simpson. By 1982 I had shared the course material with two other colleagues, Chris George and Paul Taylor and the three of us were regularly delivering it to project teams within STC.

We set up a series of in-house one-day conferences on formal design methods, to which we invited external industrial people and academics, where we discussed more general formal methods and techniques. Each conference had a theme such as "Trends in Design Techniques" or "Emerging Formalisms". We thought that it was important to get the managers of software projects on our wavelength, so we held a symposium specially for software managers. After a lot of consultation with Cliff Jones, Mel Jackson, Roger Shaw and I started to give courses on VDM ourselves to projects and teams in STC. Several times we held these outside office premises, in small conference locations in the south-east of England. Some of these had other attractions: a large ex-manor house in Ware had beautiful Edwardian plumbing fittings in the bedrooms, and a more modern facility in High Wickham had its own swimming pool, squash court and snooker table. One STC telecoms project, code-named Midwinter for the mundane reason that it was initiated on December 21st, agreed to use VDM for specifying at least parts of its software. This was a substantial exercise in technology transfer, with tailor-made courses and internal consultancy. Midwinter had a whole lot of technical features that would certainly be changed and extended over its lifetime. We had to consider how to devise a central design philosophy to facilitate the attachment of

these features after installation and delivery. We were fortunate to have a champion in the STL personnel department, Charles Harding, who had some responsibility for training within the laboratories. He decided to record one of the courses on videotape, and set up a CCTV camera at the end of the lecture theatre. It was early days in 1981 for this kind of thing, and the lighting had to be turned high over the stage and low over the auditorium. It was the first time I had been "televised" and I found it unnerving not to be able to see my audience and their reaction to what I was saying. STL retained Cliff Jones as a consultant for some time in order to assist with technology transfer to projects such as Midwinter.

Over the next few years, 1981 to 1984, we transferred VDM to two more STC telecoms projects, Fridge and Telspec. But we also gave presentations on VDM to a few companies outside the group, notably GEC and BP. The main protagonists of VDM were Cliff Jones, then at the University of Manchester, and Dines Bjørner at the Technical University of Denmark, who had started up a campus company, the Danish Datamatics Centre. Inevitably, slightly different usages and conventions with the language of VDM began to emerge, and we all agreed that we should try to coordinate the evolution of the method, to try to keep variations to a minimum. So a VDM coordination committee was set up in 1983, with representatives from several companies and academic institutions. Throughout this time, minor changes and improvements were discussed and made to the VDM language. Together with Cliff we worked on more techniques for proving the consistency of a specification in VDM and for proving that a program fulfilled a specification. We tried VDM out on several case studies within the Fridge and Telspec projects to see how readable a specification would be, how easy it was to make it complete and consistent, how easy to check if it was correct, to develop a manual of "style", and in general to test the "usefulness"of the technique. All of this required not just the technical work but also producing the accompanying literature, internal brochures, posters and course notes, and writing proposals for funding the collaboration between us as a group in the research laboratories and the teams who were doing the "real" work of writing software for the telecoms project.

If a software development team learned to use VDM, it was important to get the team manager on the same wavelength. Because it was a novel technique, even if the managers were more experienced than their team members, they would not be familiar with VDM. So we developed a version of Cliff's two day managers course for them. This covered the aims of formal methods, their impact on the management process, the rôle of formal specifications, and an exercise in reading them. To my surprise I found that many managers were keen to come on these courses. I had expected resistance – old dogs not wanting to learn new tricks. But in fact many of them welcomed the opportunity to escape from their administrative duties and recall some of their technical expertise, which had in some cases been in suspension for some years.

My colleagues and I were filled with enthusiasm to propagate formal methods, and VDM in particular, as widely as we could. We were keen to market these courses, which we had developed, further afield, outside the company. Doing so would also validate the credentials of what we were doing: if other firms were willing to pay for it, it must be good! However, this met with some resistance amongst our upper management. They saw it as giving away our technical advantage to our business rivals. At one point, STL's managing director saw the originals of the VDM brochure in the print room, asked what is was about and stopped the print job. Neither he nor the print room told me, and I only discovered this had happened when I chased the progress of the work. This irked me considerably at the time. We at length settled half way: it was agreed we could give the courses to STC customers, but not in general to other external companies.

Gamma was a software design technique developed by Mike Falla at Software Sciences Ltd. in Macclesfield. Barclays Bank, who had a large team of some 80 programmers designing and programming the software for the bank's up-coming automatic cash-point system, was partially funding the development of Gamma. In return Barclays received the Gamma system and tuition on how to use it. Barclays' technical personnel policy at the time was quantity rather than quality. Only the leader of this 80-strong team had a university degree. SSL were keen to find other customers to fund Gamma, at a rate of about £25k each. We prepared a case for submitting to ITT headquarters for our funding of Gamma.

Gamma was essentially a tool that could support the use of a software development method. It had been used with JSD, the Jackson System Development method which was based on JSP, but SSL were keen to pilot its use with other methods. We were impressed by the good management of the Gamma project, their working papers, sound scheduling techniques and work analysis. However, we had the problem that all ITT research funding had to be re-justified every year, and we felt that we would be unlikely to obtain the necessary authorisation for a subsequent year. It fell to me to drop this bombshell to SSL at a quarterly Gamma meeting in August 1979. The Barclays representative was particularly irked by my warning that we would likely pull out in the new year. He talked about the damage to goodwill and wondered aloud whether they could exert any influence through the DoI or discover anyone with joint directorships in STC, the British Oxygen Company (who owned SSL) or Barclays Bank. We held numerous subsequent internal meetings in which we went over our own motives for our interest in Gamma and tried to decide future policy. We were more concerned to gain input to our studies in methodology rather than to acquire the Gamma technique itself. Our support stumbled on; we attended the next two quarterly reviews but by July 1980 we decided definitely to pull out. I composed a letter to SSL, consulting with George Power, our contracts manager, who was the nearest person to a lawyer that we had at STL. He added a paragraph and the letter was despatched.

For some time STL had recognised that the next generation of computerised telephone exchanges would embody, not minicomputers like the 3200 (a more well known mini was the PDP11), but microprocessors. A microprocessor research department had been running for some while, under the leadership of David Wright. Microprocessors are distinguished from minicomputers by having the electronics of their central processors held entirely on a single chip. Raw microprocessors were available from a few manufacturers such as Intel. To build a computer based on a microprocessor and capable of being embedded in a telephone exchange required a substantial amount of digital electronic design. A research project within David Wright's group was designing the architecture of the Next Generation Machine System, based on a microprocessor, namely a member of the Intel 8080 series. We had numerous discussions on this NGMS architecture, of how it could be made to support high level languages easily, of the means of avoiding the glitch between two or more communicating processors, by, for example, synchronising their clocks. The glitch, by the way, was originally a very specific event in digital electronics, when two signals occur absolutely simultaneously, resulting in two mutually exclusive paths being partially taken. The probability of this happening is astronomically small, but when the processes are performed millions of times per second over months and years, that astronomically unlikely event eventually happens, perhaps quite often. I believe that this may be a consequence of a quantum physics phenomenon, possibly resulting from Heisenberg's uncertainty principle. The word "glitch" has now through popular usage lost its original highly specific technical meaning, and has come to mean almost any computer related malfunction.

The microprocessor research group began to design an operating system for the Intel 8080 series. But eventually this work was overtaken by the Microsoft product, MS-DOS. I can imagine that many parallel pieces of work like this were going on in different establishments, most of them in time abandoned. When I had worked at ICL my department manager once remarked that the majority of the software developed would be thrown away. But he asked me not to tell too many of the other staff.

Chapter 9 The Search for Grants

The UK had been a member of the European Union since 1973. The EU then numbered nine countries, and Greece joined in January 1981 bringing the number to ten. With ten member states, the EU was beginning to encourage innovative projects, to improve the union's prospects of prosperity and advancement, and not just to apply assistance to deprived areas, although it was doing that as well. Within a year the European Strategic Programme for Research into Information Technology, ESPRIT, would be set up, but in 1981 this had not

yet happened. Preliminary to ESPRIT, research projects were funded on an ad hoc basis. I had for some time had an idea that I would love to see explored. A dream of software engineers is that of a reusable library of useful programs. So much effort was and is spent on programming, repeated, wasted effort. If only one had a means of finding that program one needed to write already in a library somewhere. I had my own dream that one would discover that the same programming problem occurred in a variety of very different areas, say commercial databases and compilers and civil engineering for example. A piece of software is blind to the application in which it is put to use; the same computational problem may be being solved over again in dramatically different contexts. If only one had an application-independent way of describing and indexing theses pieces of programming, one could maybe then build a library and look it up to see if the required program had already been written.

One clear candidate for an application-independent description of a program is a formal specification of it. A formal specification defines what a program does, not how it does it nor in what context it is used. Even so, with specifications written in the then two most popular formal languages, Z and VDM-SL, there is a great deal of freedom to express the same specification in different ways: there is a considerable freedom of expression. These two languages are examples of model-based specifications. One composes a model of the function of the program, using set theory. There are even more abstract methods, based on universal algebra[1], for expressing specifications. These come under the heading of Abstract Data Types, ADTs. The Ada programming language was a step towards programming with ADTs, and the more recent Object Oriented techniques and Java programming language are a step further along that path. The functions in ADTs need to be defined, and the most abstract means, and therefore the easiest to process automatically, consist of axioms expressed as equations. So we decided to use ADTs with equational axioms as specifications and would explore how to search and index them.

Looking into this problem required some fertile brains. We had a few at STL, but could use more for a project that required such deep appreciation of underlying theory. Further, the European Commission, who were the body that let EU research funding, favoured “pan-European” collaboration and indeed required projects to be a cooperation between organisations in more than one EU country. I had met several people on the Chill committee. Working on technical committees gives one an excellent opportunity to tell how capable other members are and whether one could work with them. I contacted Rudi Meijer, who had been on the Chill IF and who worked for the research labs of the Dutch PTT, Dr. Neher Laboratories, DNL. He demurred but suggested a colleague of his, Kees Middelburg. I knew Kees too and was quite happy to contact him and propose this collaboration. There followed several meetings with Kees and DNL, with our own contracts people at STL, and with the administrative officer from the EC. There was a limit to the funding that the EC would

[1] See Cohn, 1981.

provide for research projects like this. The limit was 100,000 ECU – European Currency Unit. The Euro, the European currency, did not exist then; it came into existence as coins and notes at the start of 2002, although the hard currency had been distributed as starter kits, not legally usable, from September 2001. But the European Monetary System, EMS, established a currency unit, the ECU, at the end of 1978. This was the official European Union accounting unit until the end of 1998. On the 1st January 1999, the Euro came into being, replacing the ECU and having exactly the same value. So although there were no coins or notes, transactions and bank accounts could be set up in ECU and later in Euro. Indeed, I had a ECU bank account myself in 1998 and it magically transformed into a Euro account on 1st January 1999.

So, with Dr. Neher Laboratories in the Netherlands, we submitted a project proposal to the European Commission. After scrutinisation by a review committee, and some consequent revisions, our proposal was accepted. Many organisational details had to be sorted out. STL would be the prime contractor, so we had to subcontract to DNL. We could choose in which country's system of law to make the contract: the EC suggested Belgian law, but I think we agreed on the laws of England and Wales, which were close to those of the Netherlands. Our proposal had to declare our methods of interworking between the two participants, to provide track records, organisation trees, and lists of personnel and their CVs. This was to be the norm for project proposals submitted to both the EC and the British DTI.

Whenever the EC or the DTI funded a project, they would allocate a project officer from their own staff to it. The project officer would often champion the project from the beginning, making the case for its funding amongst his or her own colleagues within, in this case, the EC. Our project officer turned out to be Rudi Meijer. Unbeknown to me, he had been seconded from DNL to the European Commission. His secondment must have been in progress when I first approached him to collaborate on our project. Now I understood his initial reluctance; he would soon be our "customer" and could scarcely be a collaborator in the work. It was clear that Meijer was sympathetic to our project. "It was the only proposal we have received that is scientifically respectable!" he said to me, with inverted hyperbole.

Unfortunately, we never came up with a good name for the project. Future projects in the forthcoming ESPRIT would have acronyms like RAISE or IPSSI. Ours remained the verbose "Methods of Defining, Cataloguing and Retrieving Specifications of Abstract Data Types". Not very snappy. The team consisted of Will Harwood, Paul Taylor and myself from STL and Kees Middelburg and Jos Feinig from DNL.

We started off by defining "scenarios" of how someone might use a library of ADTs that we envisaged. There was a lot of reading to do: researches were apace in the USA, particularly by a number of computer scientists known as the ADJ group, and at the University of Edinburgh. Several experimental axiomatic specification languages were published: AXES,

INA JO, AFFIRM, OBJ, CLEAR. We studied these and several others which we later put on one side as they were less relevant to our purposes.

An abstract data type comprises a signature, which is the set of data types, and the operations upon them. It also includes the axioms, which we had decided would be expressed as a set of equations. If a user of the library wants to look up a data type, he/she would present it with the signature and axioms of the desired ADT. The first thing that the system should do is to try to match the signature of the presented ADT with the signatures of those in the library. If a match is found, then the axioms have to be compared.

Two sets of axioms are equivalent if each can be deduced from the other. Each set will comprise theorems provable from the other set of axioms. So automated theorem proving techniques would be necessary. Plenty of research work was being done in the field of automated theorem proving. Automated theorem provers used a technique of term rewriting: transformations of logical terms which preserved their truth values. PROLOG was one of the first logic languages but others were around too.

Matching two ADTs turned out to be difficult. We considered going for an interactive approach, where the user interacts with the system and guides it, rather than a completely automatic one. The user might be able to interrogate the axioms to see whether some theorems are deducible from them, and to do experiments with formulating hypotheses. We experimented with various usage models for the proposed system. To give ourselves focus, we chose a case study: a database for an employment agency. The research of Rod Burstall and Joseph Goguen from the universities of Edinburgh and California at Los Angeles was particularly relevant to us. We had included a budget for some consultation with experts in the project plan, so we arranged a visit from him at STL. Rod Burstall and his colleague Don Sanella spent a day with us in January 1983. We had meanwhile also tried out as an additional case study part of the aircraft monitoring system, which had been figured in the Augusta study. Our meeting with Rod Burstall and Don Sanella gave us some more insight into deriving one ADT from another. Burstall was beginning work on a topic for which he coined the term “Institutions”. These were abstract algebras with a more flexible underlying logic than equations and term rewriting. All this was very relevant to our project.

We carried out experiments on the two case studies, but after a lot of intense work and discussions, by June 1983 we were forced to the conclusion that automatic matching of specifications as we had conceived them was not possible. We were fairly sure of this conclusion but thought that the EC may wish to seek further investigations to verify it. We nonetheless felt that there was a place for such a library of ADTs to assist software development on scientific principles. The study was nearing its scheduled end and we had a final report to produce. We had produced some half dozen technical papers delivered to the EC during the course of the study, but also nearly a hundred working papers restricted to

circulation within the project, and over sixty administrative papers and minutes (written by myself). We sent the first draft of the final report to the EC in August 1983.

Rudi Meijer, the EC project officer, wanted to hold a review of the project by a panel of experts. He telephoned me and gave me a long list of some of the most respected computer scientists in Europe and asked me to invite them to a project review. I felt as if I had been asked to invite a firing squad of marksmen to my own execution. I duly telephoned the list and rather to my alarm well nigh all of them, thirteen in number, agreed to attend. We all went to the review meeting at the European Commission in Brussels in December 1983. Will and I gave the majority of the presentation. We described the rationale for the study, the two teams from STL and DNL, and the preliminary literature survey of methods and formalisms for defining ADTs. Then we went over our choice of a formalism, the library structure and access, matching of signatures and equations, and on through the course of the project and its problems that we had encountered, our conclusions and possible further work.

The comments from the reviewers were extensive and detailed. Frankly, they were very critical. Towards the end of the meeting, one member asked how long the study had been. The answer: 390 person-days. The reviewers' criticism turned to some astonishment: they all felt that for a study of this scope, far longer should have been scheduled. Their criticism turned on to the EC for having let the project on such a limited scale of time and effort. It was the chairman's turn to defend the project. It was a feasibility study and as such a negative conclusion of "this cannot be done" was a legitimate and useful result. If it revealed research problems, that was a result they could live with. The Council of the European Communities intended to launch a pre-competitive work plan for ESPRIT. The study has given them some hints about research in the area. More rewriting of the final report was desirable than could be done at the end of the project, but something useful would result if the first part is reworked.

So we rewrote parts of the final report and delivered a final draft in April 1984. If the technical results of the project were disappointing, this was probably because we had been too ambitious. At least, unlike the majority of software-related projects, it was delivered on time and to budget: we had spent 99,405 out of our allowed 100,000 ECU.

ITT had by now completely shed STC as an owned company. STL was a part of STC. Some of us wondered if ITT top management had fully realised that in divesting themselves of STC, they were losing one of their most prestigious research laboratories. We no longer made annual cases for funding to ITT headquarters. Instead we had to make cases for STC funding, and we were free to seek funds and grants from external bodies, provided they were not direct business competitors of STC; and even then, provided we were not giving away company "secrets", there was some leeway. We had already successfully bid for EC funding with the ADT project. Other possible sources of funds were the UK DTI, RSRE and BT. The

successful completion of the Augusta project gave us and the other members of the consortium the momentum of enthusiasm to continue with related investigative work. We made a proposal for a software development method, aimed at Ada programming. This would be based on abstract data types, because the language constructs in Ada called packages were strongly inspired by the ADT concepts. Tools supporting this method would form part of an APSE, an Ada Programming Support Environment. We had parallel conversations with the EC to develop a wider range of Ada support tools to populate an APSE, with the knowledge of all parties. We held numerous meetings, exchanged letters of intent, discussed staffing levels, the team leader, and pricing arrangements. We drew up project plans, partitioned the project into tasks and allocated them amongst the proposed participants. After six months, by July 1982 we had switched attention to the CEC, Commission of the European Communities, for seeking funds; the DTI and RSRE, although willing in spirit, did not seem easily to find the mechanisms to channel a grant our way. To get CEC funding under the ESPRIT initiative, we needed a partner from another European country, since an aim of ESPRIT was to foster pan-European technological cooperation. So we approached the Danish Datamatics Centre, DDC, the campus spin-off company that was the brain-child of Dines Bjørner. Dines was one of the originators of VDM. Rudi Meijer in the Commission wrote a letter of encouragement to Dines and STL's commercial man, Alec Bell, worked with Leif Rystrøm, the managing director of the DDC. Soren Prehn would be the technical participant from the DDC. We drafted a new project plan with contributions, budgets and task allocations involving the DDC. By now it was November 1982: we had been running with this proposal for eleven months.

Again nothing transpired. By April 1983, however, the DTI had undergone some reorganisation. There was now a route for applying to them for funds for research into IT. A committee headed by John Alvey established a five year programme of "pre-competitive collaborative research in the enabling technologies of information technology". It was sponsored by the DTI, the MOD, the Science and Engineering Research Council, SERC, and industry. The government agencies would provide at least half of the £350 million budget over at least five years. This programme became known as the Alvey programme and the part of the DTI who managed it, the Alvey Directorate. Our proposal needed redrafting. For example we had to put some emphasis on marketing prospects. Again, no funded project resulted. But the Alvey programme was to become an important influence on UK research and development into IT.

During the CEC-funded ADT study, one problem that hampered us was that Abstract Data Types are often composite, especially those of any appreciable size. It is natural to compose a complex ADT out of smaller ones. But there can be several different ways of combining different component ADTs to produce the same composite one. Out of this problem arose our

interest in Category Theory. Category Theory gave us a language for defining the general relationships between components and composites ("injection") and for deriving composites from their components ("pushouts").

Category Theory is a branch of mathematics initiated in the 1940s by Samuel Eilenberg and Saunders Mac Lane. It is an advanced branch of mathematics: despite having a degree in the subject, I had not heard of it before and did not recall there being any course in Category Theory offered even in the post-graduate Part III of the Cambridge Mathematical Tripos, in the early 1960s. Since the days of Georg Cantor in the nineteenth century and David Hilbert in the 1900s, set theory had been perceived as the foundation for mathematics. Everything could be related to and re-expressed as sets. For the next half century set theory was regarded as the most fundamental concept of mathematics. In the 1930s a group of French mathematicians working under the nom de plume of Nicolas Bourbaki attempted to produce a fully axiomatised presentation of the whole of mathematics. This massive task extended over forty years, producing many volumes, but Volume 1 was devoted to sets and is still influential today. The modern formal specification language B[2] is based on set theory and shows, I believe, a marked inheritance from Bourbaki.

The emergence of Category Theory changed the perspective of set theory being the foundation of mathematics. Already that foundation had been rocked by Gödel in 1931 showing that no system based on finitary methods could produce a complete axiomatisation of the arithmetic of the familiar whole numbers[3]. Category Theory enabled one to consider the collection of all sets and all functions between sets and other "large" collections without falling foul of Russell's paradox. While avoiding Russell's paradox in itself is not of obvious interest to computer science, the more general approach to functions has particular advantages. In set theory, a function f can be characterised by its graph, which is the set of pairs of values $<x, y>$ where $y = f(x)$. In tax tables the effect of applying a formula to a sum of money is illustrated by listing results of the formula application next to the input value. This is a tabular form of the graph of the function. But in the context of computing it would usually be very cumbersome to represent a function using a table. The distinction between the tabular or pair-wise representation and the actual notion of an operation is prominent: Category Theory returns to the sense of a function being an operation that is applied to an argument, the input value, a member of the *domain* of the function.

Category Theory gives us another benefit when used to model programming languages. It would be possible to define and write a compiler for a language that allowed so-called generic data-types and functions. Most programming languages allow conditional expressions for all permitted data-types:

```
if p then exp1 else exp2
```

[2] See Abrial 1996.
[3] See Nagel and Newman 1959.

With the current methods of defining program language semantics, the meaning of this construct has to be defined for each principal data-type, whereas in fact its sense is type independent. Category Theory could give a way of providing a generic semantic definition, to match the generic nature of the construct. In a language that allowed generic types and functions, the above expression could occur in the defining body of a function, where the arguments exp1 and exp2, perhaps provided as input parameters of the function, were of any type. It would be difficult to define the semantics of such a function using a formalism that did not pay at least implicit reference to Category Theory.

In the ADT study we tried to use the ideas of Category Theory to give us a handle on the process of defining larger data-types as combinations of smaller ones and manipulating them. I think that categories could also come in useful for treating the whole idea of modularity, where one compiles different modules of a system separately and combines them after compilation. The usual present-day theories of semantics do not adequately cover this issue.

So Will Harwood, Paul Taylor and I learned about Category Theory and tried to apply it to the ADT study problem. I was very much the neophyte; Will and Paul were my tutors. One regular conference covering the more theoretical aspects of computing was the Mathematical Foundations of Computer Science. The tenth symposium was to take place in Štrbské Pleso in Czechoslovakia at the beginning of September. I decided to go to it. Štrbské Pleso, now in Slovakia, was at the foot of the High Tatra mountains near the border with Poland. In 1981 Czechoslovakia was still an eastern bloc, soviet allied country and its economic prosperity was frugal. Hotel accommodation was scarce and we had to double up in twin-bedded rooms. I shared with a polite young American mathematician. At least we had a common language. There was only one other British delegate, Leslie Valiant from the University of Edinburgh, who was giving a paper. In fact, almost all the delegates were presenting papers, and the presentations took place on a rigid time-scale in multiple parallel sessions. A few people I knew were there, including Dines Bjørner who was one of the very few others not presenting. The mountains were spectacular and one afternoon was given over to a conference "social event" in which a local mountain guide led a few of us over a snow covered pass. Chains were embedded into the rocks in places to help ascent and descent. The papers presented at the conference were arcane and I got the impression that the main motive for the event was to enable more publications, a metric of academic success. At the conference dinner, some other delegates sharing my table joked about a colleague of theirs. He had apparently published 60 papers in the past year. That is more than one per week, a well nigh impossible task unless one is repeating reports of the same work or riding on the backs of junior colleagues. The academic imperative of "publish or perish" continues today, if anything even more strongly. On my way to the airport at the end of the conference I had a meal in a restaurant. There was limited wine on the menu, and I was the only diner drinking any. I remember that a litre bottle of vodka from a food store cost the equivalent of about 40 pence. At the airport in

Prague there were no luggage carousels. Instead lines of men in blue overalls laboriously passed travellers' cases from hand to hand. A plane landed or took off about once every twenty minutes. I flew to Prague again nine years later in 1990. By then the airport had changed dramatically and had all the usual modern features.

I have mentioned several times how the theoretical analysis of concurrency in software systems was a compelling issue. Several formalisms could model concurrency, each enthusiastically championed by its proponents, but there was no clear way of comparing them or determining which was the best to use in a given application. The Software Research group at STL decided to host a workshop and invite the champions of the different approaches. We devised ten sample problems in concurrency and sent them in advance to academic researchers, inviting them to try out their techniques on them. We received 27 solutions in advance and another six were produced during the workshop itself. After all the solutions were presented, we had a debriefing session during which we hoped to make some useful comparisons between the different techniques. We held the workshop in Cambridge in September 1983; this was during the university vacation and all of us stayed in rooms in Clare College. We were able to obtain funding for the academics' travel from the SERC. After the event Will Harwood, Mel Jackson, Mike Wray and I put together a volume of proceedings which were published by Springer Verlag in the LNCS – Lecture Notes in Computer Science – series[4]. Bernie Cohen and Paul Taylor were also on the organising committee. In addition to the academics, we invited eight participants from some of the bigger industrial players, British Telecom, Central Electricity Generating Board, GEC, ICL, MJSL, Software Sciences, and Systems Designers. Organising this workshop took a considerable amount of effort. We appointed rapporteurs for the sessions to report back on the solutions and plenary discussions, wrote guidelines for them, organised projection equipment, notice boards and stationery, wrote a justification to apply for the SERC funding, and as well as the final proceedings, wrote following reports for the SERC and for our own management. I had to analyse and approve the bill for accommodation from Clare College – it was detailed down to the last teabag.

The whole task of planning the workshop, holding it and drawing it to a conclusion took some 14 months, which is pretty normal for an event of this kind. Producing the proceedings from contributions written by multiple authors was a considerable task. In those days there was such a variety of word processing and other text preparation tools that it was impossible to impose a standard format on all the authors. Nowadays every journal or book series has its own down-loadable text style that intending authors use to create a uniform compatible format. Then we had to rely on camera-ready copy. Authors' texts had to be proof-read, corrected, page numbers played with and so on. We wrote an introduction and conclusions,

[4] See Denvir et al, 1985.

no mean task. I felt that we should circulate these amongst the academic contributors for comment in advance. Only one had criticisms of my conclusions, and I think my modifications met his objections. Was the workshop useful? I definitely think so. While we found no philosopher's stone, we made progress. I could compare it to the conference on concurrency held in Evian in 1979[5]. There most of the presenters simply stood up and explained their theories. In our workshop there was much more interaction; providing a common set of problems for the participants to solve seemed a good framework for encouraging discussions. Afterwards we felt more confident about choosing a formalism for modelling concurrency and as a basis for a development method. We had a good idea of the strengths and weaknesses of each individual method in terms of abstraction, manipulability, ability to make provable deductions of properties, and ability to hide, decompose, structure and refine to a more reified design.

The managing director of STL, Bernie Mills, stepped down on 1st July 1983. He had ruffled some feathers during his term of office. The previous MD had frosted glass doors to his office suite. When Mills moved in he had the doors replaced with heavy solid wooden ones. He lowered the "delegation of authority", this was the amount of money one could authorise to be spent on purchase orders and the like, of a whole range of middle managers at a stroke. His only stated reason was that they did not need it. He communicated this by instructing a junior clerk in the accounts department to send a memo. Many of the managers did not realise what had happened: the memo was scrappy and not clearly expressed. They discovered only when clerks in the purchasing department began bouncing their purchase orders. Shortly before he left, Bernie Mills walked down the corridor past my office. I always kept my door open unless I was having a meeting. He saw me, doubled back and dropped in. After some small-talk, I said, "I expect, like all the rest of us, you are looking forward to your retirement." If he noticed my deliberate ambiguity, he did not show it. With the departure of Bernie Mills, there came substantial reorganisations in the upper reaches of the company. We began to notice still more changes of the kind that began under his regime. The authority of middle managers slowly diminished. When we belonged to ITT, rules could be bent occasionally; if one showed initiative in pursuing the benefit of the company, one was rewarded. Now, rules and bureaucracy were paramount. STL slowly became a less pleasant place to work.

VDM courses continued to be a significant part of our technology transfer effort. We had produced and delivered a one-day course for managers and a one-week course for software engineers. Next, we developed an advanced workshop. There we covered the more intricate

[5] See Khan 1979.

features of the VDM language, and some of the more advanced aspects of the theory underlying it. VDM is based on set theory and logic. This mathematical basis enables one to prove properties of the specification, and of programs which fulfil the requirements that it specifies. But any proof written by a human can contain errors. A lot of work had been done on so called mechanical theorem provers; these are computer programs which can generate a mathematical proof of an assertion written in symbolic logic, or which can check a proof that is provided to it. These mechanically generated or checked proofs have to be in far more intricate detail than the proofs that are in the usual mathematical tradition. Only in the discipline of mathematical logic does one find the same level of rigour. For such rigorous proofs, the theory in which the propositions are stated have to have a consistent set of axioms, sufficient to enable proofs of useful theorems. There are several alternative systems of axioms for set theory. The two most usual ones are NBG and ZF. In the mid-1920s John von Neumann[6] proposed a system of axioms. Later Paul Bernays and Kurt Gödel further developed von Neumann's system and the result became known as NBG. In 1901 Ernst Zemelo provided a slightly different axiom system. Again, in 1922 this was extended by Abraham Fraenkel, and the resulting system is known as ZF. NBG and ZF are very similar, and in certain circumstances can be shown to be equivalent. In general terms the logic of VDM is based on ZF. So in the advanced VDM course, we included an explanation of ZF set theory[7].

One of the purposes of having a VDM specification is to prove that a program is correct. One also should prove that the specification itself is consistent. A newcomer to the topic could be unsure about what propositions exactly one needs to prove in order to demonstrate consistency and correctness. These propositions we called "proof obligations": one is obliged to prove them in order to demonstrate consistency and correctness. Our advanced course covered these proof obligations, as well as various other topics such as more detail on data types and desirable "style" of writing specifications. We wrote a manual for course lecturers, and after a lot of discussion, an outline of qualifications that lecturers needed.

In VDM and, indeed, in computing in general, functions are often partial. That means that they only produce a defined result when applied to some of the values of the type of their domain. Most functions in mathematics are total, producing a result for every value. Some are partial, however; one cannot divide a number by zero, for example, or obtain a real result by taking the square root of a negative number. With VDM specifications, there can be many logical expressions in which a partial function is applied to an argument. To produce logical proofs of correctness and consistency, the logic has to be able to cater for partial functions. Cliff Jones had been doing research into this subject, abbreviated LPF, and with his cooperation we included LPF in the advanced course. In simple terms, in classical and other

[6]John Von Neumann proposed a machine architecture that was the blueprint for all subsequent computers: see Von Neumann et al 1947.
[7] See Devlin 1994.

conventional logic a proposition can only be true or false. In LPF, a proposition can be true, false or undefined.

Our VDM courses had now been given to quite a variety of organisations, who were beginning to use the method. IDEC started to specify part of the new TX4 telephone exchange software in VDM. The government's DTI sponsored a project to develop support tools for VDM under the Alvey programme. We held a VDM users' conference in which we displayed the latest developments and activities and shared case studies. We tried to pull together the several variations in the VDM language that different developers and researchers were using and produce a standard, to be submitted to ISO, the International Standards Organisation. The EC funded project, RAISE, was beginning, which was to develop a considerable extension of VDM. These were topics to be aired at the users' conference. A coordination committee with members from several companies and academic institutions discussed the curriculum of the courses. We approached the NCC, National Computing Centre, with a view to giving courses to the "public", where anyone could reserve a place, rather than their being invariably in-house.

The desire to prove programs correct was not the only reason why programmers were interested in symbolic logic. Artificial intelligence is the attempt to duplicate various human cognitive processes with a computer. An important one of these enterprises is the understanding of human language. There are many aspects to this. Sentences in the language, with all their inherent ambiguity, have to be parsed. One needs a means of representing the knowledge denoted by the language script. To fully explore the meaning of a script requires a logical deductive system. Another topic within AI is expert systems. Here the workings of human experts are recorded over many trials and recorded in a database. The expert system analyses all the data and creates an automated "expert". This technique has applications in medical diagnosis and many other human skills that take a long time to learn and which cannot be completely defined.

At STL Nigel Steele, a member of the software research group, had for some time been pursuing his own research programme in AI. He had reserved himself a place at the 1984 International Symposium on Logic Programming, to be held in Atlantic City. Some time later he decided to move on, and although he was still working out a period of three months' notice, STL would not allow him to attend the symposium; they were unwilling to fund the travel and fee for someone who was not going to be an employee for much longer. I thought this was a touch short-sighted and churlish, for he was the one who would benefit most from attending, owing to his specialist expertise. However, the company was not to be moved and, since the booking was already made, I was sent instead.

There were a large number of attendees at this symposium: 350. The organisers had expected about 100. Most of the papers were related to Prolog, a programming language using logical expressions rather than the usual imperative commands. The Prolog interpreter is able to draw deductions from the logical expressions and prove a desired goal. The logical expressions have to be of a restricted form, called Horn clauses. These are named after the logician Alfred Horn who pointed out their significance in 1951. A characteristic of Prolog programs is that the interpreter rapidly devours large quantities of computer processing time and memory space. The interpreter uses a technique called “unification” to draw inferences from the logical expressions in a program. Prolog had been invented by Alain Colmeraur twelve years earlier in 1972. The last leg of my journey to Atlantic City was in a small aircraft of about eight seats. He was one of the other passengers and all the remainder were attending the symposium. The other delegates in the plane showed him a respect verging on awe. I learned that there were many organisations at the symposium advertising for staff with expertise in logic programming, AI and expert systems, and some specialist firms were concerned exclusively with the area.

Many of the papers in the symposium were devoted to techniques of reducing the amount of time and space, known as the complexity, required by programs. Some extended the language until it was scarcely recognisable; others attempted to overcome the inherent computational complexity of logic programs by taking advantage of massively parallel architectures that were only on the drawing board at the time. I noted that the state of hardware architecture design still seemed to be in the same primitive condition as software used to be in the nineteen-fifties,oriented to the machine rather than the function and lacking in abstraction. I felt that quite a few of the papers in the symposium would be of interest to our EST and AI projects.

There was a notable interest from Japan in the symposium. Some 25% of the papers were from Japanese institutions including a recently established Institute for New Generation Computer Technology. For this reason I was highly irritated when a president of IBM flew in to give an after dinner talk at the conference dinner. His theme was “The Japanese Threat”. He was referring to commercial competition, but I thought he could have registered the international nature of the event before choosing his topic. The Japanese delegates applauded politely at the end of his speech.

Artificial Intelligence is the research topic that attempts to replicate human thought processes by computer. One notable cerebral capability of human beings is the ability to reason, to construct a logical argument and reach a conclusion. Every process in a computer program has to be expressed in abstract symbols. So one topic, among many others, of great interest to researchers in AI is automated proving. The same topic is also of great interest to software

engineers pursuing formal methods, for proving that a program is a correct realisation of its specification would go a long way to reduce errors and faulty behaviour in software. A small but vigorous research team in STL had been pursuing its own efforts in this direction. EST was a project, led by Will Harwood, which was investigating the construction of a proof system based on the logical rules of program proof, starting from the rules of equational reasoning, but moving on to other logics. The ITT laboratories in Madrid were interested in obtaining a copy of EST to conduct their own researches. EST, however, was a prototype; Will's vision was to construct a generic engine which could be parametrised with the codified rules of a logic, and could thence be instantiated as a proof suite for that logic. This was rather like the step from building a compiler for an individual language to building a parser-generator, which can be parametrised with the syntax rules of any language. This aim, to build a logic-based proof system generator, was the objective of NIMBUS, a successor project to EST.

Tony Hoare, who headed the Oxford University Programming Research Group and devised CSP, the formalism for modelling communicating sequential processes, had been made a Fellow of the Royal Society. I think it is true to say that he was the first person to be so honoured for contributions to theoretical aspects of software, although Maurice Wilkes, the director of the Cambridge Computer Laboratory was a FRS before him. Maurice Wilkes's principal achievements were in computer hardware. He designed and oversaw the construction of one of the first stored program computers, the EDSAC, which was completed and operated successfully from May 1949. After its successor EDSAC 2, the next computer in his laboratory was the Titan, designed and installed in conjunction with Ferranti and a "sister" machine to the London Atlas. He is also credited with several developments which paved the way for high level languages.

So software research now had a representative in the Royal Society. In February 1984 the RS held a "meeting for discussion" on Mathematical Logic and Programming Languages. Tony Hoare was one of three organisers, the others being Michael Atiyah and J. C. Shepherdson. Sir Michael Atiyah, also an FRS, later became the president of the Royal Society. He had, incidentally, lectured on linear algebra in the Cambridge mathematics tripos during my undergraduate years. The two-day meeting at the Royal Society was very stimulating. The speakers comprised some of the most well known names in computer science. I had encountered several of these, Tony Hoare himself, Robin Milner who developed CCS and LCF, the logic of computable functions, Bob Kowalski from Imperial College London, who was an authority on logic programming, and others. But I heard two other speakers for the first time, who had been remarkably influential in theoretical computer science. Dana Scott had developed the kind of domain theory that provides the foundation for recursive data

types[8]. Edsger Dijkstra had devised the notion of guarded commands[9] and published *A Discipline of Programming*[10] which had so inspired me in the mid seventies. As would be expected, logic applied to programming languages was the main theme of the meeting. Proving programs correct can be computationally time-consuming and difficult. There is a trade-off between expressiveness of the logic language and the efficiency of its "execution", that is the process of using theorem provers to deduce consequences from premises. One has a choice between developing proof theories and algorithms, and developing programming languages that are not imperative but are more conducive to constructing proofs. Some of the talks focussed on the former and others on the latter. Robin Milner's talk described LCF, the Logic of Computable Functions[11], in which strategies and tactics for proofs can be described. Kowalski described proof techniques for Horn clause[12] logic. D I Good described a verification environment called GYPSY for developing programs. It included a verification condition generator. Verification conditions are much the same as proof obligations already mentioned. GYPSY also had a proof checker that checked the validity of proofs. Most of the other talks sought ways of expressing programs so that proofs fall out on the way, so to speak. Functional programming languages are close to specification languages like Z and VDM, but are executable. The proof of correctness follows the construction of the program. I came across one of the speakers for the first time, Per Martin-Löf. His approach is that the intuitionistic theory of types and constructive mathematics can be viewed as a programming language. The inference rules of the type theory are themselves the rules of correct synthesis of programs. So the correctness of a program written in the theory of types is proved formally at the same time as the program is synthesised. This was to me a very novel way of looking at programming. Martin-Löf showed how the axioms of set theory are analogous, indeed are the same apart from differences in syntax, as those of intuitionistic logic.

From the sublime to the, if not ridiculous, severely practical, in April the same year (1984) the first spell checker came our way. It made a half hearted attempt to distinguish between British and American spelling, but was incomplete in many ways. It did not recognise many words, some of them technical, others prosaic. It recognised the electronic busses but not the vehicular buses; homological but not heterological; it did not recognise instantiate, powerset, coproduct, codomain, morphism, bijection or more everyday words like lorry, puce, watertight, scruffy. But it was the first spell checker I had come across and it was almost usable.

[8] See Scott 1976.
[9] See Dijkstra 1975.
[10] See Dijkstra 1976.
[11] See Gordon et al, 1979.
[12] See Horn 1951.

FACS, Formal Aspects of Computing Science, a special interest group of the British Computer Society, was founded in 1978. I had been attending their meetings for five years, since 1979, and in 1984 Dan Simpson, its chairman, asked me to join their committee. FACS had established working relationships with the London Mathematical Society, the Association of Mathematics and its Applications, and the European Association for Theoretical Computer Science. I was gratified to be invited to join their committee, which organised meetings and generally ran the group. In my first committee meeting with them, because of my previous experience with the DTI and the Augusta study, my name was put forward to be the FACS representative in a BCS task force liaising with the Alvey Directorate. We discussed several future meetings on topics such as Mathematics for Computing, Petri Nets, OBJ – an algebraic specification language, ML – a functional programming language, Knuth Bendix and Unification algorithms, and HOPE – another functional programming language that was a predecessor to the later languages Miranda and Haskell. This was the first of many FACS committee meetings that I would attend over the next nineteen years. FACS was something of a ginger group that tried to stimulate new ideas in applying computer science theory to practical software development.

In 1983 the British firm INMOS designed and built a computing microprocessor with a concurrent architecture called the "transputer". Having a concurrent architecture meant that it could carry out several computations in parallel, that is, using several CPUs working together simultaneously. The transputer was designed to work with a parallel programming language called OCCAM, designed by David May of INMOS in association with the Oxford PRG. OCCAM was in turn based on Tony Hoare's CSP formal language and shared many of its features. Indeed it could be said to be an executable version of CSP.[W]

Bill Roscoe of the PRG developed a semantic definition of OCCAM. Although in the long run neither the transputer nor OCCAM could be said to be grand commercial successes, they were both in their way very influential over later computer architectures and principles of concurrent program language semantics.

Shortly before ITT shed STC from its conglomerate family, we had to contribute to one last review in the USA. I was asked to give a talk on 10th April 1984. The next day my children, young teenagers then, were performing with their school orchestra in the Albert Hall in London, a prestigious occasion. I decided I could just fit it all in. I flew out, gave my talk on VDM, Ada and how they could work together, had a short nap on my hotel bed for an hour and flew back. That was the only time I took a day trip to the USA.

In 1984 Dines Bjørner, Professor at the Technical University of Denmark and founder of the Danish Datamatics Centre, suggested that we should get together with several other institutions to submit a proposal to the European Commission's ESPRIT initiative to develop a new practical formal method for software engineering that incorporated all the advantages of various existing languages and methods. We had numerous meetings and discussions. The result was the RAISE project---Rigorous Approach to Industrial Software Engineering. Manchester University's computer science department under Professor Cliff Jones were also involved at the early stages of getting the proposal together. The central components of project was the RAISE Specification Language, RSL, and its support tools, together with a "method" for using these in software development. The language would incorporate all the features of VDM plus facilities for concurrency, so that systems with concurrently executing components could be specified. The method would include guidelines for discerning the requirements of the system and managing the project, as well as the usual methodology of using a formal specification and refining it to an executable implementation.

Incorporating concurrent features into the specification language required us to combine features from different formal specification languages: theories had to be combined. For large systems of software, managing size needed a way of bringing specification modules together. To reach these objectives, the project needed to start with a theory study. The DDC held a preliminary workshop on combining specification methods. Many academic experts contributed to this: Michel Sintzoff, Dana Scott, Gordon Plotkin, Laurence Paulson, Ugo Montanari, Jim Thatcher, Willem P. de Roever, Albert Meyer, Hans Langmaack, Andrzej Blikle, Jeannette Wing, Hartmut Ehrig, Eric Hehner, O-J Dahl, Allessandro Fantechi, Matthew Hennessey, Manfred Broy, Mads Tofte, Peter Mosses, Otthein Herzog. Rod Burstall agreed to be a consultant to the project. Dines had an astonishing list of contacts and a deal of influence.

If we were to convince other organisations, especially industrial ones, that RAISE was worth using, some demonstrator projects would be useful. The proposal needed to lay out motivation for the project, its objectives, and a strategy and work-plan for carrying it out. Then there were the more administrative issues to be sorted out like a partnership agreement and the sharing of intellectual property rights. Each partner should be free to market the results but also to use them for free within their organisations. We aimed to start the project at the beginning of January 1985. Within STL we had to convince our management and peers that the project harmonised with company research strategy. There was potential conflict with current internal projects. If we committed to RAISE, there was a danger that STL's NIMBUS project, a successor to EST, could collapse with staff being diverted to RAISE; it is organisationally imprudent to have two projects with many similar aims. The company had invested considerably in EST and NIMBUS. On the other hand, the DDC had produced an

ADA compiler with EEC funding and were offering a licence to IDEC, an STC company close to STL, at 50% discount.

The biggest investment which a company like STC makes is in teams of people who can work together coherently. This occurs when a team can identify with a project's objectives. I privately reckoned that if the NIMBUS team were reassigned to RAISE, several of its members would leave. I could see four possible solutions: abandon participation in RAISE; abandon NIMBUS and collaborate wholeheartedly in RAISE; divide STC contribution to RAISE into two: STL and IDEC, with IDEC providing the lion's share so that we could pursue both projects; acquire more staff so that again we could do both.

I felt that we could not abandon RAISE; too many of our company objectives would not be met. I thought we should go for solution 3, and see if the DDC would accept sacrificing some commitment from STL but using consultative style contributions from us and commitment from IDEC. I knew that Cliff was not in favour of this solution, but I thought it should be put to the DDC and their reaction tested. Failing that, we could explore solution 4. But that was probably not feasible: STC had made 390 staff redundant in the last year, having overstretched their financial resources by recently acquiring ICL, so the company climate was hardly conducive to recruitment.

If a choice had to be made between abandoning RAISE or NIMBUS, STC would have to decide whether it wanted to invest in RAISE, a substantial enhancement of VDM, over the next five years, or continue investing in NIMBUS, which was more long term and might be in a prototype stage in three years' time. I believed that abandoning NIMBUS was not a real solution. While one could assign the NIMBUS staff to RAISE, one could not be sure of "assigning" their commitment to it. I felt that one of the "compromise" solutions should be pursued as vigorously as possible. The worst thing that could happen was to be indecisive and follow the course of least resistance, i.e. to assign staff to RAISE without their proper commitment and to pretend that the NIMBUS project can continue unaffected. If that was allowed, all objectives would be missed. I sent a memo to director level management listing the possible solutions, my concerns and my recommendations.

By 1985 ICL had joined the RAISE consortium. We held workshops on the theoretical problems of combining specifications and on aspects of VDM. By the end of February the European Commission had approved the proposal and the project was under way. As I feared, no firm decision was made about the dilemma between pursuing RAISE or NIMBUS. Will Harwood who led the EST and NIMBUS projects left STL and set up his own small organisation under the wing of Imperial Software Technology, a campus company sprung from Imperial College's Department of Computing. There was a severe danger of other members of his team following him and leaving STL with a drastic reduction of staff with experience in formal methods. The scenario I had foreseen as worst case had happened:

inaction over RAISE-NIMBUS. Again I alerted management up to director level of the situation.

There were other changes in STL, most likely resulting from ructions following STC's acquisition of ICL and subsequent redundancies. STC sold off its prestigious headquarters building at 190 The Strand: not a good indicator. Reorganisations in STL produced a vacancy for our own senior division manager position. It was advertised nationally, and locally on a company notice board. I decided to apply for it myself. Since the management had made, in my view, serious policy mistakes over RAISE-NIMBUS affecting all of us, I thought that if I was higher up the management chain, I might at least limit the damage and prevent further similar errors. I also thought that the move might help the morale of the staff around me to recover. I obtained a copy of the job description and found that I exceeded all the qualifying requirements by considerable margins. I applied and had a one-to-one interview with our technical director. I heard nothing until one of our research staff greeted me with the words, "Have you heard the good news?" Valerie Downes from Imperial College, whom we all knew quite well and whose abilities we respected, had got the job. This was the first and only notification I ever received that my own application had been unsuccessful.

More RAISE workshops took place over the next nine months, in which we carried out intense technical work, as well as in between times. In May I was appointed joint system architect alongside Dines Bjørner. I felt honoured and gratified. A technical board and a management board were set up. The technical board sorted out more strategic technical matters like how we should use external consultants, schedules for workshops and more specific technical decisions. The management board dealt mainly with the project's relations with the European Commission, the issue of reports to them and deliverables, costings and other such items.

RAISE attempted to define the actual process of developing software by a formal model, expressed in mathematical terms. Each step in the development produced a new product, a specification or other document, that, to be valid, had to bear a definitive relationship to its antecedents. The set of documents in a project together with the relationships between them formed a Project Graph, and the process of developing it was termed a Meta-program. We attempted to incorporate non-functional requirements into the whole concept too, typically performance requirements such as timing constraints and ability to handle capacities of data. A project in its final form could be seen as a rationally reconstructed history. RAISE would include a raft of software tools to support the method and concepts underlying it.

The earliest part of a software development is the elicitation and analysis of the requirements for the project. This has always been an elusive part of the development cycle, because it necessarily involves human intuition. The vast majority of software disasters have resulted from the delivered software not meeting the actual requirements of its working environment;

disasters rarely result from the software being technically wrong, although a few have. See, for example, Robert L Glass's book, *Software Runaways*. Formal modelling could be used to help understand and analyse requirements through an interplay between the examination of conjectures expressed as formal models, intuitive creative work, constructing lists of questions, experiments, abstraction, generalisation, and recognising common and orthogonal features.

The RAISE project continued to a successful conclusion after I eventually left STL. Its products and support tools were developed and continue to be available, used and marketed. Other institutions became involved in RAISE after the end of the EEC funded development phase, notably Terma A/S and the United Nations University in Macau. The Danish Datamatics Centre transformed into the independent Danish company IFAD, and the involvement in RAISE moved from them to Terma. Dines Bjørner for several years transferred to UNU and established a thriving computer science department there. His recent work on domain engineering is, in my own perspective, a natural evolution from the work on meta-programs that we did in the RAISE project.

In parallel with the RAISE project, there were, as always, a host of other smaller activities engaging my attention. The Alvey Directorate used to work with the other government funding agency, the SERC, to foster industrial and academic cooperation. These could take the form of joint projects with Alvey part-funding the industrial contribution and SERC funding the academic part. But academic research projects were also encouraged to engage some industrial involvement by appointing an "Uncle" from industry. The idea was for the industrial partner to take an advisory rôle, attending meetings with the project every three or six months and providing some industrial perspective to help keep in sight some prospect of eventual exploitation. Calling the rôle an "Uncle" was rather quaint, to my mind, having evocations of being, well, avuncular, rich and kindly perhaps. I received a phone call from Robin Milner at Edinburgh University asking me if STL could fill this rôle. The project was to produce a BS standard version of ML, a functional programming language developed by Edinburgh and Cambridge University. Apparently they had asked ICL first, but ICL had been more possessive over exploitation rights and so forth than Edinburgh had been willing to agree. Now they were looking for someone who was not a computer manufacturer, to avoid the same problem. I was pleased to show willing.

The one ITT research laboratory in the USA, the Advanced Technology Center, were keen to gain as much information from STL before the final date beyond which STC was no longer an ITT company. ITT had mostly done most of its research in European laboratories, STL, LCT in Versailles and ITTLS in Madrid, for the pragmatic reason that, at the time, British

and European labour rates were cheaper than US ones and research was labour intensive. We agreed to a visit from ATC staff in August of 1984. ITT had recently enlarged the ATC with some fifty new staff, all PhDs and involved in computer software in some way. ITT's hope was that they could put all these fine brains together and stand back to watch the resulting intellectual fireworks. This did not work out quite as anticipated. Certainly, the staff at the ATC included a lot of talent, but many were somewhat charismatic and idiosyncratic individuals. There was apparently not much in the way of structures set up for them to work together cooperatively.

In September 1984 a research group in STL developed one of the first LCD flat screens. It used "semectic" LCDs. The screen required 150 volts to keep it charged but otherwise used very little power, about two to three watts. Its response was slow, too slow for television and slowish for word processing. This first prototype produced monochromatic images only. The size was 720 by 400 dots, giving bit mapped images and 24 rows of 80 characters, using a character matrix of 9 by 16 dots. It had a capacitive overlay for touch entry and control. The active matrix controlling the LCD screen was a silicon slice 10 to 15 cm wide.

The telecommunications industry was perhaps the major recruiter of computer science graduates in the U.K. Industry had a considerable interest in the skills and knowledge with which graduates emerged from their degree courses, and hence in the content of the courses. The universities on the other hand had an interest in providing courses that equipped graduates well in their search for post-graduate employment. This mutual interest resulted in the formation of the Joint Curriculum Development team. The industrial members were STL, GEC and Plessey. We met at intervals to discuss details of undergraduate and graduate courses and tried to interest the Alvey Directorate.

Dan Simpson, who had worked at Elliott's in Borehamwood, moved first to academia, Sheffield Polytechnic, which later became Sheffield Hallam University, and then to the Alvey Directorate. He was a founding member of BCS FACS and as its chairman had asked me a little earlier in 1984 if I would participate in the FACS committee. He had a strong interest in formal methods and later in the same year helped to stimulate a project exploring the application of formal methods to protocols. In telecommunications a protocol is the rules and meaning of a specialised language for introducing the context of an electronic message. For example, today when one receives a telephone call one's equipment can display the number of the caller. That number is encoded together with the notification of the incoming call. So that this information is universally understood by all equipment, there have to be agreed rules about the format of the information and its interpretation. These rules form national and

international standards, of which the CCITT is the international standardisation authority. With the onset of digital telephony, protocols were becoming more numerous and of burgeoning importance. They are called protocols because they are like the rules of etiquette, of saying “how do you do?” that two pieces of computational equipment exchange before getting on to the real exchange of information.

So the seeds of an Alvey funded project, Formal Methods Applied to Protocols, FORMAP, were sown. In October 1984 Dan, together with Howard Nichols of the Alvey Directorate, held a meeting with British Telecom, ICL, GEC and STL to see if we could put a proposal forward. There were already Alvey projects working on Z, VDM and ML, but none on CCS or CSP, two of the principal specification languages for modelling concurrent processes. Protocols almost invariably denoted some measure of concurrent behaviour. All of us, David Freestone from BT, Ken Turner from ICL, Peter Scharbach from GEC and I, were enthusiastic about the idea. Our first steps were to draw up a collaboration agreement between our companies, send it to the Alvey Directorate and expect a letter of intent from them in reply. After that would follow a work-plan and cost estimates. We expected to get started in the new year.

E-mail was just beginning. It originated in the US as the US Department of Defense Advanced Research Projects Agency Network in the 1960s. ARPANET grew gradually through the 1970s until by the early 1980s the number of hosts had reached over 200. Following this lead, the UK SERC pioneered a research network based on the X.25 protocol. It was originally built for communication between academic institutions, and all further and higher education institutions and government research organisations were in time connected to it, schools eventually following. Because of the close links between SERC and the Alvey Directorate, Alvey were able to set up a mailbox supporting email between partners on an Alvey funded project. So it was that our first experience, corporate and individual, of email was on the FORMAP project, rather appropriately since the project focussed on protocols, on which email and its supporting network relied. Compared to today, this email was decidedly clunky and took a great deal of effort and time to set up for the project, but it was there and usable.

The administrative processes for initiating the project were immensely protracted. Alvey's letter of intent was contingent upon their receiving a report from referees. These were delayed. Because protocols were so bound up with standards, we involved the British Standards Institution and the National Physical Laboratory, who work hand in glove with the BSI. Even in April 1985 there was still uncertainty whether Alvey needed separate contracts with each partner or just one with a prime contractor who would subcontract to the others. At least the project work had started and Alvey assured us that time and resources spent could charged in arrears once the proposal had been approved. This kind of contractual delay to a project start was to be entirely typical of both Alvey and CEC funded projects. We eventually

received the letter of intent on 25th March, backdated to the 4th. Meanwhile we had our own managerial arrangements to do: setting up a management and a technical committee, drafting template subcontracts to consultants and agreeing on our own "protocols" for project communications: email for short documents, numbering and a classification scheme for project reports and documents.

In mid-March Robert Milne, working for STC, joined the project. I felt that this was something of a coup. Robert had achieved some academic fame by writing one of the first books on formal semantics of programming languages with Christopher Strachey in 1977[13]. He was working on an internal project that had some overlap with FORMAP and wanted to know what the relationship between the two should be. He was expert on higher order logic, algebraic data types and modal logic, all of which were highly relevant to the theoretical foundations of FORMAP.

We began by jointly carrying out a reading of all current papers and research that had a passing relevance to the project and providing short commentaries of them. Between us we trawled through back copies of thirteen different journals and sent for other research reports from the US Government Printing Office and the Science Reference Library in London. To make comparisons easier we all wrote our commentaries in an agreed review format. With a classification scheme and preliminary findings the results formed our first project deliverable in mid-1985. Alvey themselves held an annual conference and they wanted a poster on the project from us for the next one during 25-27th June. At the same time our contractual terms with Alvey were just about to be signed, with separate contracts with each partner and a collaboration agreement, which Alvey wanted to approve too. Of course, since we all worked for large organisations, all these had to be approved by everyone's legal departments, who all had their own agenda of demonstrating to their companies that they were protecting their employers' commercial interests and thereby were justifying their own existences. I have observed then and several times since that this leads company legal departments into long negotiations about clauses protecting against contingencies that could never in reality occur. In the words of the Flanders and Swann song, "it all makes work for the working man (sic) today". In all Alvey contracts the government funded 50% of the industrial costs, and the company funded the other 50%. The government through the SERC funded all of the academic costs. The idea was that the industrial partners invested some of their own money, which would focus their minds on only submitting projects that had a plausible future. STC's central funding constraints meant that we were limited to supplying one person-year per year to the project. We had to work within that limit and allocated half a person year to STL and the same to IDEC. My own involvement would be limited to a few days from time to time, but I would keep abreast of the work. The literature survey would continue throughout the time of the project, updating the first report as new information and perceptions emerged.

[13] See Milne and Strachey, 1977.

Protocols had until then been defined in a very procedural, mechanistic way; there was precious little high level specification in the standards. In my own view, protocol definitions did not describe the key, they described the lock it would open. We found that there was little abstract specification, data structures etc., and that terminology was often loosely defined. Structured, top-down ideas were little in evidence and there was not much indication of deriving design from requirements, at least in the documents available from the protocol design work. So there was an interplay between the techniques available and the approach used by protocol designers, or rather between the lack of techniques and the limitations in approach.

While Alvey agreed to contracts with the separate individual partners, an arrangement desired by some of the companies, they still required a coordinating partner to be responsible for deliverables. The coordinator also had to approve invoices from the other partners, so in effect this became like a single contract with a lead partner in all but name. Again this presaged a subsequent preference of the funding agencies: they far preferred to deal with a single point of contact in the consortium. Alvey nominated an independent Monitoring Officer who attended project management meetings and could attend technical meetings if he or she wished. The MO oversaw the running of the project on behalf of Alvey and would try to keep the objectives of the project related to those of the Alvey software engineering initiative.

In parallel with examining some thirteen standard protocols and their existing definitions we brought each other up to date on a range of formal techniques. We gave each other mutual tutorials on *CCS*, *ACT1*, *LOTOS,* Temporal Logic, *CCS*, and Petri Nets.

By the end of November 1985 the collaboration agreement was being circulated for signature and Alvey had sent a grant letter to all parties, nine months after the project had started and over a year after the ball had begun to roll.

Sheffield Polytechnic, now Sheffield Hallam University, got in touch with me about another Alvey-funded project. I was well advanced with turning the course I had given in STC on discrete mathematics for software engineering into a book. Sheffield were working with the Open University and the BBC to construct a course on mathematics for computing, delivered in modules in the high standard OU format with videos, manuals, tutorials etc., a project very much in sympathy with the aims of my book. As usual, the initial discussions were once again all about legal matters, intellectual property rights of the end product, marketing, equity, who could run public courses. I suggested that equity should be in proportion to each partner's contribution to the development. The OU could market through the National Computing Centre. The industrial partners would be ICL and STC, STC working through STL and IDEC. The other academic partner was Hatfield Polytechnic. I dubbed the project

“Polymaths”, but this name never stuck. It came to be called the BEAVER project. In the end STC’s rôle consisted simply of me taking part in the steering committee.

The course resulted in eight modules, Introduction, Sets and Logic, Functions, Recursion, Relations, Logic and Proof, and two Case Studies. The Introduction would be the usual “How to study the course”, motivation for it, and course contents. This project sped along quickly. By October 1985 Hatfield and Sheffield had nearly completed the case studies, modules seven and eight. There remained to complete the final bells and whistles, a tutor's guide, a written rationale for the course, bibliography, glossary of terms, modes of studying it, a profile of expected attendees, expectations of what attendees could do on completion, solutions to exercises, and a script on how vital were the exercises. ICL offered their own tutor guides as a template. My last task was to send comments on the two case studies. The project was complete.

In October 1984 Neil Davis of Systems Designers Limited telephoned me to propose another Alvey project. At that stage in the game formal methods mostly consisted just in notations for writing specifications, semantic definitions of languages and models of computations. Deriving programs from specifications, proving correctness of implementations and proving consistency were all mostly done by hand, although there were programs available to verify proofs and even to generate them. But there were no automated tools specifically tailored to any of the formal specification languages. To make a technique like VDM really useful, some support tools were needed. A project to develop a suite of support tools for VDM would be very useful for the software industry. Manchester University were working on a tool for assisting rigorous software development called MULE, a closely allied objective. It made sense to collaborate with them under the Alvey initiative.

In developing a substantial software project, one will be working on a large collection of connected programs. If using VDM, for each of these programs there will be a VDM specification, hence within the project there will be a library of specifications. At that time, to write and edit a specification on a computer, one would use an all-purpose text editor. Much more useful would be a structured editor, one which knew the syntax of VDM and which would prevent one from writing syntactically incorrect VDM. Another useful tool would be a pretty-printer, which would print out a specification formatted and indented so as to reveal the structure of the script. The library would contain the scripts of specifications, programs and statements of requirements. A configuration control tool to relate different versions of specifications and programs to each other and to track requirements to parts of specifications which fulfilled them would be ideal. Finally, an aid to animating the specifications, replaying the effect of their implementations to the customer, would be most desirable. We had only a general idea of what this last tool would do and how it would work, but these were our first

thoughts on the scope of the project. A structured editor would in particular would release the writer from concerns about lexical conventions and could select between any alternative formats before printing or storing the final text.

We thought that introducing concurrency into VDM would also be of great benefit. Over the next few months we spent some time considering different concurrency techniques, temporal logic, *CCS* and so on, and how well they might combine with VDM: whether any experiments had been done to date, experience so far, whether the technique handled parallelism, degree of abstraction and so on. We also considered making use of the results of the NIMBUS project to construct VDM proof tools. The VDM Tools project continued to November 1985 and beyond, after I had left STL.

At the end of 1984 STL complimented Mel Jackson and me by giving us an award for our contributions to bringing VDM technology into use in STC. The company hosted a reception in the evening, took photos and presented us with cheques for £500. This was a very pleasant, relaxed occasion and good for both our CVs.

For some years now we, along with many researchers in other places, had been concentrating on the process of producing software that was correct, that is, that fulfilled its specification. The languages that we had devised for writing specifications were rather technical, so that most customers were not able to produce them themselves. The model of development that emerged was that the specifier would talk to the customers and discover their requirements, and then frame them as a specification. This would then be played back to the customer, "animated" we used to say, to confirm whether the understanding was right.

It was becoming clear to many people that this process of sorting out the requirements in order to write a specification was the weakest link in the whole software development process. In succinct terms, we had been concentrating on "producing the software right", whereas the biggest problem was "producing the right software". This whole area has grown into a topic called variously "Requirements Engineering", "Requirements Analysis", or "Requirements Elicitation". The customer's requirements usually start off as a number of goals: for example, to automate an existing manual system, to coordinate a disparate set of existing systems, or to assist a process with automated support in some way. From these often vague objectives, the specifier has at the end of the day to produce a specification from which the software will be developed. So part of requirements analysis is the building of a specification that reflects those requirements. What exactly is this process of specification building? What sort of systems were we going to specify? We tried to characterise them. Systems could display modularity, that is, be built of connected parts; they could be state-based, that is having a memory or state that persisted from one activation to the next; their

modules may need to communicate with each other or with their environment, possibly concurrently; they could be time-dependent, non-deterministic or probabilistic.

It was becoming clear to us that, while technology was fairly well defined, the areas of application were usually not so. When devising a specification one is, in effect, defining the technological properties of the required system. The process of devising the specification will in turn reflect back on the understanding of the requirements. One major difficulty in all this is that the presence of a major piece of software will alter the environment in which it is intended to operate. If we were going to have criteria for the success of the various phases of the specification process, they have to be testable. The CORE method, with which we experimented in the Augusta project, analyses requirements by recording different viewpoints of the desired system, and attempts to resolve any clashes.

Specification, then, is a process consisting of cycles of clarifying one's understanding of requirements, building models, formalising them, and testing them. Many of us followed the ideas of Karl Popper, the philosopher of science[14]. For us testing meant the search for a refutation of claims of consistency and fulfilment of requirements. Having built a specification, one needs to understand all its implications, what its behaviour will be when implemented. One can do this by constructing models of it. Often the model will be the implementation. A specification lies at the interface between deriving it and understanding it, in this sense. We needed tools to support both activities. Building models could be a way of crystallising what one is talking about when describing requirements. In this way we should reach a specification that more closely reflects the real requirements. For example, a requirement might be "the system should have a user-friendly interface". Building models may be useful for exploring such requirements where in fact those requiring them are uncertain of what exactly they want.

The search for a good method of analysing requirements continued, and does so to this day. One subsequent project which I was to encounter later was the FOREST project, which took the approach of building formal models of requirements in order to reflect and analyse them. One lesson I have learned is that a large software project should be developed incrementally, with small steps being delivered and piloted in a cumulative fashion. All big bang projects fail!

During my last year at STL my colleagues and I attended plenty of stimulating seminars and other events. We gave a good measure of talks ourselves, too. I gave presentations to Coventry Polytechnic, the Open University and to ICL in Bracknell, where I had worked myself twelve years earlier. My topics were a syllabus for discrete mathematics foundations needed for learning formal methods, approaches to teaching, how to choose a formal method

[14] See Popper 1972.

and concurrency in particular. Mel Jackson, Roger Shaw and I gave a paper[15] at the 1985 TAPSOFT conference detailing our experiences with bringing VDM to STC: our reasons for choosing VDM, our evaluation process, how a formal specification language could be a cognitive tool, the training programme, and further developments. This was a very rewarding conference with papers by many well known names, including Professor Andrei Ershov, a pioneer of computer science from the USSR Academy of Sciences in Novosibirsk, Siberia. A symposium on Program Transformation at Reading University was a little more rarefied. Several aspects of formal methods were the subjects in a BCS FACS meeting. Dr. Hartmut Ehrig talked about algebraic specifications and data types, a topic based on universal algebra, which was pioneered by the mathematician Paul Cohn in the 1960s, although the subject started in its present form in the 1930s. Hartmut Ehrig developed his ideas into a book on Algebraic Specification which I found immensely useful a few years later[16].

John Cooke from Loughborough University of Technology and chairman of BCS FACS had interests in the mathematics of computing much in accord with my own, having published a book "Computer Mathematics" the previous year and other books subsequently[17]. He suggested that he and I got together to give a double act at a FACS meeting on mathematical concepts in computing. John had developed a curriculum for the maths needed for their degree course in computing at Loughborough. He had given a lot of thought to the need and rationale for the mathematics that needed to be taught. A computer program takes information in and gives information out. A program is therefore a function between input information and output. His course started from that premise, and began by showing how to specify and define functions. Programs can represent partial functions, where not all inputs lead to a specified output. Although there are partial mathematical functions, such as $\sin^{-1}$ and $\sqrt{\ }$, they are rarer and mathematical tradition does not dwell much on partiality. The partiality of programs as functions leads one on to domain theory, where a flat domain is a set such as the set of real numbers, together with an undefined element, written $\perp$. Dividing by zero or taking $\sin^{-1}(2)$ for example, produces $\perp$. An assignment statement like $x = x + 2$ in a programming language is a function from the state of the program to another state. From there one is led on to sets, logic, relations, lambda calculus, which is a notation for defining functions, and fixed-point theory, which gives a way of defining recursive functions and data-types.

John Cooke's work and my discrete maths course and book-in-progress were very much in sympathy with each other. He was exploring what mathematics was needed for a degree in computing, and I was exploring what maths was needed for software engineering. I was determined that one should take a scientific approach to the engineering of software. One needed to take notice of the mathematics underlying the science of computing in order to

[15] See Jackson et al, 1985.
[16] See Ehrig and Mahr, 1985.
[17] See Cooke and Bez, 1984 and Cooke 1988.

accurately engineer computer programs. Several of us had been greatly struck by the work of certain philosophers of science, notably Karl Popper[18] and Imre Lakatos[19]. Part of their thesis is that a scientific theory can never be proved correct. However, the theory must, to be scientific, be amenable to experiments that can have alternative outcomes, some of which would refute the theory. This can, incidentally, quite cleanly distinguish scientific and theological theories. This emphasis on refutation seemed to provide a fruitful approach to the software development life-cycle. Testing a program could be regarded as a search for a refutation of the claim that the program fulfilled its specification. A specification of a program has a dual rôle. It is a prescription for an acceptable implementation, and it is a theory of the problem to be solved. Mathematics is "the language of science" and discrete mathematics is a good way of expressing many of the problems we try to solve by constructing computer programs. Discrete mathematics is also a good tool for formulating the theory of the phenomena which are at the root of the application area of many programs, the "phenomenological theory". A formal specification thus is an interface between the application domain and the implementation, the program.

John and I gave our talks to a reasonably well attended FACS meeting held at Manchester Polytechnic, now Manchester Metropolitan University. A month later in July 1985 FACS held another meeting on automated theorem provers. The first theorem prover was probably produced by Robert S. Boyer and J. Strother Moore[20], who had been working on the problem of automated theorem proving since the 1970s. They explained the working of their theorem prover at the FACS meeting, going into details of the strategy used. There were several processes, generalisation, destructor elimination, elimination of irrelevancies, induction and heuristic use of equalities among others, which all interacted with a pool of formulae. This was a very exhilarating meeting. A final FACS event in 1985 was their annual one, the Christmas meeting. There was some emphasis on algebraic specification techniques: OBJ was a programming language that had many of the attributes of the later object oriented languages, with abstract types, generic modules and types with multiple inheritance. Another topic was modular specifications, something many of us were to skate around for some time to come.

The computer science department at the University of Surrey in Guildford held a four day tutorial and workshop on category theory. The list of speakers read like a collection of the most talented computer scientists from across the world. The first intensive day dwelt mostly on the mathematical nature of categories. We were introduced to both the basic and the more elaborate categorial concepts: objects and arrows, universal properties, duality, products and coproducts, functors, natural transformations, forgetful functors and more. In the workshop over the subsequent three days the speakers moved on to less well known and developed

[18] See Popper 1972.
[19] See Lakatos 1976.
[20] See Boyer and Moore 1979.

ideas. We saw how categories could be applied to algebraic semantics, Dijkstra's weakest preconditions which he used to give meaning to statements in imperative programming languages, typed lambda calculus, and the logic of formal specifications. Most of the ideas current in the formal specification area were looked at from a categorial perspective. This experience set me back to thinking about what theoretical computer science was really about. Perhaps it is a search for an answer to the question, "What are we talking about when we program a computer?", just as the philosophy of science is the search for an answer to the question, "What are we talking about when we do science?" This leads on to the questions, "What are we doing?" and "What ought we to be doing?" when we do programming.

The Institute of Mathematics and its Applications was found in 1964 and aims to advance the knowledge and culture of Mathematics in the UK and elsewhere. Some of my colleagues, notably Dan Simpson, had dual membership of FACS and the IMA and so were able foster links between the two organisations. So it came about that the IMA invited me to give them an informal talk on the rôles of mathematics in software engineering. This I did with pleasure, as usual eager to propagate the message that proper scientific development required a mathematical approach. I was impressed and humbled by the mathematical expertise of many of the IMA members. Later, in 1988, the IMA held a one-day conference on Mathematical Structures for Software Engineering, in which I gave a revised version of my talk[21].

The notion of an APSE, Ada Programming Support Environment, aroused a lot of interest. Its context became wider. A support environment that was independent of the programming language used would be much more useful. So was born the idea of an IPSE, Integrated Programming Support Environment. At the same time, there was some uncertainty about whether the "P" stood for Programming or Project, and trended towards Project. Various levels of IPSE were defined, 1, 2, 3, and the Alvey Directorate approved a project to develop an IPSE 2.5 with attributes partway between 2 and 3. The environment would contain tools for supporting not just program and specification development, testing, verification and so forth but also configuration management, version and compatibility control, resource planning and expenditure; hence a Project rather than a Programming Support Environment.

STL took some part in the IPSE 2.5 project from its conception, but always with some corporate uncertainty. We were not sure about whether it was in the company's interest to be involved. IPSE 2.5 would support different rôles in the development process including management, requirements analysis and the tasks involved in rigorous development. All this was quite closely allied to the work on NIMBUS and RAISE; indeed, the concern within STL

[21] See Denvir 1991.

was that if we got involved in IPSE 2.5, effort would be diverted from those other projects. At first IPSE 2.5 was to be a support environment for general software development, not specifically for formal development. So involving our department did not at first seem especially relevant. But as the IPSE 2.5 concept evolved towards greater rigour, STL became more inclined to participate. ICL and Manchester University were progressing with the project. Both Manchester's MULE and STL's NIMBUS project could benefit from mutual collaboration on IPSE 2.5. NIMBUS could quite easily add to the IPSE 2.5 proposal for packets of work which they already needed to do and which would fit into the overall scheme of an integrated support for software development. IPSE 2.5 was therefore a possible funding opportunity for NIMBUS. Much the same applied to Manchester's MULE. So there were arguments that IPSE 2.5 would both benefit and detract from the work on NIMBUS.

Other groups in STL were interested in IPSE 2.5, however. Robert Milne was working with others on hardware development methods. Hardware support systems could be another "instance" of a development hosted on an IPSE. We had a meeting with Alvey and discussed the progress of the proposal. Alvey wanted more detail about the database that is central to an IPSE, the database that holds the versions of all the products and documents produced during the course of a development: how tightly would it depend on specific equipment and platforms, and would knowledge-based techniques be used, for example. They were particularly keen on the management support features. Alvey would pay for the user trials, but not the basic development; however, they would be generous in the interpretation of that distinction. They were concerned that the end result would not be locked into VME, ICL's proprietary operating system.

We were in January 1985. The tentative timetable for IPSE 2.5 was to produce the proposal revision by 11th February, obtain company approvals by 18th February, funding approval from Alvey by 18th March and start the project on 1st April.

Will Harwood left STL at the beginning of March. He was an inspiration to the NIMBUS group and I felt that there was a danger the others might also depart; we should exert ourselves to keep them. People with experience in formal methods were in short supply and STL should maintain its lead in that arena; we would also need such skills for IPSE 2.5. For reasons of budget, morale and investment in skills, we should strive to make a working environment that encouraged people with formal methods experience to stay with us. I made these points urgently to our technical director. It was to no avail. Most of Will's team left and joined him in setting up a largely independent team to continue the NIMBUS work under the umbrella of Imperial Software Technology. IST was a "campus company" sprung out of the computer science department of Imperial College, London.

Over the previous three years I had expanded the lecture notes for my course on discrete mathematics into a draft text for a book. I had offered it to Addison-Wesley, and they sought the advice of a reviewer, an academic from the Netherlands. He was at best lukewarm at the outset, apparently not being in sympathy with my aims of presenting aspects of set theory and logic to an audience of software engineers. After I had sent the whole text to A-W, he became vociferous in his objections. My main thesis was that set theory can be used as a language for expressing the abstractions of typical software engineering problems. I aimed to communicate this to typical software engineers, for whom discrete mathematics is generally unfamiliar. In my experience, the biggest hurdle was that of associating the entities of the problem domain with "sets". I tried to establish a cognitive connection in the mind of the reader between abstract set-theoretic concepts and the phenomena of software engineering. Indeed, many competent mathematicians introduced to software engineering are slow to perceive this connection. I attempted to do this by gradual introduction of the concepts and copious examples at each stage, almost to the point of tedium. The reviewer, an experienced mathematician, wanted to strike out passages of discursive text, to reduce the number of examples dramatically, and to separate out the exercises from the rest of the development. It seemed clear to me that he found my approach unnecessary and possibly irritating. I could sympathise, but did not agree. I felt that it would be an uphill task for me to continue with A-W.

I spoke to Cliff Jones about my difficulties with A-W. He arranged a meeting for me with his department head at Manchester, Professor F. H. Sumner. They were both enthusiastic for me to cut my ties with A-W and offer my book to Macmillan. Professor Sumner was the consulting editor for their Computer Science Series. Much encouraged, I agreed to go along that path. In fact, despite our differences, there was much in the A-W reviewer's criticisms with which, on reflection, I agreed. In particular I needed to improve my presentations of proofs. At Cliff Jones' suggestion, I enlisted the help of Paul Taylor, my colleague at STL.

Paul was immensely helpful, the Macmillan editor, Malcolm Stuart was friendly and courteous, and after another few revisions I eventually delivered the final manuscript in mid-1985. Macmillan were still using old-fashioned typesetting, a method coined by publishers then as "hot metal". So a copy-editor took my script and copied it all out, querying minor issues along the way and laying out the whole text, including especially the mathematical sequences, of which there were many. The result was a galley proof, which I had to check over, and finally page proofs, which again I had to check. The opportunities for errors in all this lengthy process were great, especially since I was becoming mesmerised by the script after reading it all in intense, character by character detail for about the fourth time. The book[22] came out in print early in 1986; some of those errors embarrassingly persisted and required a small errata sheet. Today's approach in which the author prepares a text close to

[22] See Denvir 1986.

the final layout in electronic form and the publisher merely massages it into the final copy may be a little more labour intensive for the author, but is far less error-prone.

As part of the impetus to make academic courses and research industrially relevant, universities were setting up liaisons with firms to add some kind of industrial credentials to their curricula. Loughborough University of Technology invited me to contribute to a steering committee whose brief was to set the direction of their undergraduate courses. Their computing science courses included an interlude in industry. At the first meeting of the committee the department head presented statistics of those attending a conversion course: applicants (250), failures (1), resits (2), and outlines of industrial projects. Loughborough preferred the flavour of these industrial interludes to be non-critical. Pilot studies, extra pieces of work the firm would like to do but which were not essential, and tasks which were not part of a long-term project or on a critical path were ideal. These industrial interludes gave the students considerable motivation, increased their perception of the relevance of the course and engaged them hands on in a "real" project. Loughborough wanted to supplement these arrangements with external lecturers from industry and visits to IT companies who could display activities in computing, communications and human factors. There was also a continuous need for collaborative projects in which MSc students could engage.

At this time we had given VDM courses mostly to other STC or ITT organisations. The company were too wary of not losing commercial advantage to let us disseminate these good practices to the world at large, much as we would have liked to have done so. However, sometimes a customer of an STC company was invited on one of our courses. Laurie Robbins worked for the Central Computer and Telecommunications Agency, who were buying some consultancy from IDEC. As the audience were settling in their seats, he greeted me by name. "Good morning", I replied, puzzled, for he looked familiar, but I could not recall from where. After I had started the introduction to the course it came to me. I broke off and looked straight at him. "I remember", I said. "We were on a mountaineering course together!" We had both recently been on a course on mountain craft run by the Enfield Council. The change of context had thrown me. I did not even know he worked in computers. "That's right!" he beamed, pleased that I had recognised him. Some time after the course CCTA approached IDEC for advice on using VDM-style rigour in their program development. The CCTA had a large number of people working as computer programmers. They had for some time traditionally used SSADM, a graphical methodology supporting structured programming. Indeed, since 1983 they had mandated using SSADM for all new information system developments. They did not want to throw away the investment they had made in SSADM

and staff training, but were persuaded about the advantages of VDM. Could the rigorous techniques of VDM be grafted on to SSADM?

Laurie Robbins had attended my course on VDM and well understood its principles. His thought was that SSADM, Structured Systems Analysis and Design Method, was effective at the early stages of the development life cycle, but the ideas in VDM could provide a semantic model with abstract syntax for SSADM. This could increase the deep level understanding of SSADM by "experts" and extend the checks on validity that could be made through a support environment. He hoped in addition to identify possible future scenarios for using more advanced and formal techniques. He had a time window of twelve months.

We put together a project outline and proposed a finger in the air estimate: 150 person days, £75k. The project began to roll. At this stage STL and IDEC were working together increasingly in certain areas, in particular in software, and in a short time STL was to incorporate substantial parts of IDEC, becoming a multi-site research laboratory. Patrick Goldsack joined IDEC and came on the project team. He had excellent background in formal methods and was to be the principal researcher on the project, which, as often occurred, accumulated a heavy steering team. He started by going on a three-day SSADM course. The course lecturers were accustomed to an audience of data-processing people. I think they were rather stunned by Patrick's searching questions: he was trying to track down what should be a formal semantic model of SSADM. Two of the central concepts in SSADM were Data Flow Diagrams and Logical Data Structures. Both of these were well amenable to formal semantic modelling.

The formal semantic definition of DFDs made possible support tools with dialogue design. This could give an SSADM-style front end to VDM. By September 1985 the feasibility study was nearly complete we started to put together the final report.

We had last heard from the European Space Agency seven years earlier. They had shown an inflexible approach to their systems development that originated from an entirely necessary desire for high product and process quality. This time it was clear that their attitudes had changed. They were using high level languages, no longer fearful of the unknown quality and accuracy of commercial compilers, and were contemplating using formal methods and Ada.

ESA had a language for testing software called ETOL. It was interpreted by a suite of Coral66 programs. They were considering reimplementing this suite in Ada. They proposed to redesign the whole system using formal methods with an implementation in Ada in mind. Then they would recast one module in Ada and compare this implementation with its predecessor. They also wanted to consider the possibilities and feasibility of having mixed implementation languages. Using the results of these exercises they would determine whether

to go ahead and reimplement the whole suite. Prior to comparing the Ada module with its predecessor, they needed to determine a set of criteria for making the comparison.

It was heartening to find that the approaches of the ESA had advanced so far from their previous rigid state. However, I don't think the discussions resulted in a project for STL.

Various contracting firms, GEC, Ferranti, Marconi Data Systems, CEGB and STC had got together and formed an Ada UK special interest group with a subgroup focussing on formal methods. At first this SIG simply had self-instruction meetings where members demonstrated aspects of FMs to each other. A popular exercise problem was that of a set of lifts in a building. Up and down buttons were positioned on each floor, together with lamps, and call buttons for each floor in each lift, and the floors had doors to the lifts which could be open or shut. This was an excellent, familiar problem in interaction. The lifts, buttons, lamps and doors could be controlled by binary logic circuits or by software. At the first meeting we considered three techniques applied to the "lift problem": OBJ, Mascot 3, and JSD.

Mascot 3 was scarcely a formal method. It defined interfaces between functions and the configuration of software, which are equivalent to the arities and invocations of functions, without defining the effects of functions in any way. Data types too were glossed over. JSD approached requirements analysis by modelling, which resulted in formal models used early in the life cycle. The Ada UK formal methods group continued to meet periodically.

The US President, Ronald Reagan, was much exercised by the perceived threat of nuclear attack by the Soviet Union during the Cold War. In March 1983 he proposed the "Strategic Defence Initiative". This consisted of a system of laser weapons based both on the ground and in space, which would destroy any incoming nuclear ballistic missiles. SDI would form a protective virtual umbrella over the United States. The detection and laser defence systems would be connected by computer networks in order to work in close coordination. Crucially, to achieve a rapid response to immediate situations, the system would be autonomous; decisions and actions would be taken using Artificial Intelligence techniques without human intervention. The media dubbed SDI "Star Wars" after the 1977 film directed by George Lucas, itself highly innovative in that it made pioneering use of computer generated graphics. A massive amount of R&D would be needed to make these intercommunicating systems feasible. The White House originally requested $5.3B, reduced by Congress to $3.8B.

This proposal caused much debate in the computer science community. Although computers and software had been used for military purposes before, this would be an unprecedented concentration of effort on a military objective. There was also a serious question of the feasibility of the whole enterprise; the complexity and area of deployment of the system was

far in excess of anything that had been done before. David Parnas, a renowned computer scientist[23], had publicly expressed reservations about the whole enterprise, and resigned from the SDI Panel on Computing in Support of Battle Management, a government advisory committee, when his criticisms were unheeded. Yet SDI also promised an unprecedented volume of funding for addressing many problems in computing, especially in the area of concurrent computation. Some academics and industrial researchers took the attitude: so it won't work, but meanwhile we'll get lots of funding for doing research into very interesting and useful problems.

The BCS had established rules of professional conduct and ethics for its members. The Star Wars question was raised in a meeting of the Software Engineering Task Force, a Special Interest Group of the BCS. Was it ethically acceptable to take SDI funding when there was a widely held professional opinion that the desired system would never work with the available technology of the time?

There was a more general anxiety amongst those working in British software engineering that the military was taking over computing research. Much of the work done under the Alvey initiative was of interest to the MoD. There was a natural relationship between Alvey and the MoD, since the latter were a major software developer. A story in the popular trade newspaper *Computing* warned of an impending MoD hijacking of Alvey. Ada, the US DoD sponsored language, might be "just another language", but it had special technical properties and was a field ripe for R&D. The Alvey Directorate themselves asserted that they had avoided excessive intrusion of Ada and the MoD on their policies. There was, they said, no diversion from the public stance expressed at the outset: specific concentration on Ada and defence applications would be handled outside the programme.

At a meeting of the BCS SE Task Force we decided to have a special meeting to explore the SDI question, and we would invite special extra guests who might have an input on ethical matters. We had in mind Bernard Williams, a British philosopher who had written extensively on ethics. Meanwhile we had considerable debate and came to some preliminary conclusions. Infeasibility was nothing new: the Great Wall of China, TNT and many other inventions and constructions were originally believed infeasible. David Parnas need not have resigned, one argued. He was an advisor, his advice was genuine, he offered it, so it was not his fault if it was not taken. The BCS took the position that they did not advise on moral issues nor even on technical ones which were not generally agreed across the technical community. But, given that they had a policy on professionalism, they could advise on where it could be applied. Whether the BCS could proclaim on national issues was less certain at the time (since then they have done so periodically). We formulated the question for discussion: what considerations arise from principles of professionalism in computing which should determine whether one should work on projects whose feasibility one may question? Other

[23] See e.g. Parnas 2001.

strategic notions besides feasibility may affect these considerations such as the social outcome of a project.

The meeting with Bernard Williams took place on 13th December, 1985, two and a half weeks before I left STL. Also present were Richard Ennals, who was coordinator of Logic Programming and of the Flagship parallel computing project in the Alvey Directorate and who wrote extensively on artificial intelligence, Professor M. M. Lehman from Imperial College, London, acknowledged expert on the software development process, and Dr. Henry Thompson of Edinburgh University, who co-founded American Computing Professionals for Social Responsibility and British Computing and Social Responsibility.

SDI was "an automatic knowledge-based weapons system operating in real time without human control"[24]. This was an alarming prospect. Alarming because:

1. Current techniques were inadequate to build such a system.
2. Software engineering did not have the reliability necessary for a system of such criticality.
3. The current nature of research in Artificial Intelligence would not assist in building a reliable system of this kind.

These were among the arguments put forward by Professor David Parnas when he resigned from the SDI Office Panel on Computing in Support of Battle Management. He also considered that the SDI Office was an inappropriate vehicle for funding research[25] [Ennals 1986, p.52; Parnas 1985]. Ennals graphically likened SDI to a game of celestial snooker. It might be possible to program a robot to play snooker, but it would be a long term research project. SDI proposed to pot numerous incoming nuclear ballistic missiles using nuclear powered cues of X-ray lasers, without fail and without practice, while the balls were moving at speed in three dimensions.

At the BCS meeting on December 13th, 1986, Bernard Williams was at first bemused at being asked for an "ethical opinion". But he thought that perhaps philosophers should come out and declare their ethical views on current issues more than they had done to date. He said he would confine himself to ethical considerations of the question of whether people should work on SDI. He said he could not give ethical prescriptions, but could help to sort out ethical issues. He summarised the arguments he had heard in the meeting so far.

The system could "fail". This could mean:

a) it does not get off the ground;

b) the system gets constructed but doesn't work. This would be extremely dangerous especially if, for speed, human intervention was cut out;

[24] See Ennals 1986, page 90.
[25] See Ennals 1986, page 52 and Parnas 1985.

c) the system works but not as a complete strategic defence system.

He considered the position of those who were opposed to defence related research in general. One might think that they would therefore be opposed to SDI. On the other hand, they might argue: SDI won't work, so it is all right to work on it! That would not hold water, because in that case SDI would exacerbate the danger. Those opposed to nuclear research might feel able to support SDI because at base, SDI did not involve nuclear weapons.

SDI would exacerbate tensions, encouraging pre-emptive strikes, and therefore could lead to catastrophic results whether it failed or succeeded. "Working from within", a policy supported by some, is a suspect argument, possibly leading to sabotage!

Some people were concerned that it would be dishonest to work on something one knows will fail. Bernard Williams thought this was not so, provided one declared one's opinion throughout.

If one thought that SDI would lead to catastrophe, then that would be an individual reason not to work on it. If one thought that it is unwise, for example unwise for British computing, British science, the British economy etc., then that would be a professional judgement. Hence it would be within the ambit of a professional body like the BCS.

At the end of the discussion Bernard Williams remarked that he had the impression of a society (the BCS) that had just lost its innocence. This loss of innocence is reflected in an article published a year earlier by Dr. Henry Thompson, also at the 13th December BCS meeting, in New Scientist:

> As computer scientists, we know what computer systems are like, what computers can and cannot do. Not one of us, or I am confident any other responsible computer scientist, could ever literally or figuratively turn the switch which placed the means for starting a nuclear war under fully automatic unsupervised control. From that it follows that it would be profoundly and morally dishonest to connive at the creation of any programme with that as its stated goal.[26]

Meanwhile there was increasing national concern over the fact that the UK government had, along with Western Germany, agreed the secret US Memorandum of Understanding, without debate or more general consensus. This Memorandum of Understanding was classified as secret in perpetuity under the Official Secrets Act, but was leaked and printed in Aviation Weekly in January 1986. Causing even greater concern was the ever widening proscription on research results, classified as of military significance under the US COCOM rules. It seemed that Britain could either cooperate in SDI or continue with the Alvey Programme and other initiatives, but not both.

[26] See Thompson 1985.

Of course, SDI and the fierce controversies that it stimulated are now history and mostly forgotten. The Memoranda of Understanding have all been filed in official waste paper baskets.

Dana Scott developed his theory of domains from the late 1960's into the 1970s. Scott's domains enabled the mathematical modelling of computable functions and the full range of recursive data types that computable functions permit[27]. The logic with which one can reason about computable functions and data types following Scott's theory became known as LCF, the Logic of Computable Functions[28]. At the same time, through the seventies and into the eighties and beyond, the ability to prove desirable properties of programs became something of a great quest. At the time of writing, one of the Grand Challenges initiated by the UK Computing Research Committee in 2004 focuses on Dependable Systems Evolution. The goal of GC6 is to produce a Verifying Compiler and a repository of verified software. Proving properties of computable functions remains a quest and challenge. This UK Grand Challenge GC6 is related to the EU ISTAG Grand Challenges in the Evolution of the Information Society, challenge number 4[29].

The meanings or semantics of programs are elements of Scott domains and Scott's LCF is a notation for writing those meanings. It was a form of λ-calculus, typed with two base types, natural numbers and Boolean values. Robin Milner further developed Scott's LCF with a more elaborate type system, allowing recursively defined types. Theorem provers for theorems written in Milner's LCF[30] were then developed from the late seventies into the eighties. To develop a theorem prover, one needs a language in which to program it. Robin Milner designed ML[31], an abbreviation for Meta-Language, expressly to facilitate theorem proving in LCF. ML is a functional programming language, but with a powerful polymorphic type inference system. ML has a type checking mechanism based on a Unification technique. Unification was used in Prolog and, although as an algorithm it is quite simple, it is effective in deducing logical terms from their predecessors in a proof or argument.

The essentially beautiful idea in ML is that a type can be defined as a set of logical terms, with the axioms and the type inference rules as the rules of deduction allowed by the logic. By defining LCF as a data type in ML in this way, proofs as it were come for free along with the type inference system. The polymorphism in ML, an ability to allow types themselves to stand as data in a function definition, enable proof strategies, which Milner called "Tactics", to be expressed as functions. Operations that combine tactics could be expressed as higher order functions delivering more elaborate strategies, called "Tacticals".

[27] See Scott 1982.
[28] See Scott 1971.
[29] See ISTAG 2004.
[30] See Milner 1979.
[31] See Milner et al, 1990.

LCF and ML are together an example of a remarkable collaboration amongst academic institutions. Milner's original LCF was developed while he was at Stanford University in Connecticut. Further developments resulting in increasingly useful and powerful versions of LCF took place in Edinburgh, Cambridge and elsewhere up to 1987, resulting in Edinburgh and Cambridge LCF. At the same time ML was developed further and "Standard ML" was later defined and published in 1990, revised in 1997[32]. Implementations of SML have been produced by many institutions, some of them extended in radical ways and today, almost all of them available as open source. LCF has in turn been extended into other useful and effective proof systems and languages, notably HOL (Higher Order Logic), ProofPower (by ICL), and proof assistants for NuPRL and Martin Löf type theory. Probably unknown to most practising software engineers, the work on LCF and ML, carried out over some 25 years, has underpinned the foundations of future reliable software engineering.

Several countries were beginning to recognise the impact that software would have over their national economies. Nationally funded software research programmes began to flourish. In Britain an advisory committee chaired by John Alvey made a list of recommendations to the Department of Trade and Industry. The government accepted most of these and the Alvey programme began in 1973. This was a wide programme involving three government ministries, the DTI, the Ministry of Defence and the Science and Engineering Research Council in the Department of Education and Science. Industrial and academic research groups were to take part in the programme. The Alvey programme covered four enabling technologies:

- Very Large Scale Integration (VLSI);
- Software Engineering;
- Man-Machine Interfaces (MMI);
- Intelligent Knowledge Based Systems (IKBS), more generally called Artificial Intelligence (AI).

One of the aims of the Alvey programme was to encourage more coordination between these different areas. Workers in the four areas had not communicated enough with each other, it seemed. Another aim was to foster industrial and academic collaboration. Industry would give commercially useful focus to academic research and academia would bring the best of theoretical understanding and knowledge to future industrial products. That was the theory. In collaborative projects, the industrial partners were generally funded up to 50% and the academic partners 100%. Other, longer term research projects could have academic-only partners.

[32] See Milner et al, 1990 and 1997.

Around the same time the European Union approved the ESPRIT programme, the European Strategic Programme for Research into Information Technology. The motivation for this initiative was very much an economic one. The research activities of European companies were considered much smaller than their overseas rivals. Cooperation was seen as the way to international competitiveness for the EU. The European Commission ran ESPRIT, but each phase of the programme would be approved by the EU Council of Ministers. Each phase, called a Framework, would be initiated by a call for proposals. Successful proposals generally received 50% funding. Each Framework, of which there have been some seven now, would have a number of technical areas. The initial areas of the ESPRIT programme were:

- Advanced Micro-electronics;
- Software Technology;
- Advanced Information Processing;
- Office Systems;
- Computer-integrated Manufacture.

Each Framework would embody a five-year programme, but the Frameworks overlapped in time substantially. Work under at least two Frameworks could be progressing at the same time.

Slightly earlier, in 1981, Japan announced its intention to start a brave new generation of computers, called the “Fifth Generation”. One could fairly argue that the need to keep competitive with Japan stimulated the initiation of both ESPRIT and the Alvey programme. The Japanese initiative placed great emphasis on AI. Their plan was to implement “mechanisms for inference, association and learning” in hardware. Then artificial intelligence software would be developed to make full use of these hardware functions. These proposals were put forward by the Japanese Institute for New Generation Computer Technology (ICOT). This was a radical strategy, with heavy focus on AI and PROLOG, the logic programming language. The programme would require highly parallel computer architectures and very large scale integrated circuits in order to enable the massive computer processing which implementing such far-reaching AI would require. (The human brain is slow and lumbering compared to a computer, which is why we cannot calculate as fast as one. But a brain has a massively parallel “architecture”, which is why even the relatively minute brain of a bird can recognise visual patterns far more quickly than any computer). The long term aims of the Japanese programme have today not yet been realised, but their research programme has put Japan into the forefront of AI research. Several popular products have emerged from this effort, such as the Sony “dog” that can learn and be trained. The ICOT programme set some challenging targets for the rest of the computing world.

Other national programmes of research began following ICOT's publication of their initiative. The most notable was from the USA, where funding came from the military. The Defence Advanced Research Projects Agency (DARPA) launched a Strategic Computing Programme with a substantial budget over five years.

Meanwhile all practising computer scientists and software engineers have to work for some institution or other. I was still with STL. Software had become more significant in the company. At the top of the organisation's hierarchy, just below the managing director, there were three technical directors. One of these now had specific responsibility for IT and information systems. With this newly enlarged significance of IT and software, we were frequently reviewing our strategy. What were we here for, in corporate terms? Because there was no long-standing company tradition in software, upper management were mostly experienced in other engineering specialities. This meant that we, at somewhat lower levels of the hierarchy, had more control over our own technical destiny than we otherwise would have done. Software technology was rapidly becoming ubiquitous, like transistors and semiconductors. To plan for our own future we needed to be technologically aware. Attending conferences, keeping abreast of scientific literature, taking part in collaborative research, and interacting with our peers in other institutions all worked to improve our awareness. We had a rôle in assisting other companies within STC who were developing software-intensive products. We helped them to exploit new methods and development tools in the short term, up to three years ahead. In other words, we had a technology transfer rôle. We also had a rôle in longer term R&D, more than six years ahead, say, in order to ensure the company's competitiveness for the more distant future. We therefore had to decide proportions of investment and emphasis for each. To date our acquisition of projects had been opportunistic rather than according to a policy. We involved the technical people who were "on the job" in these policy discussions, because we ourselves had technical backgrounds. We devised a meta-plan for selecting and steering projects. Policy provides a direction, a programme which evolves but has a theme, formal methods for example, and an objective like "efficient software systems". Criteria can evolve and change; for example cost and correctness may be traded off. Sometimes one might focus on a product or service, but these are merely milestones in a more general route, even if they take many person-years to build. We had already started to engage in collaborative projects, something we had not done all that much before: We were negotiating actual and prospective collaborations with ICL: FORMAP, IPSE2.5, RAISE and LOTOS. Given the knowledge and experience that our software research group had acquired over the previous few years, we could, if we wanted, develop and sell engineering products. Should we do so? This had not previously been part of STC's business. We gave several talks to own management in order to "sell" our ideas on software strategy.

All this local soul-searching and policy formation was part of a larger company-wide reshaping. A new company within the group, STC Technology, was formed. In October 1985 STC Technology divided into two, STC Technology Labs and STC Technology Enterprises. The Software Engineering Technology Centre was part of the SW directorate within STL. The latter was formed from units previously within STL and IDEC. So STL itself became multi-site. We were used to the Harlow site being part of our identity, so this change affected individuals quite strongly.

STC had taken over ICL and then found itself financially strapped. To save money they sold off their prestigious headquarters building, 190 The Strand in central London. This was an ominous move. What had been ICL was sold on to Fujitsu. Large numbers of people were being given early retirement or redundancy notices. The NIMBUS team had already departed earlier in the year. One of my closest colleagues, Bernie Cohen, had been offered a Chair at the University of Surrey in Guildford, and had accepted with the STL management's blessing. The software research group was seriously disintegrating. I and two other close colleagues, Mel Jackson and Roger Shaw, looked for pastures new and we all left at the end of 1985. I made a list of my responsibilities in order to hand them over tidily. I was involved in five main projects and about eight minor ones. I was a member of the Technical Board and joint Project Architect with Dines Bjørner in the RAISE project. In the FORMAP project I was the local site coordinator attending project management committee meetings and communicating their decisions down the line. Springer Verlag were publishing the proceedings of the workshop we had held on the Analysis of Concurrent Systems two years earlier; I was the liaison with Springer and the participants, who should receive copies when they were produced. The VDM Toolset project was about to start but had not yet imposed any great commitment. A consultancy contract with the CCTA, for which I was the coordinator, was to all purposes complete. Other consulting rôles and membership of committees I could hand over to other individuals or simply continue as a personal commitment. I felt obliged to ensure that my leaving STL did not harm any of these endeavours.

Chapter 10 Theory in Practice

Roger Shaw, Mel Jackson and I had all felt that we would like to push the exploitation of formal methods further and more energetically than ITT or STC had been willing to do. Our employers were not in the business of software development as an end in itself; they were interested in software only as a component of telecommunications products. We thought long and hard about setting up a company of our own, but hesitated, because without any contracts

in sight, this would have been a massive risk. We all had families and mortgages to support. Then one day I noticed that Praxis, a software house in the city of Bath, was beginning to make a name for itself, branding itself as a company dedicated to high quality software and keen to use formal methods. Bath was in the west of England, 130 miles from London. Praxis was formed from the South-West Universities Computer Centre, when that organisation was disbanded by the south-west universities. The time had come for nearly all universities to have computer centres of their own. SWURCC was helped by several organisations to transform itself into a commercial software house, making its way by writing bespoke software for customers. SWURCC had taken part with us in the Augusta project. I felt that the manager of SWURCC, Martyn Thomas, and I were on the same wavelength. He had said complimentary things to me after my presentation on the Augusta project at its completion conference at the National Physical Laboratory, so I thought he had some belief in my abilities. Martyn was now chairman of Praxis. I suggested to Mel and Roger that we approach Praxis and proposed that the time was ripe for them to set up a London office, and we three would spearhead it. I composed a letter to this effect and sent it to Martyn Thomas. He telephoned me a few days later and said that my letter was waiting for him when he returned from holiday.

Over the next few weeks Roger, Mel and I had several meetings with Praxis directors. We worked over a business plan for forming a London office in some painstaking detail. For the first time in fifteen years I came face to face again with the financial rigours of operating a small company that depended entirely on sales for its income, unsupported by grants from a larger organisation or group. In the end we were all forced reluctantly to the same conclusion: a London office was not a viable proposition for Praxis at this time. Then the Praxis directors suggested that the three of us join Praxis in Bath as senior but otherwise regular employees. By that time the three of us had got to know and like the company and found this prospect tempting. David Bean, the managing director, had a one to one chat with each of us and we each received job offers that day. Mel and I both accepted, but Roger decided to stay at STL for the time being. He indicated that he might join us later. So on January 1st, 1986, I changed jobs and employers. I had already booked a holiday in the Lake District with my family, and slightly to my surprise, Praxis were willing for me to start my new employment with a week's leave. Furthermore, since I was eligible for a company car with Praxis, as I had been with STL, Praxis permitted me to take delivery of my car in advance to use on holiday. I was impressed!

I still lived in London, my wife Hazel was doing research into mathematics education at Chelsea College, London University, and both my children were established at an excellent school in North London on course to take their GCSE and A-level public exams. I did not plan to move to Bath, so Praxis and I came to a deal: I would commute in the reverse direction to normal from London to Bath, staying overnight if necessary, and take on any

London based business and contracts that I could. Sure enough, within days I became involved in a contract with Oracle, the database company. I had never worked in databases before, knew only the bare principles of what they were about and protested to that effect. My protestations were waved aside. We all have to be technically flexible, they said. You'll pick it up quickly.

Oracle's product was a relational database hosted on the UNIX operating system on IBM personal computers. IBM designed and manufactured the first personal computers. Only later were they required to publish sufficient details of the design so that other companies could sell PCs in competition. Relational databases were based on the mathematical theory of relational algebras, pioneered by E. F. Codd[1]. Oracle had sold one of their database systems to BT, who wanted to have a custom user interface developed. This user interface was to consist of a syntax driven text analyser. Oracle estimated two to four person weeks for developing this user interface, but wanted help with the design. Their own staff would then implement it, so that they would keep knowledge of whole system within their company and be able to update it further as it and its environment evolved. I was to learn that over the next few years more and more customers wanted this kind of arrangement with Praxis and, I assume, other software houses. The software house would carry out part of the early stages of the development life-cycle and the customer would then do the rest, including the implementation. This way the client keeps control of the design and is able to continue maintenance without being dependent on the supplier.

Databases were the mainstay of commercial data processing, an area I had been away from since my days at ULACS. A database typically contains large amounts of data in a structured form. The user of a database is able to query it and obtain information about how many records exist with certain properties, and other such information. The user queries the database in a purpose-designed language. The first databases were developed in the mid-1970s, but the more sophisticated Structured Query Language, SQL, was developed a little later. The first commercial versions were released by Oracle and IBM in 1979. By 1986 the American National Standards Institute adopted SQL as a standard. ISO followed suit a year later.

Oracle saw this contract with BT as an opportunity to upgrade their own product and improve their competitiveness. They were already using a specialist database consultant, John Ashford, but they required more expertise with drafting the syntax of the SQL extensions in BNF. I was surprised that any firm needed extra help to write some BNF, for it had been around for nearly thirty years and formed the foundation of any language definition. But apparently familiarity with BNF was less widespread than I thought. The implementation would be programmed in the C language. We were not to count on any particular operating system architecture. At a meeting with John Ashford, he pointed me towards several journals

[1] See Codd 1970.

specialising in database technology and related topics: the Journal of Information Science, the Journal of Documentation, Intelligent Information Retrieval, and some books and papers. In the contract negotiations we agreed that Oracle would subcontract fifteen person days to Praxis, ten of which would be allocated to me.

Praxis' emphasis on quality meant that they had written procedures for every activity, both technical and administrative. I was used to company standards from working in ITT, who had extensive quality standards and procedures, recorded in multiple written volumes. Before I could start on the work proper, a Project Authorisation had to be signed. We needed a statement of the technical requirements from BT.

The text-oriented extensions to SQL that BT and Oracle required included extended Boolean conditions, in particular where data contains a defined text expression. I imagine this would be used for on-line directory enquiries. The diagnostics needed to be consistent with those already available with the SQL facilities, so that users accustomed to the existing systems would not have to relearn anything. We had to design the architecture to enable the diagnostics to be orthogonal: an error in using part of the extended syntax should produce its own diagnostic rather than an irrelevant one from the central analyser.

I spent ten days on this project. It took me six days to understand and sort out the actual requirements, and four days to write the syntax extensions. These amounted to four pages of BNF with some introductory and explanatory text. I was bemused by anyone paying some thousands of pounds for four pages of BNF, but everyone seemed to regard it as good value. I then learned Praxis' involved procedure for closing a project. I had to write an internal Debrief Report, have it reviewed and signed off at a review meeting, write a Closure Report for the client, write a Project Charges Summary and authorise Accounts to send the invoice. The Debrief Report and its review meeting took me another two and a half hours.

Oracle requested a meeting to discuss the invoicing. It turned out that BT had decided not to proceed with the extensions after all, so Oracle had lost their contract with them. Oracle therefore proposed not to pay Praxis because the work, which we had already completed, was no longer required. Naturally, Praxis objected; we had a contract with Oracle and Oracle had to fulfil their part under the terms. Praxis received payment after sending a solicitor's letter. I felt a bit sad that the first job I had done for Praxis led to a legal conflict, but it was not of my doing.

Personal computers were the outcome of microcomputers, small computers constructed around the microprocessors that were developed in the mid-seventies. As the name suggests, personal computers were designed to be used by one person at a time, interactively. At first there was a proliferation of designs of microcomputers; in the UK the BBC micro and the Sinclair ZX80, ZX81 and Spectrum were among the most popular during the early 1980s. IBM joined the personal computer market in 1980 in response to the competition from half a

dozen other firms. The IBM PC became dominant, but competition again was rife with many firms manufacturing PC clones. However, when I joined Praxis in 1986, the dominance of the IBM PC was by no means yet clear. Other personal computers presented serious competition. Furthermore, for use in offices and scientific and engineering institutions, multi-access machines where users had a "dumb" terminal on their desk linked to a central mini-computer were still for some time the norm, rather than a lot of personal computers served by a file server enabling co-operative use and shared data. A dumb terminal would consist of a keyboard and monitor with little or no processing power of its own. More powerful personal computers designed for engineering applications, with greater processing power and graphics, were known as Workstations.

In fact Praxis had no in-house computer system when I joined them. They were soon to do so, however, and a central Vax minicomputer was installed with dumb terminals on individual desks. The contract with Oracle was to extend software that would run on a personal computer, an IBM PC. The usual operating system for an IBM PC was PC-DOS, developed by Microsoft on contract to IBM. Microsoft developed variations and upgrades which they sold themselves as MS-DOS. At the same time AT&T Bell Labs developed an operating system called UNIX, an evolution from previous systems, which was hosted on a number of workstations and minicomputers. A version was developed for PCs, and the platform for the database in the Oracle contract was PCs running UNIX.

Operating systems were normally written in an assembly language, because an operating system needs to drive the machine hardware such as peripheral devices, printers and hard discs, directly. UNIX broke new ground by being probably the first OS to be written in a high level language. The language C was designed by Bell Labs for writing UNIX. C was a block structured, imperative programming language with fairly normal high level features, but also with low level facilities for driving the hardware. Nonetheless it encouraged machine-independent programming and transportability across machines. Although being designed for systems programming, it was and is used for programming applications too. The extensions to Oracle's database system were to be implemented in C.

The contract with Oracle had some momentum, in that John Ashford, the independent consultant we worked with, was quite keen to continue the experience, as it were. He had a prospect of a contract with the Foreign and Commonwealth Office, having recently done work for the Scottish Office, another government department. I accompanied him to the FCO several times to bid for the task. They were in the process of upgrading their internal library system, which covered five sites. They were also about to move amongst these sites, so the system needed to be flexible to cater for these moves. Any purchase of hardware, software or consultancy had to conform with EEC regulations.

The work, being more intensively in databases, would not involve me personally, but being based in London it was easier for me to take part in these initial negotiations than other Praxis staff. There were about a dozen possible Praxis people who could do the actual work, in addition to John Ashford. John wrote the first draft of the proposal, and then I and others discussed it and tossed many modifications back and forth. Review meetings in which documents, including bids and proposals, were scrutinised and modified through several versions, were a prominent part of the Praxis Quality System. But these kinds of quality procedures were by no means unique to Praxis; the recent U.K. Standard, BS5750[2], defined a 'model for quality assurance in design, development, production, installation and servicing' in industry, and an increasing number of firms were attempting to be certified by BSI to BS5750 conformance. The international standards organisation, ISO, later adopted BS5750 so that it became ISO9001 and ISO9002, parts of a series, ISO 9000. But in this case no contract resulted from our bid.

When I joined STL I stipulated that I did not want to work on military projects. Since my first post was managing the 3200 BSCC, there was no problem with this because we had no connection with the military, only with civil telecommunication systems. Towards the end of my time there, the scope of work had changed considerably and I was at one time asked to discuss a possible contract with GCHQ, the Government Communication Headquarters, an intelligence and security organisation. GCHQ is a Civil Service Department, which works closely with MI5 and MI6. I objected, explaining my position on military projects. My managers were unaware of this. Many changes in organisation and in the reporting hierarchy had taken place over the years. I had a number of awkward interviews with some senior administrators. I had another reservation about GCHQ. They had recently been required by the government not to allow their staff to belong to trades unions. I did not want to support an organisation that denied its employees' rights in this way. But that was a more difficult objection to raise.

Praxis had a company policy not to work on weapons. This was a welcome contrast and the policy attracted employees with packages of particular ethical outlooks. For example, about 40% of the staff were vegetarian. The company did not shy away from contracts with defence organisations, however, so long as these did not involve work on weapons. But the policy was not written down and was open to much debate. Should we seek a balance of clients? How much should we respect employees' individual ethical views? We debated whether a vegetarian employee might refuse to work on a database for a meat distributor. More general discussions on marketing policy followed. Do we seek business from large or small companies? From customers with any particular kind of end product or from any particular sectors? For example, large scale consumer products, automotive, TV, radio, airlines,

[2] See BS 1991.

aerospace? From public service, utility or fuel companies, BT, CEGB, British Gas, BP, Shell? From suppliers to other industries, telecommunications, software houses (such as Oracle, the client in the recent contract), data processing companies, computer manufacturers? To what extent, if any, should we sell products as opposed to services? To date Praxis had sold services only, not products such as a software package of some general use that might be sold to several, or even many, customers. The whole finance of projects to develop products was radically different from that of projects to develop bespoke software. In the latter case the development costs have to be covered by that single sale, whereas with products the development costs can be recovered after a projected number of sales. How should we decide such questions? The strengths of the company, its reputation and growth, the need for security (e.g. not to have too many eggs in one basket) were some of the factors. All these questions revealed the youth of the company.

In fact, Praxis had one single product which a team maintained and marketed. Ella was a language for simulating digital electronic hardware. With Ella one could construct a computer model of a circuit consisting of nodes connected by wires. The nodes themselves are multiple functions, which could be defined to perform as typical electronic gates and so on. The signals flowing through the wires consisted of data of some type; types could be defined much as in any normal high level programming language like Pascal.

There were other languages for defining and simulating digital hardware. STL, my previous employer, had defined a language called LTS. Robert Milne, whom I knew well, was instrumental in the work, which was well grounded in process algebra theory. One of my early tasks at Praxis was to examine Ella and LTS in depth and write a report comparing the two. STL planned extension to LTS and were seeking Alvey funding to develop a simulator for the extended language. Praxis was a possible partner in the project, but the considerable preliminary work, which spanned a couple of moths, did not result in any contract.

Praxis began life three years earlier in 1983 and had been profitable from the start. It had grown progressively and had some seventy staff when I joined; I was employee number 69. Its ambitious objective was to become a foremost UK developer of high quality software. One of the selling points was the “Praxis quality culture”. They achieved certification to BS5750 in the year that I joined. Slightly to my surprise, there were no sales staff; the notion was that everyone was a salesperson. This contrasted from my two previous experiences in software houses at ULACS and RADICS, where a sales team (one person in the case of RADICS) handled all the marketing and sales promotions and bids. In Praxis producing a bid for a contract was the first stage of the project. I spent much of my time working on the composition of bids.

The Alvey Directorate awarded grants and funding to support partners from UK industry and academia to carry out collaborative research and development projects. Each project had a nominated Project Officer within the Directorate. However, the Alvey Directorate was a small organisation. The limited number of Project Officers meant that each had responsibility for many more projects than he or she could keep an eye on. So for each project, the Project Officer would engage an independent Monitoring Officer from outside the Directorate, someone either from industry or academia, to observe the progress of the project and report back. The fees for this task were individually negotiated, but fell within a broadly standard band.

One such project was "Analyst Assist". Its aim was to take the JSD design method and enhance it with the Artificial Intelligence techniques of Intelligent Knowledge-Based Systems. In the JSD method one starts by modelling the real-world entities of the problem before proceeding with the design of the computer system which is to deal with them. One difficulty that system designers had was with the initial step of extracting and representing the real world entities. IKBS use a database of existing knowledge on a subject and the automated inference techniques of AI to attempt to solve a presented problem. By applying IKBS to the initial stages of JSD, the Analyst Assist project hoped to produce an enhanced tool for the requirements analysis phase of system design. Some eight organisations were to take part, some of them representing potential users of the end result and playing a reviewing rôle in order to keep a focus on usability and commercial exploitation during the project. Data Logic, a software and systems house, were the principal contractor. Other industrial partners were Scicon and MJSL, Michael Jackson Systems Limited. The academic partner was UMIST. Two of the reviewing partners were the UK car manufacturer, Rover and United Distillers, who were about to merge with Guinness. I was asked to be the Monitoring Officer for this project.

The Alvey Directorate understandably stipulated that its Monitoring Officers maintain strict commercial confidentiality. I would be exposed to the research and development work carried out by the commercial partners. I had to sign a confidentiality agreement not to divulge any information, including my own reports to the Directorate, to any other person or organisation. But the Praxis quality standards insisted that all my reports in a project, especially those sent to a customer, should be reviewed by other staff in a review meeting, before delivery. They could make no exceptions, for this practice had to apply to all projects in order for Praxis to conform to and maintain their certification to BS5750, an achievement which they had won only after a great deal of work and investment. These conflicting requirements very nearly led to an impasse, but the Alvey Directorate finally agreed that one other Praxis member of staff, our quality manager, Chris Miller, could sign the confidentiality

agreement just for the purpose of reviewing my reports to the Directorate. In the event, Chris was entirely relaxed about these reviews, which were no kind of obstacle during the project.

The project rolled on over the next three years, the work was done, I attended many project management meetings, which were occasionally quite acrimonious but on the whole proceeded to plan, and I wrote my reports to the Directorate. As was typical with funded collaborative projects, initial contractual difficulties made for a slow start to the work. After ten months Distillers withdrew from the project because, following the merger with Guinness, their new management did not have the same motivations for taking part. With all funded projects like this, the funding provider, in this case the Alvey Directorate, require deliverables and milestones at intervals in order to monitor progress and to have concrete evidence that something useful is being done. It was the Monitoring Officer's job to check that the milestones are really reached and examine the deliverables, reporting on whether they met their stated aims. The partners had particular difficulty in agreeing on the development hardware that they should use. But a year in, all eventually agreed. One question perplexed me: some of the deliverables were pieces of software. It made no sense for the project actually to send these software items to the Directorate, who did not have the platforms on which to run them, and who would not be interested in using them anyway. The Directorate told me they simply wanted a statement from me saying that I had seen a demonstration of the software and that it was working.

During the project there were the typical many hiccups all along the way, with questions of copyright of deliverables, the evaluating partner wanting to share some deliverables with a third U.S. Party, and some partners having staff leave or be assigned to other "more urgent" projects. But the whole project did useful innovative work and provided a maturing experience for the participants, leading to a stronger group that might embark on more collaborative projects.

Right at the end of the project the lead partner made the project manager redundant. The work had been done, but the final report had not yet been written. As Monitoring Officer I contacted the lead partner and insisted that the final report must be delivered, otherwise they would not receive their final payment, and could be liable to repay some of the grant already paid to them. As I hoped, they re-engaged the project manager on contract to write the report. This he did, in September 1990, and the report was of very good quality.

The concept of the APSE, Ada Programming Support Environment, led to more general Project Support Environments. PSE became the flavour, not just of the month but of the next several years. RSRE on behalf of the UK MoD began to solicit bids for a UK funded PSE. This went through a gamut of initial reports evaluating current practice, evaluating options and requirements, risk analysis and final conclusions. With a variety of software suppliers,

insisting on conformance to a standard development environment would ease maintenance across different sources, reduce dependence on suppliers and maximise transport through advancing computer architectures, technologies, development methods and practices for the MoD client. We gave a presentation to RSRE to make a claim for our credentials for bidding. These presentations were all to have a similar form: we would play back our understanding of their requirements, with added detail of our own to demonstrate our insights into not just the requirements, but also the technical background quality considerations, relevance of a PSE to the client's business, available options fro the way forward and our own take on the best approach and wider social and strategic impacts. These bid efforts, much greater and more intense than I had experienced in the research environment of STL, consumed a considerable amount of effort. The cost of a sale had to be absorbed into the eventual contract, or if no contract ensued, into overheads. Many such bid efforts, like this one, came to nothing. This was not necessarily because our bids were not competitive; frequently the client decided not to go ahead with the work at all, for internal economic or political reasons. I think also that some clients would be doing an elaborate thought experiment in their strategic planning and use invited bids to add insight and knowledge to their own ideas.

IFIP, the International Federation for Information Processing was established in 1960 by UNESCO following the first World Computer Congress in Paris in 1959. IFIP contains over one hundred working groups covering numerous aspects of information technology and computing. National computing professional bodies are affiliated to IFIP, which often leads international research in knowledge and practice. Every two years IFIP runs a world congress, a major event in the computer science calendar. IFIP'86 was held at Trinity College, Dublin, and I was invited to give an introductory talk on the scientific aspects of software engineering. My talk was one of a pair, the other being given by Professor M. M. Lehman of Imperial College, London. I sought to draw parallels between schools of thought in the philosophy of science and those in software engineering. Formal methods and the use of proof in software development were akin to the theories of Popper[3] and Lakatos[4], which emphasised rationality, whereas metrics and the more sociological studies of the development process were related to the ideas of Carnap and Kuhn[5]. Professor Lehman's paper concentrated on an empirical account of the laws of software development, resulting from extensive observations.

The difference in viewpoint between followers of formal methods and advocates of metrics evolved into a considerable rivalry, at times approaching hostility, and reflected, I think, the differences between the two cohorts of philosophies of science. The metricists would claim to

[3] See Popper 1963.
[4] See Lakatos 1976.
[5] See Khun 1962.

have the more scientific approach because they measured the phenomena of software development, and experimentation and measurement are fundamental to scientific method. The formalists would claim that there is no scientific theory without an underlying theoretical model of what software actually was. In fact, I believe, there is no real need for conflict between the two viewpoints.

Working for Praxis, I came across models of business relationships and techniques that I had not encountered before. On a visit to London Transport in Acton, I learned that they accepted software from their suppliers, GEC and Westinghouse, on a sale or return contract! In other words, if the software did not perform to the customers satisfaction, no payment was due, it seemed. LT, the customer, validated the delivered software in parallel with the suppliers. Satisfactory performance was related to calculations of MTBF, Mean Time Between Failures. These failure calculations were related to reliability claims. Thus, metrics and reliability have relevance where the software sits in an environment whose characteristics are not fully defined or understood.

Despite being much nearer the commercial coal-face, opportunities for attending and contributing to conferences, and participating in professional committees continued. I gave papers to a conference on Electronics in Oil and Gas, continued to take part in the BCS Software Engineering Task Force (renamed as the SE Technical Committee), the VDM Standardisation committee, and attended seminars on OBJ, CSP, models of polymorphic λ calculus, the analysis of CMOS circuits. Although Praxis were keen for their programmers to learn and use VDM and other formal methods, there were always too many pressing jobs on hand for any substantial group of employees to take time away to attend a course. One day Mel Jackson and I decided to lay on a brief course at lunchtimes over a few days and we simply notified all staff using the local electronic news-board and held it. The manager responsible for day-to-day working matters was away and we wondered how he would react when he returned. In fact he was very pleased with our initiative and more courses followed, in VDM, proving programs correct, and discrete mathematics - logic and set theory. We began to think about offering these courses outside Praxis to customers as we had done to some extent at STL.

RSRE, the government MoD technical research establishment, designed and built a computer for high reliability embedded systems, called Viper. The Viper machine was specified in LCF, Logic of Computable Functions originated by Robin Milner at Edinburgh University. They had gone to considerable lengths to make it near impossible for the machine to go into the kind of error states that frequently plagued other computers, endless loops, deadlock,

arithmetic overload, division by zero, array bound violation, et cetera. RSRE also designed a language called NewSpeak, primarily for programming the Viper machine. NewSpeak had various unusual features: finite types enabling compile-time bound checking, a limited form of recursion to prevent infinite recursion at run time, and others. The language design would necessitate a rather particular programming style. RSRE were inviting bids for writing a compiler for NewSpeak, and wanted to have constructive discussions with the implementers on the language design. I had some concerns about parts of the language, because I thought there was one area where ambiguity might be possible. Back on Praxis premises we had numerous internal meetings to discuss the bid. Our proposal needed to emphasise our understanding of the client's needs, as well as our own skills and competence to do the job. The bid was scrutinised in several review meetings by the most senior staff. The Viper machine already had government funding, and the NewSpeak compiler would be financed from the same source. RSRE did not want NewSpeak to be implemented in itself, a process called bootstrapping, and for a time favoured Algol68 as an implementation language. The whole piece of work would be a cooperative effort between supplier and client.

In the end, once again no project resulted from this bid. As far as I know, NewSpeak was never implemented.

The National Computer Centre was preparing to produce a new series of guides for the IT industry on current best practice, the STARTS Guides. These were to follow a number of successful advisory publications on software development and usage. They were preparing to produce a STARTS guide on software development methods, giving a comparative account of the various methods available, much in the manner of a Consumers' Association Which? Report, with scores allocated to each method for a list of criteria. Joe Rhodes of EASAMS, an IT services and consultancy company, was to be the team leader. Joe was assembling a team from different organisations, a common practice for independent government-funded institutions like the DTI and NCC. The practice was intended to minimize commercial bias. He approached Praxis to provide a team member. The NCC had a fixed consultancy rate for this kind of activity, £300 per day, substantially lower than the normal Praxis charge-out rate for someone of my grade. But the new Managing Director, David Allen, considered that this was "strategic" work and wanted to go ahead with the task. I certainly wasn't going to complain, but I was surprised, because I had had to argue strongly in favour of other more worthy contracts which were likewise less profitable than desired. So I joined the work along with four other recruited team members, most of whom I already knew.

The first thing we had to do was to decide on the criteria against which to score the various design methods. We discussed these at length, Abstraction, Data Refinement, Information Hiding, Functional Decomposition, Module nesting, Import and Export control, Expressive

Power – that one expanded into a long list of subcategories, Traceability, Support for Designers – expanded again, Ease of tool use, Performance... We reckoned to have a first draft of a scoring scheme with all the criteria by the beginning of December 1986, three weeks later. One month later we had the scoring scheme and agreed to rate seven methods, giving reasons for each score. The methods were to be SSADM, Yourdon (a method based on the principles of structured programming), Object Oriented Design as promulgated by Grady Booch[6], Smalltalk, VDM, Z[7] and OBJ. We agreed to spend up to half a person-day for each. I set out to score the formal methods, VDM, Z and OBJ.

Like all other work done at Praxis, my contributions to the STARTS Guide work had to go through an internal review. My Praxis colleagues had some reluctance to discuss the scores I had given. Still their other comments were helpful. When the team began to compare scores for the different methods, we found the same difficulty. As the project proceeded, we found it necessary to adjust the criteria. Some facets, such as Availability (public domain, in-house only, in development etc.) were easier to compare. We allocated different weights to different facets in order to obtain an overall score and comparison, and again, we debated and adjusted these weights continually through the project.

The project rolled smoothly to its completion at the end of 1987 after just over a year. It was one of the most unproblematic projects I have ever worked on, substantially owing to the relaxed and capable leadership of Joe Rhodes.

Praxis made a great play of its quality standards. They were a selling point for the company's business. One member of the small management, team, Chris Miller, was Quality Manager. He insisted that the programming staff owned the local standards, so that there should be no sense of their being a bureaucratic burden. This did not entirely work; the staff struggled to meet the standards, but the will was certainly there. Chris asked me if I could draft some new standards, as at the time I was not fully engaged on revenue-earning projects. So I agreed to write three technical standards on Modules, Data Structure and Metrics.

For a piece of software to be easily manageable, it should be divided into suitable pieces called modules. Each module needs to be as self-contained as possible, with simple interfaces to other modules. There should be a clear relationship with the overall specification of the software. Much has been written on this topic over the years, from about 1972 onwards. If software is written according to good principles of modular structure, it is easier for the maintenance of the software to be handed on from one programmer to another, inevitable if the software will have a long lifetime. Good modular structure helps to make the software comprehensible to the next programmer to delve into it. One of the most authoritative writers

[6] See Booch 1980.
[7] See Abrial 1996.

on the subject was David Parnas who has been affiliated to many universities[8]. Modules can be hierarchically nested by which modules make use of the services that others supply.

When constructing a program, designing the data is at least as important as designing the operations that are to manipulate it. The primary data is a model of the information that permeates the problem in the real world that the program is to solve. It is often useful to design the data before considering how to design the code that works on it. If one is using a more primitive programming language, one has to represent the information with numbers, text strings and possibly the Boolean values, True and False. With a more advanced language one can use more elaborate types, records and arrays, or invent one's own to suit the circumstances. Michael Jackson's system development method[9] recommends considering the entities in the problem in order to reach a useful data design. Data has value, structure (its type), access mechanisms and properties. Different modules may have access to different pieces of data. So different modules have various ownership and access rights; Dijkstra's principles of information hiding can assist the separation of concerns that help to make a program well designed, straightforward to understand and less error-prone.

I had more reservations about writing a standard about metrics. The original starting point for software metrics was to help predict the amount of effort it would take to produce a piece of software, how fast it would perform and how much storage space it would require. These ideas were fairly useful when programmers were using low-level languages, for much of the effort would be spent on coding from the design. But with high-level languages, the greater part of the effort was spent on the design; coding was largely automated by the high-level language compiler and other tools. Consequently, more complex systems could be built with the same effort. So the formulae of software metrics were continually being rendered obsolescent, and collection of all this metric data over many projects was to my mind, of limited value. Nonetheless, large quantities of metric data had been collected and was reported in the literature.

I reviewed the available literature, trade-offs, empirical formulae, proportions of effort on different tasks in the life-cycle, claims of different writers. Putnam[10] coined a "technology constant" which had a dramatic effect on cost and development time: productivity increased if one used more advanced technology. At the same time, observations were made on the processes involved in the building of software and laws proposed for patterns in these quantities[11]. Much of the work reported in the literature seemed to consist of measuring first and hypothesising afterwards, which made me very agitated. Experiments and measurements

[8] See Parnas 1972.
[9] See Jackson 1983.
[10] See Chapter 4 of Putnam 1980.
[11] See Lehman and Belady 1985.

should be carried out to test a hypothesis, preferably by trying to refute it. This was the modern view of scientific method[12]. It seemed to me that only when one had reached a certain point in the progress of a project could one make any predictions on its cost. As the state of the art advances, the less is the cost of the later stages because they become automated, hence, the total costs become less predictable. The total costs will also themselves become less, of course. So each advance in technology tends to wreck any investment you may have made in establishing a prediction method. For example, in one published text[13] I found that one project had been deliberately omitted from the authors' analysis because it showed a productivity an order of magnitude greater than the rest! They also admitted that, because "understanding the algorithm" is a more substantial cost than the coding, one gets completely different effort/code ratios for "easy" and "hard" algorithms[14]. By "understanding the algorithm", they must mean understanding the problem and theorising an algorithm to model it. At the time I quipped that investment in metrics was a force for sustaining mediocrity. As we introduce more technology into the software development process, we automate the parts we understand well, reducing the effort and cost required to achieve those parts. This reduces the total effort and cost, the remainder of which relate to the parts we understand less well and thus cannot predict. Hence, as we bring more technology into the development process we reduce total cost but increase our inability to predict it. I dubbed this "the Denvir effect", but the name did not catch on! All contemporary estimation techniques required as an initial input an estimate of the eventual program size. I wondered if there was any evidence that effort estimates based on program size were any more accurate than direct estimates of effort. Conte et al claimed there was[15] – but all the subjects in their study were students. Development environments could cause variations in productivity of factors up to 6 times[16].

Of course, any talk of productivity begs the question of how to measure programmer productivity; the crudest measure was lines of code per person-day, but in languages of different expressive power, the same number of lines represent vastly different amounts of functionality. There were several attempts to devise more sophisticated measures of productivity, by Halstead[17] in particular. Halstead's metrics counted the number of occurrences of "tokens", certain language entities, in a program rather than lines of code, among other things. But this can vary considerably with different programming styles, and did not take into account the scope rules which most high-level languages possess. Several other metric schemes were current, all of them with substantial limitations.

In my report I reviewed the literature and the state of the art of the time. Software metrics is a behavioural science akin to sociometrics or econometrics, perhaps classifiable as a "science

[12] See Popper 1972 and Lakatos 1976.
[13] See page 179, Conte et al, 1986.
[14] See page 210, §2 and footnote 3, Conte et al, 1986.
[15] See pp. 217-218, Conte et al 1986.
[16] See pp. 242-243 ibid.
[17] See Halstead 1982.

of the artificial" in Herbert Simon's terms[18] rather than a so-called exact science. I recommended not spending much on special metrics schemes and support tools, but taking some simple measurements to gain a profile of some of our typical projects. On the whole, I received positive reactions.

Integrated project support environments had been a strong technical aspiration since 1983. It had grown out of previous less focussed research on Computer Aided Software Engineering, CASE, tools, such as ICL's CADES. Researchers had spent much time investigating the desired properties of an APSE and of the more general IPSE. Environments became a driving obsession in software engineering for the next decade. A coordinated, integrated environment of tools for management and development should, everyone hoped, raise the level of software quality all over the UK and Europe. At the end of 1990 ECMA published a standard for PCTE[19] and in 1993 the PCTE Interface Management Board, PIMB, published the first general introduction to PCTE[20]. ISO adopted the standard in 1998[21]

The first step to achieve this noble aspiration was to define the interfaces between the environment and the tools which it supports. The specification of this interface comprised the Portable Common Tools Environment, PCTE. With this specification in place, any software developer could produce an environment which conformed to the specification, and tools could be ported from one proprietary implementation to another. Standardising the interface would assist the portability of tools. In 1986 many firms were involved in PCTE and its implementations: Bull, GEC, GIE Emeraude, ICL, Nixdorff, Olivetti, Siemens and others. The implementation of PCTE and the use of it was the topic of a number of European projects, some of them supported by ESPRIT: PACT, Sapphire, GRASPIN, and others.

Meanwhile across the Atlantic, the USA DoD supported a similar project, CAIS. The first version, CAIS 1, provided portability, data sharing and interoperability. CAIS A, the second version, provided more: data types, database schemas, bit-mapped screen support and security controls over access. Softech had a contract to implement CAIS A. CAIS was really an updating and amalgamation of previous work on the APSE. CAIS was started first but in 1986 PCTE seemed more technically advanced, although lacking access controls at first, and more advanced in development. CAIS had support from the DoD but little from US industry, whereas PCTE was getting massive industrial and governmental support. Formalising the PCTE interface and designing a PCTE over distributed hosts and target machines were examples of further PCTE projects. A version called PCTE+, providing additional features for security of access, was sponsored by the MoD and other European defence ministries and

[18] See Simon 1996.
[19] See ECMA 1990
[20] See Wakeman and Jowett, 1993.
[21] See ISO 13719.

their contractors. PCTE+ would be available for civilian use, so civilian software houses could contribute to its development. I was amused to hear the Italian MoD representative assert that they could not afford to fund PCTE without very necessary industrial sponsorship; in the UK the boot was definitely on the other foot. The organisations involved in PCTE grew to a plethora and the relationships between PCTE and other project support environment and CASE tool projects began to look like a metro map. Implementations of PCTE were, of course, themselves pieces of software, and so their continued development should be supported by PCTE where possible. Schemes of progressing through this bootstrapping process were much discussed. A central feature of PCTE was the Repository, a database to hold the tools and services which would interface with and "sit on top of" PCTE. A purposeful aim of PCTE was to support Object Orientation and Object management.

A PCTE implementation would not normally be produced on a bare machine, but on top of an existing operating system, on a host machine. At the time, the probable operating system was assumed to be UNIX, since that was favoured by software and other engineers. There was much discussion over intricate details, like the hosting of more specialised local tools on top of a "normal" collection of tools for software development. The first draft of PCTE+ was scheduled for August 1987 and the final one for December. We hammered out procedures for its evaluation and recording the result, all to be supported, of course, by PCTE.

Praxis encouraged all its staff to try to "sell". "We're all salespeople", they told us. Selling is a skill that does not come to me easily. I feel awkward asking anyone to buy anything, even a raffle ticket in an excellent charitable cause. Nonetheless, I got in touch with previous colleagues and told them of the fine qualities of Praxis. Slightly to my surprise, whereas many people had listened attentively to me when speaking on my own behalf as a proponent of particular methods and techniques, now that I was making a pitch on behalf of Praxis, I was mostly met with tolerant cynicism. Still, one of my contacts eventually led to a contract, albeit a very short one. A previous manager at STL, David Pitt, had become a director at a small firm, Renishaw, operating in the charmingly named Wotton-under-Edge in Gloucestershire. Renishaw made mechanical sensors of high precision. The firm has since expanded and covers many areas of machine tool components and metrology. I wrote a letter to David Pitt outlining the services that Praxis offered. Several months later he telephoned and I visited Renishaw. At the time their main business centred round a particular design of highly accurate mechanical sensor, based on a patented invention of the managing director. A probe was held in position by three springs and in electrical contact with three internal conductors. Any minute disturbance to the tip of the probe broke the circuit and the presence of an obstacle was detected. The probe was tipped with an industrial ruby to limit mechanical

wear. The internal design was such that the sensitivity and accuracy of the probe was 50 nanometres. The probes were typically attached to a robot arm driven by servos and controlled by computer software. In the entrance hall to the firm a probe on the end of an articulated arm was feeling its way round the surface of a teddy bear, and linked by software to a machine tool that was carving out an identical teddy bear – in brass. This demonstration was running continuously without supervision to impress visitors.

The manufacturing plant made use of Renishaw's own probes and software, in order to duplicate their own products. I was impressed by how few staff were needed to oversee the whole manufacturing process. There seemed to be just a handful of people watching over the automated processes, most of them eating sandwiches or reading magazines while keeping an eye on things. David explained the product and general design of the whole facility, and spoke with some admiration of the very clever software that their programmers produced to drive it all. I did not visit the software group on that occasion.

Fourteen months later I heard from them again. They wanted to talk to us urgently about their software team and its product. This time, Martyn Thomas, Praxis MD, and I went to speak to Renishaw's MD, David McMurtry and other senior members of the company. They wanted to know how much the software produced by their team was worth. During the conversation I began to understand that the software team was "out of control", doing what was required but not under control of the management simply because the management did not understand what software engineering was about or what the team were doing. A few days later I returned with two other Praxis software engineers and we spoke again with one member of management and then with the team. Relations between the two were clearly not good; some individuals were scarcely on speaking terms. I was almost certain I knew already what the answer to the question would be. The team were highly competent but lived a maverick existence. Standards of software development were almost entirely absent, because there was no member of management who understood the pressures and needs of software production. For the same reason the team felt isolated and consequently made their own rules: idiosyncratic decorations and habits of dress, a pet tarantula in the team leader's office, and no tedious regimes of documenting their software products and processes. They were also able to purchase all the latest high-tech equipment, for no-one above them in the hierarchy was in a position to judge whether it was really necessary.

The management was anxious that the software team would not cooperate in giving us information. In the event, they talked to us fully and frankly. Their relief at meeting someone outside their own group who understood their language was palpable. The team leader was an intense man of great charisma. He told us how they had decided to construct their own office furniture in order to acquire exactly what they wanted. They took a week to do so and I have to say the results were excellent. The meeting was a steep learning curve for those of us from Praxis, for process control, the use of embedded computers in manufacturing processes, had

its own terminology and technical culture. For example, when they referred to a PC, they did not mean a Personal Computer, but a Process Controller, which could be a programmable logic array or other small device that was essentially a small computer designed for process control. We spent most of the day with the software team before returning to the management later in the day, when Martyn Thomas, Praxis' MD, rejoined us.

Like anything else, the value of software is how much someone is prepared to pay for it. Without supporting documentation explaining how it should be used and how it works so that it can be maintained and upgraded, no-one would be prepared to buy it. Reconstructing the detailed knowledge of the software would probably need more effort than rewriting the software itself. Hence I had a highly unwelcome message to deliver: despite the firm having spent half a million pounds on the software, and despite the fact that it worked fairly well, it was worth nothing at all.

The meeting with management was a little uncomfortable, but they accepted our judgement. The reason they wanted to know was that they suspected that the software team might go AWOL taking the software with them or selling it. I thought and said that there was no danger of this, and that the team would probably get the software working well enough for a forthcoming exhibition which the firm regarded as a showcase for their products. We also recommended that they make organisational changes that brought the software team more under control and more recognised by having representation in management.

All these events happened in 1987, over twenty years ago. Today Renishaw are a high achieving company who give their considerable software expertise a high profile, being an essential part of their product range. I continue to be impressed by this UK firm. Their products are of great expertise and fine up to the minute technology. Their record of innovation and expansion to a world wide market is inspiring.

The British Computer Society, the professional society that had represented the interests of those working in software and hardware research and development for over twenty-five years, was changing to move with the times. They were deliberately seeking an industrial input to their policy making. They established a "Young Professionals" group. They started Quality Control and Career Development plans to fit with industrial employers' schemes. In all this, the society was moving away from its previously exclusive academic stance. However, membership of the BCS was to be an academic qualification and would lead to the opportunity of becoming a Chartered Engineer. Membership was by examination, but there was to be a route for "mature" candidates with some ten years experience after an honours degree exempting them from the examination. Anyone whose membership had lapsed could rejoin with no formality by paying a year's membership fee.

I left the BCS in 1964, but thought that I might rejoin. I telephoned their headquarters and was assured that all I had to do was give details of when I was last a member. When I gave the date, 22 years earlier, there was a pause on the end of the phone. All membership details older than ten years were kept in the basement; they would phone me back. They called back later to tell me that archives had only been kept since 1965. I would have to reapply using the mature entry rules. This I subsequently did and resurrected my membership.

Most of Praxis' work comprised relatively short-term projects. As a result, we all spent considerable time seeking new work, approaching potential customers and preparing bids. Sometimes other consultancy firms approached us with an opportunity for which they did not have all the requisite skills. Enator had spotted an opportunity. IBM Vienna Laboratories had implemented an interpreter for the language PL/1 and wanted a compiler developed for it. All Enator's compiler people were occupied on another project for the next six months. They proposed that, if we won the contract, Enator would take commercial responsibility for the job while Praxis would provide the technical effort. Praxis would have to agree not to pull out and bid via any other route. Enator would take 20% of our fee rates as a commission and administrative overhead.

I had a number of questions for IBM, mainly technical, about what subset and features of the language were to be included, the host and target machines, and why they were subcontracting the work anyway. After all, IBM Vienna Laboratories were a research institution with an international reputation for their technical expertise. I wanted to know if there was a definition of the language subset available at the time we were to make the bid. Numerous questions occurred to me, to none of which Enator had answers. What implementation language should we use? It might be possible to use PL/1 itself given that there was already an interpreter; the implementation could be bootstrapped. Was there already a run-time library? What documentation were we to produce, for example, user manuals? Did they have a validation suite (a suite of PL/1 programs which when successfully run on the compiler would constitute a criterion for accepting it)? A compiler is a complex piece of software and I had numerous other technical questions. We arranged to meet IBM at their Vienna Laboratories.

There was a lot of snow in Vienna in late November. Cars parked at the side of the road appeared as just large car-shaped snowy mounds. I thought they would be there for the winter, immovable. Eighty technical people worked at the laboratories in the Programming Product Development Centre. They were subcontracting the work because, once again, all their compiler people were engaged on other projects. They still had to get internal approval for the contract; it could yet be cancelled! Clearly, the whole proposed project was at an early status and subject to change. Various ambiguities in the work plan meant that we would be

best advised to do the work charging for time and materials spent as we went along, rather than as a fixed price job. The customer envisaged three or four people from Praxis and one or two from Enator, all working in Vienna for three months. They were agreeable to a time and materials contract, at least at first. They wanted corrections of errors after delivery guaranteed and fixed free of charge. This was unusual for a time and materials contract, but they were willing to negotiate; higher rates could be charged to compensate, or some such arrangement. Some flexibility could be accommodated. As for documentation, they would require an implementation guide and a user guide, but IBM had their own publication department who could produce finished manuals. I suggested that we could provide some part of the implementation guide in VDM, knowing that VDM had originated at IBM Vienna Laboratories. This produced some embarrassed shuffling of feet and an admission that no-one working there had any knowledge of VDM any more, and they'd rather not, please. Of course, we could use other means!

More meetings and discussions, mostly by telephone, ironed out issues of quality and commercial confidentiality, which were considerable. That was why they insisted on our doing the work on their premises. It was not even clear that the customer would allow us to produce any documentation in our own offices at Praxis. IBM had their own quality and test plans and these would have to be coordinated with those of the Praxis quality system.

Over the next few months and more telephone calls, after all the effort, nothing transpired; we had no contract. If IBM ever did the work, I think they carried it out using internal staff. This was a typical story and happened many times with many different customers; IBM was in no way exceptional, but these procurement efforts that led to nothing could leave one with a sense of some frustration.

ESPRIT had been operating for several years. Those of us who were enthusiasts for VDM sought some sponsorship from ESPRIT to set up an international special interest group to promote VDM. We thought that an international, Europe-wide group could accelerate the use and understanding of the method. We found a champion within the ESPRIT organisation, Karel de Vriendt, and his support enabled the initiation of a group that flourishes today, now independently of EC support and with a wider focus on formal methods generally. Formal Methods Europe continues to hold international symposia and itself funds a few small projects.

At the end of 1986 we had our first meeting in Brussels We drafted our terms of reference, a list of proposed activities, and planned the first VDM Europe symposium. This was held in Brussels in March 1987 and covered the history, experience of use, support tools, standardisation process and some foundational questions such as the precise mathematical

model of types in VDM. There were also a few tutorial papers. For a first symposium, this was a mature and balanced event[22].

Very soon after I joined Praxis I found that I was once again working on multiple projects: twelve in March 1987. We found email a great facilitator for communications, at least within the company. People would sit silently at their desks, exchanging notes with others who might be only a few yards away. It enabled the regimen of staying silent within the open plan office area; those who wanted face-to-face discussions were urged to book a small meeting room. But the teething technology of email was less reliable hence less useful between companies. Frequently messages would bounce back with spurious error reports like "unknown host" when one knew very well that the host was definitely present.

The British Standards Institution and the International Standards Organisation published standards in a huge variety of fields. Computer science and software engineering were becoming increasingly the subject of standards. Languages, protocols, and methods of development and quality assurance were being standardised. Many of these standards included definitions of languages and other technical items. What better means of expressing definitions could there be than formal methods? In the spring of 1987 the BCS held a meeting on the use of formal methods in standards. This meeting gave birth to a working group and Springer-Verlag together with the BCS published this working group's conclusions in 1990[23]. "Standard" practice and the technological avant garde were catching up with each other.

There was a lively exchange of messages on the local electronic news-board at Praxis. These days one might call this facility a blog. From time to time a discussion sparked off some hurt feelings, which time usually healed. One day someone reacted rather upset to a charge of sexism in something he had written. I thought I might mollify this upset by saying how we all found it difficult to change our linguistic ways after years of habit and avoid gender specific turns of phrase. I had distributed my own book on discrete maths for software engineers[24] to those members of Praxis who attended the various courses that Mel, Roger and I had given on formal methods. I said that I had tried hard to be gender-inclusive in my book, but I would buy anyone a drink if they found examples where I had not done so, the first in each case (I slightly feared an avalanche of discovered transgressions). I had an ulterior motive, to encourage people to reach for my book and read it! I began to be alarmed as I received three

[22] See Bjørner et al, 1987 for proceedings.
[23] See Ruggles, 1990.
[24] See Denvir 1986.

emailed examples within ten minutes. Fortunately no more arrived, so there was no serious damage to my wallet.

IPSE 2.5, in which the software research department at STL had shown an interest, was moving forward, with ICL, STL and the Universities of Manchester and Newcastle as partners. Praxis was fairly easily persuaded to join the consortium. The proposed environment in IPSE 2.5 would support both informal and formal methods of software development, management, formal reasoning, and the integration of all these activities. These five themes were allotted to different partners, with Praxis handling the management support. Bob Snowdon from ICL was chosen as the project architect and Anthony Hall headed the Praxis effort. Anthony was keen on, and knowledgeable about Object Oriented methods, and we determined to use an object model as a language to describe the management activities in a software project. In the process of a development, people carry out different rôles which are responsible for various activities. Activities can interact with each other, waiting for another activity to finish before being able to start, sharing information and so forth. Clive Roberts, ex-STL, had joined Praxis a the same time as I had, much to my surprise; I had no idea he was considering a change of job. Clive, also an enthusiast of Object Orientation, defined the language in which a development process could be described, having considerable discussions with Anthony and me along the way. We called this language PML – Process Modelling Language. I was to provide a definition of the semantics of PML.

If we were carrying out this project today, UML – Unified Modelling Language – would spring to mind as an obvious choice for the process modelling language that we wanted. But UML did not come into being until 1997, ten years after the IPSE 2.5 project.

I had great difficulty in persuading Praxis of the amount of time that defining the semantics of a new language under development would require. I managed to get the originally proposed six days increased to ten, and in that time I produced a first draft, much of it hand written on account of the extensive mathematical text involved. I could happily have spent three or ten times that amount of time and produce a more polished result, but the other partners in IPSE 2.5 seemed to find what I produced acceptable. I was particularly relieved that the academic partners approved of it. To this day I feel it may be the most intellectually demanding piece of work I have ever done, so to have had to squeeze it into ten days was a considerable rush. Parts of my handwriting betray my writing the script on high speed trains. A tip: sitting in the centre of the carriage equidistant between the wheels reduces the vibration considerably.

VDM was coming of age. Three main academic centres spearheaded the development at the Technical University of Denmark, the University of Manchester and Trinity College Dublin.

Industrial organisations were using VDM in live developments and researchers were starting to write computer-based support tools. The VDM language was originally designed for human readers, and contained many mathematical symbols that you cannot find on a computer keyboard: logical symbols like “and” $\wedge$ “or” $\vee$, “not” $\neg$, and set theory operators like “subset” $\subset$. Dialects of the language were beginning to emerge, including computer-readable variations. It was time to standardise on the language so that publications used the same notation with the same meanings, and so that specifications written in the VDM language could be moved across tools without too much recasting. We formed a committee under BSI rules and set about defining a standard that would be published by the BSI. The language would be known as VDM-SL – the VDM Specification language.

All the tool designers would need to be involved in the standardisation effort, as well as the universities that were teaching VDM on their undergraduate and graduate courses. Industrial users contributed to the committee’s work, notably CEGB – the Central Electricity Generating Board, who were the nationalised predecessor to the now multiple private electricity companies in the UK – ICL and STC. We needed to agree on the use of terms, often with duplicated meanings, circulated working papers and notes, and met every few months. Because several institutions were writing support tools, we also needed to agree about the meaning of the language elements, that is its semantics, so that different tools would agree about the validity of VDM-SL scripts and their properties. One of the most important properties of a VDM specification is its context conditions or proof obligations; these are the set of conditions that have to be proved in order to demonstrate that the specification is self consistent and that a given implementation satisfies it. People were trying to produce tools that would generate these conditions. Academics in several countries were prominent in the attempt to define the semantics of VDM, and some of the meetings went into deep mathematical discussions, including over just what flavour of mathematics was best for modelling some of the concepts. The work on the semantics became the focus of a subgroup, a semantics review board, which gave advice and input to the standardisation committee.

The British National Physical Laboratory worked hand-in-glove with the BSI and two of their own researchers became interested in VDM too, contributing to the standardisation work and providing a useful link to the BSI as well. We explored new features too. To specify a large system, the VDM script could become awkwardly large to manage. Some features for breaking a specification up into modules would be an asset, but had not been part of the original language.

The technical work leading to standardisation mostly took place in 1986 and 1987, but final details lingered on, together with the procedural processes that seem to dog all international efforts of this kind. Promoting the standard from a British one approved by the BSI to an international, ISO standard added a few more months to the process and it was not until 1990,

after more than 130 working papers and many discussions, that VDM-SL became a BSI standard and 1996 when it achieved ISO status[25].

FACS held its AGM every May and in 1987 its membership was still fairly small. But it subsisted with a grant from the Alvey Directorate and £500 worth of services from BCS. We remained solvent and the newsletter, FACS Facts, continued to improve under the editorship of Roger Shaw. At Professor Bernie Cohen's request FACS were editing a special issue of the Software Engineering Journal on formal methods. We agreed to hold events on term rewriting, algebraic approaches and the specification and verification of communication systems during the forthcoming year. We proposed and agreed on fees for membership and attending meetings. We decided to negotiate with EATCS, the European Association for Theoretical Computer Science, for reciprocal arrangements, reduced mutual membership subscriptions and such-like. These were entirely typical matters that were agreed at any FACS AGM. I became secretary at the 1987 meeting and remained so for some years. I had been secretary of VDM Europe for some months and felt that doing the two tasks would be much less effort than twice that of doing one of them. I had made a routine for writing minutes and establishing actions and recording their progress that would work for both.

VDM Europe itself pressed on, coordinating the British standardisation effort with bringing a focus on VDM into the next year's ESPRIT work-plan, education material, and topics for subsequent VDM conferences. We held the second conference, VDM88, again in Brussels, complete with an exhibition of tools, computer-based demonstrations, applications, information stands and posters.

In September John Cooke and Roger Shaw approached the publisher Springer-Verlag UK to propose starting up a new international journal on formal methods, specifically as a flagship publication of FACS. We already had a newsletter, FACT Facts, but this was not a journal of serious refereed papers; it was more like a parish magazine with news items and so forth. Springer were enthusiastic about the idea, so several of us set about making the journal happen.

Of course, Mel Jackson, Roger Shaw and I continued to give VDM courses at Praxis, and to "the public" through and on behalf of the NCC. The arrangements for doing this took some negotiating, but after a few meetings were agreed and we gave the courses. These usually had a small number of delegates, but were an interesting change from giving courses to mostly one software organisation.

[25] See ISO/IEC 13817-1 1996.

The British Computer Society has many Special Interest Groups. I was interested in the Disability SIG because I believed that computers and software provide a great but unrealised potential for assisting and enabling disabled people. I met the group chairman, Geoff Busby, who was himself disabled with cerebral palsy and who was seconded from GEC to the BCS Disability SIG. The group aimed to increase awareness and to enable disabled people to be employed in IT, using specialised equipment such as specially adapted keyboards and user interfaces. Where possible, however, they preferred to use standard equipment, since in general it was difficult to modify. There was cooperation between different disciplines – robotics, optics, computing, AI, software and electronics. There was £¾ million funding offered by the EEC but at the time that was not matched by any UK grants. Some simple facilities were needed: page turners, text and graphics input on to disc, feeding machines, machines to assist in driving a car, simulation software and even arcade games.

The computing industry at large was concerned with the issue of software quality. Praxis was one of many software engineering houses that had its own quality system, procedures which had to be followed when building a product. "Quality" was defined variously as "Conforming to the Specification" and "Fitness for Purpose", a phrase which has recently been applied negatively by politicians and some media to institutions and individuals. Setting up a quality system requires substantial investment in training and establishing practices. The cost of developing software could consequently increase. Was the payback worth this cost? The benefits could be reduced by risk of failures and all the consequential damages. The BCS Software Engineering Technical Committee held a meeting where some experienced speakers talked about these questions. How can one measure these costs and benefits, time-scales, levels of reliability, maintenance efforts? The meeting sparked off considerable debate.

Honeywell got in touch with Praxis, wanting to hear what we could tell them about formal methods. They might want to become capable in formal methods themselves. I prepared a short talk, explaining the underlying theory, the kind of mathematics FM practitioners would need, and compared different methods against some criteria. Some worked at a greater degree of abstraction than others, which usefully defers design decisions to later in the development. Some could cope with developing large systems better than others. The experiences of actual use varied. User manuals and texts were more supportive for some than others. Some could define concurrent and communicating systems, others only sequential ones. The route from specification to design could be well documented or not at all.

Honeywell described their organisation structure and were attentive listeners. But two months later they telephoned to say that they had not won the work they were bidding for, so would

not be going ahead for the time being. They acknowledged that they would need to know more about FMs in time.

At one of their meetings, the BCS Software Engineering Technical Committee rather casually nominated me as vice-chairman, and without any other contenders, suddenly I was appointed. Various specialist groups were represented on the committee, the Database group, the Object Oriented group and others. The SETC attempted to coordinate matters between them: streamlining the production of the SIG newsletters and so forth. Other groups not connected to the BCS carried out related work. The IEE in particular had a subcommittee on software engineering, which had a parallel rôle to the SETC. Indeed, I was a member of that too, so it was natural for me to be a BCS representative on it and to report back when necessary. Independently of the BCS and the IEE, a UK Logic Programming group was being set up as a branch of a wider international group. We asked them to consider being affiliated to the BCS. The CCTA, a quango, supported the use of SSADM, a structured systems analysis method developed a few years earlier. Then of course there was the DTI Alvey Directorate. The SETC prompted joint working parties between these several organisations and itself, to reduce duplication and encourage coordination. It was better if we all told the same story.

The Alvey Directorate sponsored several projects to develop and propagate the use of software tools to assist different phases of development. From time to time they would hold a conference at which these various projects described their work and exchanged news of their progress. Alvey sponsored projects on predictive software metrics based on formal specifications, building software libraries, quality assurance and reliability and project support environments and metrication. The aim of metrics are in general to predict useful properties of the development, such as how long the work will take and how likely it is to have faults, from characteristics available at the beginning of the project. In a, perhaps oversimplified, view of a project development the drafting of a formal specification precedes the design and coding in a programming language. Traditionally metrics are derived from the code or its design itself, so basing these on a specification should enable predictions before quite so much of the work has been done, a clear advantage if one is trying to predict features like the total effort. Other projects examined the overhead, if any, of effort if formal methods are used.

The corresponding committee on Software Engineering in the IEE organised their own colloquia, something that the IEE effectively did and does a great deal. High integrity systems, the costs and benefits of quality assurance were typical topics. They advised the DTI on criteria for assessing research proposals for the second phase of the Alvey initiative and

regularly discussed engineering educational issues with British universities. They welcomed the invitation from the BCS Software Engineering Technical Committee to collaborate.

One of the most compelling reasons for using formal methods in software development is to prove that a program is correct. This means proving that the program does what it is supposed to do; for this one needs a watertight, unambiguous description of what the program is to do, that is a formal specification. The specification needs to be formal, that is mathematical in form, to be unambiguous and for a logical proof to relate to it. This is one of the principal motivations for methods such as VDM, Z, CSP and the like.

There are different kinds of formal methods. VDM and Z are examples of model-based specification languages. In these languages one defines what the program is to do by making a model of the program's function in terms of set theory and logic. One can then prove whether or not a given program truly fulfils the function defined by the model. One difficulty is that considerable mathematical skill is needed to derive proofs of correctness. The vast majority of software engineers do not have these skills. A dream of adherents of formal methods is to produce an automatic system which can generate a proof, given a formal specification and a program which is claimed to implement it. This is particularly difficult with model based methods, partly because there is an wide choice of ways in which the actions of a program can be modelled, even in set theory and logic. The best that can be done is to produce a "proof assistant", a program which interacts with the user to generate a proof. The user gives hints and guidance to the program, which does the mathematical donkey-work.

There are other kinds of formal method, in particular, algebraic methods. These are based on the mathematical topic of Universal Algebra[26]. Instead of modelling the program's functions with set theory and logic, an algebra is defined whose data types and functions model those of the desired program[27]. It is much more feasible to design automatic proof systems which generate proofs of program correctness related to algebraic specifications, although it is by no means easy. Such proof systems rely on rules of *term rewriting*: a term in algebra or logic is an expression, which can be a proposition. With a defined algebra it is possible to derive rules for rewriting terms without changing their meaning. If the term is a proposition, this means that the truth or otherwise of the proposition is unchanged, i.e. it is deducible from the initial term. By using various strategies a sequence of term rewrites can be generated that proceed from the axioms of the algebra to the desired target term, that is the theorem.

Automatic term rewriting systems have been devised and are continuously researched. Some of the first canonical systems were based on the Knuth-Bendix completion algorithm[28]. There

[26] See Cohn 1981, for a comprehensive text on Universal Algebra.
[27] See H. Ehrig and B. Mahr, 1985 for a tutorial text on algebraic specifications.
[28] See D. Knuth and P. Bendix, 1970.

is, however, a very large step to take from a mathematical algorithm and a computer system which uses it for such a sophisticated purpose as a proof generator, and a great deal of research work was done and continues on this problem. Nonetheless Knuth-Bendix, term-rewriting and algebraic specifications were of intense interest in the late 1980s, and continue to be so. The University of London Royal Holloway and Bedford New College in association with the London Mathematical Society, Hewlett Packard and the Science and Engineering Research Council held a stimulating conference on this topic, "Algebra and Automated Deduction", in January 1988. Not only academic institutions were active in this quest. Hewlett Packard and the Rutherford Appleton Laboratory were both developing proof systems for algebraic specifications - AXES from HP and ERIL from RAL.

One snag persists. Devising model-based specifications comes more easily to most software engineers than devising algebraic ones. An abstract algebra is a step more abstract than a model in set theory and logic. The latter is not far away from a specification written in the more traditional methods of database technology or systems analysis.

I was flattered to receive an invitation to give a seminar at Queen's University Belfast. I talked about the uses of mathematics in software development, in modelling the problem area that the software addressed, the semantics of the program languages used, the deductive system needed for proof by construction and the theories of computation. My employers, however, were not impressed. Please consider what use this is going to be for company, they said. I offered to visit some firms in Northern Ireland with whom we had so far not made contact, such as Harland and Wolff, the shipbuilding engineers. All engineering firms were using computers for various purposes by then. But Praxis were not interested. This surprised me, since they frequently said that all their software engineers should also be salespeople for the company. I considered, decided that the trip would be good exposure for the firm, and went anyway, but without contacting Harland and Wolff or any other potential clients. My audience came to life, taking notes assiduously, when I concluded with a mathematical syllabus that would be useful for industrial programmers.

In 1988 Praxis took part in an ESPRIT project called VIP, VDM Interfaces to PCTE, along with other institutions from Britain and the Netherlands. Both PCTE and its interfaces were subject to international standards, the latter being managed by PIMB, the PCTE Interfaces Management Board. Praxis made contributions mainly during 1988.

At this point after numerous collaborations with academic institutions, I frequently wondered whether the grass was greener on the other side of the fence. Bernie Cohen had crossed over

from industrial STL to Brunel University, and asked me if I would like a visiting position there. Mel Jackson had long previously crossed the other way, from Hatfield Polytechnic (later the University of Hertfordshire) and advised me against the idea. But I thought his experiences were probably not altogether typical, involving a very heavy teaching load in a non-research environment. I would have limited duties and would be on release from Praxis for one day per week. I discussed the arrangements with Professor Pat Hall, who had been chairman of the BCS Software Engineering Technical Committee. He explained the details of my proposed post: half a unit or one unit of the fourth year undergraduate and MSc Course, a lecture course on the formal development of software, my position as an Associate Reader. So I started on the second Thursday in October 1988, and would carry on for a year. There were related courses on System Software and Automated Reasoning. My course was to include "practical skills": writing specifications, refinement, VDM, Z and Equational Reasoning, Denotational Semantics and Concurrency. My first lecture would be on 13th October and I had to define an exam paper by Christmas. This was to be a 3-hour paper comprising eight questions of which the candidates had to attempt five. The MSc students would also do projects in their third term. These could arise out of the course. I was to produce sample exam questions for the students in addition to the real exam. The exams had to be reviewed and subject to scrutiny by an external examination board, which also reviewed the assignment of resulting marks and degree classes. I was irritated by the insistence of the department head, who had a philosophical rather than a scientific background, that I could not award a mark of over 90%. Only an Einstein could get such a mark, he said. I resisted reminding him that Einstein had difficulty with being accepted by academia in his early days, and presumably did not do well in his exams.

So I designed the lectures, gave them, set the exam, reviewed students' projects, attended meetings including a near-vitriolic one with the external exam board, marked and classified the students' submitted papers, and supervised several students. I found it a considerable strain, for I felt strongly that the career future of these students depended on the quality of my lectures and the accuracy and fairness of the exam paper and my marking of them. The range of performance by the students was very wide, the worst showing that the individual had profound depths of misunderstanding and the best performing better in the exam than I could have done myself, even though I had set the questions. The Computer Science department also held regular seminars which I usually attended. Many years later I was delighted to observe that one of the former students on my course had progressed to being a professor in the very topics that I had taught.

Several people in Praxis thought that my being away for one day a week had detracted from my career prospects within the company, but I was glad to have had the experience.

I continued as a member of the committee of BCS FACS while I worked at Praxis. As a special interest group of a professional society, FACS organised several events each year, including a regular Christmas workshop. We held meetings on Formal Methods in Software Engineering Education; Concurrent Systems; Term Rewriting; Graphs, Grammars and Automata; Refinement; B; OBJ; AXES and ERIL; Temporal Logic; LOTOS. The Refinement meetings became a significant series of workshop events, continuing for a decade. We published a newsletter, FACS FACTS, at rather irregular intervals. It still thrives, now being published on the web. We established relationships with other organisations, such as EATCS (European Association of Theoretical Computer Science) and the IMA (Institute of Mathematics and its Applications). Perhaps the most significant achievement of FACS in the years 1988-1989 was the initiation of a scientific, peer-reviewed journal, the Formal Aspects of Computing Journal, FACJ. It was largely the inspiration of John Cooke, and Cliff Jones rapidly added his drive and influence. John Cooke became the Associate Editor and Cliff Jones the Editor in Chief. Professor David Gries was the initial North American Editor. I was privileged to be invited on to the thirty-strong Editorial Board. The first issue came into print at the beginning of 1989, and the journal thrives to the present day, the flagship journal of BCS-FACS and, now also, FME, published by Springer London.

In March 1988 I visited the firm Program Validation Limited on behalf of Praxis, to find out what were their technical offerings. At the time their main product was the SPADE proof checker, available then for £5750. The price included a two-day course for two delegates. A language called FDL, Functional Description Language, was integral to the SPADE proof checker. FDL was essentially an algorithmic programming language with provision for assertions. The methodological approach of SPADE was post-hoc proofs on code. On my visit the presenters from PVL put some emphasis on their experience of proving assembly code programs. They seemed to have less experience then of proving correctness of high level language programs, although it looked as if that should have been easier. It also seemed to me that one should quite easily use the proof checker to generate proofs of other kinds of theorems, such as some of the proof obligations resulting from the writing of a formal specification or doing a refinement of one. (In fact some years later the SPADE principles were successfully applied to Pascal and Ada).

The method used with SPADE consisted of six steps:

1. Produce a FDL model of the source text. An automated tool was provided for translating SPADE-Pascal into FDL. SPADE-Pascal is a subset of Pascal, omitting those features which were considered to mitigate against proof generation, such as variant records, and functions as parameters to procedures and other functions.

Assembly code could be translated using a combination of manual translation and machine assistance using a tool constructed for the purpose.

2. Perform a flow analysis using the SPADE tool.
3. Construct pre- and post-conditions and assertions. The pre- and post-conditions can be derived from the formal specification, if one exists. All program loops must have invariants stated for their bodies, which must be sufficient for the prover to deduce the post-condition for the loop. These are embedded in the FDL text.
4. Auxiliary functions and intermediate assertions ("lemmas") can be defined to facilitate the path of the theorem prover and the statement of the assertions to be verified.
5. The resulting FDL text is processed by the verification condition generator. This produces conditions to be proved, in order that the proof of the program follows from the stated pre- and post-conditions.
6. The proof checker is then used to prove all the conditions generated by the verification condition generator.

The proof checker was interactive. It worked on one verification condition at a time. As individual conclusions are proved, they can be added to the list of hypotheses, under the user's control. If the prover gets stuck, the user can define a sub-goal to be proved. When that is proved, it can be added to the hypotheses in order to help prove the main conclusion. The proof checker used first-order predicate calculus employing classical logic in a natural deduction style of inferential reasoning.

The inference engine within the checker operated on a database of rules. There were about twenty files of rules in the database. To be adept at using the checker one would need to know about most of the rule-bases and when they were likely to contain a rule that was applicable to the proof being constructed. For example there was quantifier elimination, substitution, induction etc. The rule-base could be extended. By defining an "undefined" data-type, defining functions over it and then defining rules over those functions, one could effectively produce one's own algebraic data-type. In that way, the type system was extendible, but with some constraints. Higher-order functions were not possible because the prover uses first order predicate calculus. PVL hoped to establish a user group who would exchange libraries of proof rules. Using the prover produced a proof log, showing the conditions and conclusions being proved and the rules used at each stage. There were restrictions on the use of the prover, for example the verification generator could not handle recursive functions. This would have caused the system to have only limited application to compilers, for example. But I believe that later the proof rules for induction were extended so

as to incorporate proofs of recursive functions. Some components of the proof checker itself had been proved correct.

On my visit, the proof checker was demonstrated. It operated with, to me, surprising rapidity. On the whole I was impressed. The system seemed to be quite effective, efficient and easy to learn within its rather limited constraints.

The audience at this seminar were all concerned with safety critical applications. Indeed, only one person apart from myself, someone from the UK CAA, was not either from a military organisation or a military contractor. In conversation with these people I learned that there was a general consensus in safety critical applications that a number of programming techniques taken for granted elsewhere should be mistrusted or avoided. These included compilers, recursion, even procedures and functions, and high-level data types. “Straight-line code” was to be achieved if possible. One attendee said that where he worked, one had to get a high-level signature to program a loop! I thought that this was surely barking up the wrong tree as a means of seeking confidence in the correctness of programs. One may be able to prove a program correct with respect to a low-level specification, but one is just shifting the lack of confidence to a different part of the development process, and furthermore one which is less amenable to machine support. In another conversation I raised the question of how the user had to have a considerable understanding of mathematical proofs in order to use these tools effectively. People generally agreed, and considered that it was probably necessary to have a mathematics degree or to have gained the knowledge in some other way. This is a conclusion I had been reaching from reading the literature on a number of other theorem provers, and SPADE seems easier to use than most.

SPADE was subsequently developed further and evolved into the SPARK system, and included SPARK-Pascal and SPARK-Ada.

At this stage there were eight current or completed projects being done by Praxis which used VDM and other formal techniques. They were:

- Daffodil, an office communication system done under contract for ICL;
- IPSE 2.5: Praxis’ part in this project included the formal definition of a process modelling language. The definition used some techniques from VDM, and others from denotational and axiomatic semantics.
- VIP: VDM Interfaces to PCTE. This ESPRIT project to define the interfaces to PCTE was done entirely using VDM and was one of the largest VDM specifications ever produced at that time.
- PCTE+, enhancements to PCTE: a collaborative project in which Praxis was to define the semantics.

- Factory Controller: a project under contract from ICL. CSP was used for defining the interfaces of the distributed system and VDM to specify one component.
- ELLA VLSI simulator and development system. ELLA was Praxis' proprietary language for simulating electronic hardware systems. Part of the development system was defined in VDM.
- NEA (Northern Examining Association). Praxis was implementing a new examination administration system for the change-over to GCSE exams. The system test was being specified in VDM.
- CASE (Support system for SSADM for the CCTA). The 'Z' formal method was being used to specify the infrastructure, that is, the underlying database support.

As regularly happens with industrial projects of any kind, not all of these projects came to fruition, nor did all of them continue to use the same methods that they did at the outset. But a decent proportion of them were successfully concluded and delivered. Also, of course Praxis was not the only industrial firm to use formal methods. Even at the time (1988) other firms were contributing significantly to bringing them into practice, notably IBM, the Danish firm IFAD and others. By 2003 some thirty firms formed the initial founding membership of the Formal Techniques Industrial Association.

From industry's point of view, the whole purpose of using formal methods was to improve the quality of delivered software. The costs of correcting errors was becoming massive. The US Department of Defense had produced a statistic of $1009 per line of delivered, correct code. In June of 1988 I contributed a paper to an IEE colloquium on Software Quality Assurance. Brian Oakley, who then headed the DTI Alvey Directorate, led with issues of the day, and other papers were given by people from several supplier and user organisations (the latter including a speaker from the London Stock Exchange). The event included an exhibition of software tools supporting quality management. The next month I was chairing a session in the Software Engineering 1988 conference, where Praxis had a poster in the exhibition.

The work of the Alvey Directorate ran for five years from mid-1983 to 1988. The total cost of £350M was funded jointly by industry and government. Many projects involved academic partners, whose contribution was funded by the SERC. The Alvey Programme had come to an end. David Morgan, whom I had met twenty years earlier when I was at Elliotts, was the director in charge of the Information Technology division of the DTI, which division had so to speak hosted the Alvey Programme. All the projects had been concluded and their results delivered and reported on. But a rationale of the programme was that the communication between industrial and academic partners, which had perhaps only been rivals before, would

produce informal communities, with less tangible benefits to industry. David wanted to know what "secondary achievements", not yet published, had resulted from the programme. Were there perhaps themes across projects, or had duplicate work been avoided? David brought together a team of four to study this question and report back after three months. Naresh Mohindra from PA Consulting, already seconded to the DTI, was experienced in human factors. John Llewelyn, ex-STC, specialised in industrial applications. I would cover formal methods and software engineering. Professor John Buxton, later at Kings College, London, would be wide ranging, determining the general structure of the project and defining its overall roadmap. Our reporting point in the DTI would be Graham Mackenzie-Washington, someone I would come to know very well.

Although David Morgan gave us no instructions as to how we were to organise ourselves, we immediately fell into a tacit, mutually agreed working structure. John Buxton took the lead, I adopted a second in command rôle, and the other two members arranged themselves in order. I had a strange feeling of instinctive compulsion, as if we were a small pack of dogs inevitably following our inner evolutionary imperatives. We worked together very well. We would look for "secondary achievements" in four categories. Did any new projects arise involving the same partners in cooperation or in consultative rôles? Did any technical transfer arise, in particular of people from the partners to other parties? Was there any stimulus to the partners' infrastructure? Was there any technical spin-off? Technical transfer could take place through the giving of courses, distributing tools, transfer of people, and other Alvey or ESPRIT projects. Cooperation could take the form of conferences, workshops, papers, special interest groups, consultancy. Additionally, we would enquire if the initial aims of the project had changed. We would focus on a few projects, visiting them in pairs at first, then singly as we became more adept and established a firmer modus operandi. We devised a list of seven criteria for selecting projects to investigate. As well as visiting the projects, we would study the deliverables which they had sent to the DTI and which resided in files in the Department.

The DTI finally approved the contracts to engage the services of the four of us some while after we had started the work. Praxis, as would any software house that need to maximise the revenue-earning effectiveness of its staff, had by then put me on other work. So I had to announce that I would now be otherwise engaged for the next two weeks, something the other members of the team accepted with understanding. And of course, being a senior member of staff, I had to draft the letter of acceptance from Praxis to the DTI.

The project was to go for three months. We agreed with David Morgan that we would produce an interim draft of our report in two months. We could then make a case for extending the work if necessary. We selected the projects to visit, drew up a questionnaire to send them in advance, decided on a calendar. David Morgan wrote a standard letter to the project managers to ease the way for our contacts.

We carried out this project quite intensively over the next few months. I visited half a dozen projects or more, which was quite a fascinating experience. At one point there was some dilemma over issues of confidentiality. The Praxis quality system demanded that my reports to the DTI should be reviewed by another Praxis staff members in order to ensure its quality: accuracy, professional integrity, conformance to our contract with the customer (the DTI in this case), etc. But because my reports were relating to work done by other parties funded by the DTI, I was also under obligations of confidentiality to the DTI and had signed an agreement not to reveal any information about the project to anyone apart from the DTI. This latter obligation indeed had implications for the confidentiality relationship between the DTI and the partners in the projects which were the subjects of the study. Praxis and the DTI struggled with this issue for some time, both sides digging their heels in to a degree. I felt that this project was unusual in this respect and that Praxis ought to relax their normal procedures, but they did not agree. In the end one other member of Praxis staff signed the confidentiality agreement with the DTI for the Secondary Achievements project, our Quality Manager Chris Miller, and he rapidly reviewed my reports before I sent them to John Buxton and on into the DTI management. As far as I remember, Chris himself was very relaxed about the process, and never needed to query any issues in my reports. All along I felt that the "problem" was a storm in a teacup.

We finished the project, and our conclusions, reluctant I think, were that there were no significant "secondary achievements". In looking for them, we found ourselves scraping the barrel to find anything. There was a telephone conversation here and there, a few papers published, but nothing amounted to very much. John Buxton's leadership helped us come to an honest conclusion of a pretty much nil result. I must admit, left to myself I would probably have strayed a little way into the error of saying what the customer wanted to hear.

At the 1988 AGM of BCS FACS, we celebrated the group's tenth anniversary. Dan Simpson, a former chairman of FACS, gave a short talk on its history, describing it as a ginger group. I liked this description: FACS was indeed, I thought, in the avant garde of software engineering, and I had always enjoyed being in the vanguard of any movement, ready and willing to stir up controversy and shake people out of their comfortable, conventional ways of thinking. At the next Christmas workshop, Mike Shields gave a day-long account of his research into automata theoretic models of parallelism. His work was in later years to lead into significant advances in unifying theories of parallelism[29]. FACS were also conscious of the need to teach formal methods, and the difficulty found in doing so, so we held another meeting that year on "Explaining Formal Methods".

[29]See Shields 1997

The process of deriving a specification or design from a more abstract one was called "Refinement", a term coined, I believe, in the early 1980s by Michael A Jackson[30], who is a visiting professor at the Open University and the University of Newcastle. The process of refinement was very important, one could say the raison d'etre of formal specifications: they were there in order that they could be refined into correct designs. The steps taken in refinement are mathematically based processes, all preserving correctness with respect to the more abstract specification. FACS decided to hold an event on refinement at the 1989 Christmas workshop, and this was successful enough that it became the first of a long series of Refinement Workshops that is still extant, the most recent at the time of writing being held in November 2009 in Eindhoven.

In the widespread effort of propagating formal methods, Praxis in collaboration with Rolls Royce, with funding and support from the DTI, produced a video explaining their benefits and a broad brush description of their processes. The text was narrated by Eugene Fraser, whose voice was extremely well known to radio listeners in the UK and instantly recognisable. Before the days of the DVD (and any successor medium), the video was produced on VHS cassettes.

On 23rd June 1988 the Management Services Division of HM Customs and Excise in London asked Praxis to bid for a piece of work. This Management Services Division was in effect an internal software house. They wanted a requirements analysis done for a particular project; they would continue with the design and implementation themselves. Their own project management of their internal projects was impressive: they were using a variety of sophisticated computer based tools to support project management. They wanted comprehensive information from Praxis in our bid: a profile of the company and its experience, CVs of the staff who were to assigned to the job, our business methods used to do the work, a full proposal with a variety of cost options, recommendations of procedures, software packages and hardware needed to do the requirements analysis, and what method of business analysis we would use. This last item somewhat nonplussed me, because we didn't use a "business analysis method". We just talked to the client and produced a plan of how we proposed to do the work. I had questions for them: how did they determine the cost effectiveness of their current projects? What recording systems did they use, e.g. time-sheets etc.? Did they use systems of project budgeting, estimating and monitoring expenditure, assessing progress, setting milestones? They in turn wanted to have a list of our people and their availability, skills, projects and resources. They wanted our bid by June 10th, two and a

[30]See e.g. M A Jackson 1983

half weeks away. I have found that government departments and agencies are among the most demanding of clients, and this response from the HMC&E was quite typical.

As usual we didn't get the job, probably because the work was never carried out.

One credential of the maturity of a computer language is that it becomes the subject of a standard, approved by one of the standardisation bodies such as the BSI or the ISO. From mid-1988 after a great deal of preliminary work a number of us began to prepare a draft standard for VDM for submission to the BSI. This required agreeing with users and suppliers of VDM tools the exact syntax and semantics of the language. This was no easy task, not least because there were three or four developments in usage and tools for the language all going on simultaneously, in Denmark, the UK, Ireland and more recently Japan. There was an imminent danger of up to four different dialects, something none of us wanted! Technical meetings and discussions had been going on since 1986 and finally in October 1988 a proposal was put before the ISO (BSI and ISO work hand in hand with many standards). The ISO meeting rejected the first proposal, but with VDM Europe's support we were reasonably confident of its passing the next time. At least two forms of the syntax were defined ,one for publication of specification texts, called the mathematical syntax, and the other being a machine-readable ASCII version for input to support tools. The discussions within the VDM BSI committee revealed areas where the intended meaning was not altogether clear, and there was much argument over whether extra features with which there was less experience, such as the ability to write specifications in modular form, should be included. Could the two forms of syntax be mixed within a single specification, for example? An abstract syntax was defined which expressed the syntax of VDM, shed of any lexical decisions about how elements were printed on the page.

By the spring of 1989 three parsers for the machine-readable VDM syntax had been written, in yacc by the DDC, in Bison (a compiler-generating tool) by NPL, and in ProLog by Brian Monahan. All these parsers reduced the concrete syntax to the abstract form before further analysing it. LPF, the Logic of Partial Functions, was used to define the semantics of pre- and post-conditions of functions and operations in VDM. LPF is a logic devised by Cliff Jones for VDM, and indeed was inspired and necessitated by the ideas in VDM.

In August 1989 a separate subcommittee, the VDM formal semantics review board, began to concentrate on the semantics and met for the first time in Lyngby, Denmark. Andrzej Blikle and Dines Bjørner were the leading lights of this effort. Some advanced mathematical ideas were needed to cater for some of the features of VDM, and extensions were defined to Scott domains[31]. The wheels of the standards bodies ground slowly, but VDM became a BSI and later an ISO standard in 1990.

[31]See Scott 1976 and 1982

In several of the bids for contracts submitted by Praxis, the client asked what business analysis method we used. We did not have a defined method, but just interviewed the client, formed a view of the requirements and drafted a statement of them, then replayed that back to the client, discussing and amending it until an agreement was reached. Then we would mutually sign this requirements document and refer to it in the contract. I had several times been in a meeting with a prospective client in which they asked what method of requirements analysis or business analysis we used. Having to waffle in reply was embarrassing, so we tried to find out what other software houses did at this stage of a development task. The firm Oracle had a method of business analysis which used Entity-Relation modelling, a technique used in database work, incorporating a pictorial data model and a data dictionary. The "Requirements Capture" process consisted of interviewing the client and a series of feedback meetings. There were no fixed questionnaires or forms. They claimed that the feedback produced a reliable model, and they would then go and write the system specification. It seemed to me that Oracle did not do anything much different from Praxis, except that they had names for the various components of the process and used a database paradigm, not surprisingly since databases were the principal technical offering of the company. Praxis used SADT-SSADM as an in-house design method, so I wondered if we could similarly use that as the basis of a business analysis method, or indeed use the set theoretic and logical methods of formal methods as a basis. I worked for a time trying to devise such a business analysis method based on set theory and logic, but never got very far. It would have been a substantial task.

Meanwhile I continued to give formal methods courses along with Mel Jackson, Roger Shaw and Anthony Hall. We gave some courses through the NCC to all-comers, not necessarily Praxis customers or collaborators, and occasionally in conjunction with Manchester University. We constructed a variety of courses: Overview of VDM, Overview of Formal Methods, VDM for Software Engineers, VDM Workshops, Z for Software Engineers, Discrete Mathematics for Formal Methods. During the construction of these courses, Anthony Hall formulated his "Seven Myths of Formal Methods", which became the subject of a paper in IEEE Software[32] and acquired some fame. We gave in-house courses to IBM, Honeywell, the Civil Aviation Authority and CEGB. The IBM training department incorporated our FM courses into their own training programme, offered on demand to their internal departments.

[32] See Hall 1990.

One can go to great lengths to make sure that the software that is designed and written conforms to stated requirements. What often, and especially today, remains a significant problem is that the stated requirements don't meet the real need. The customer frequently underestimates or simply misunderstands what the end users are expecting or how they will try to use the product. This can be a technical issue, where the amount of "traffic" of enquiries to be handled by a system is underestimated, or a more human one of how operators work in reality, for example. Either way, the problem is that of knowing what is the environment in which the system is to operate, and this determines what are the actual requirements for the system. Discerning this at the outset of a development becomes increasingly difficult as we become more competent at the specification, design and implementation phases. The distinction between the two areas was dubbed by software people in the USA as "building the software right" and "building the right software". With our increasing competence in the later stages, larger and larger projects become undertaken and the cost of failure of the delivered system to meet requirements has risen dramatically and notoriously. Requirements Engineering has become of paramount importance and is still a less understood area. Procurers of large systems, especially in government, seem reluctant to learn that incremental developments are a way to avoid these massively expensive disasters, but still they try to seek "big bang" solutions. Suppliers too should refuse to enter such contracts, but offer incremental solutions with usability checkpoints and frequent deliveries.

I gave papers at two conferences in 1988, one on Achieving Software Quality held at the Wembley conference centre and organised by Blenheim Online, and the other on Prototyping held at the European Commission as part of the PCTE (Portable Common Tools Environment) project. The first paper[33] tried to explain how formal methods and proof were a necessary ingredient of top quality, but not a sufficient one. Validation activities are needed at every stage to check the what is being produced meets requirements. In a context of formal methods, validation is a form of prototyping, in that a skeletal or abstract form of the final product is shown to the customer for checking against requirements: "Is this what you really want?"

Advances in mathematics have been driven by applications of the subject over the ages. In the seventeenth and eighteenth centuries the needs of navigation inspired much of the advances in the mathematics of the time, and in much earlier centuries the Greek civilisation's urge to build monumental religious and civic buildings inspired the geometry of Euclid. FACS had several members in common with the IMA, Institute of Mathematics and its Applications, and at one informal meeting a few of us mused whether computing or computer science was having an influence on the development of mathematics, right at the

[33]See Denvir 1988

present time. For example, in mathematical logic, proofs have always been meticulous, step by step constructions. When we have proofs of correctness of computer programs, a proof starts to have much in common with the proofs in mathematical logic; that is, they are more rigorous than traditional proofs in "ordinary" mathematics, where one might have phrases like "without loss f generality" and "extending the notation in a natural way", without going all the way to the formality of mathematical logic. However, as soon as we start to try to construct computer-supported provers and proof checkers, the proofs have to be totally meticulous and formal, and they can become exceedingly long, thousands or even millions of lines. Mathematics has no tradition for handling or manipulating proofs of this size.

Another example is the use of category theory to provide a model of some of the more arcane features of programming languages, such as polymorphic and recursive data types. Indeed, the types and functions used in computing are similar to, but in their fundamentals, different from types and functions in mathematics. Everything in computing is computable and countable. The so-called real types in programming languages are in fact computable numbers, because they can only be generated by computable processes and functions.

We decided to have a joint conference, organised jointly by FACS and the IMA, to explore the idea that computing has inspired a revolution in mathematics. It was and is a debatable assertion and the title of the conference was deliberately provocative. The result was a conference in 1989 at which mathematicians and computer scientists presented twenty-three papers[34].

Apart from work on the regular contracts for Praxis and its technical infrastructural support, such as inter-project reviewing, I found myself being asked to chair sessions at conferences and do other "professional" work: Software Engineering '88, Discrete Structures for Software Engineering, Modula 2. In general employers were willing for their staff to spend some time on activities of these kinds, in the hope that it would improve the company reputation in the longer term. Certainly, the general exposure is helpful, and if nothing else will encourage staff recruitment.

Praxis spent some time in internal meetings discussing its own strategy, future and business policies. This was reminiscent of the software research group in STL and of RADICS, only the discussions in Praxis were more "commercial". We reviewed the monthly revenue against targets, sales orders to date, sales prospects, staff numbers and whether and how much to grow. Some managers considered a target of 30% per year was desirable, arguing that greater numbers enabled "big, exciting projects, opportunities for staff growth, and geographical spread into Europe". A growth plan of 30% annually would bring us to 500 plus in five years time (it did not happen). We discussed types of projects (developments, research), and how to

[34]See Johnson and Loomes, 1991.

get them funded. Having identified a desirable project, the construction of a formal methods-based systems analysis method, say, we could approach potential funders, such as CCTA. At the time our technical policy was in effect determined by the sales group and the opportunities they sought. This was not, we thought, the best way to set the technical direction of a group. We discussed the rôle of consultants, of which I was one. Our understanding of business sectors, the company, technologies and markets should lead to obtaining further business. We should consider expanding into the government sector, telecoms, transport, critical systems and medical electronics. For these we would need new skills in telecoms, secure databases, encryption etc.

John McDermid, Professor of Software Engineering at the University of York, asked me if I would contribute a chapter on discrete mathematics, the subject of my own book and of the first FM course I constructed, to a substantial volume he was editing, the Software Engineer's Reference Book[35]. I agreed, and when I sent him the first draft, he replied that part of my text would fit better into the Introduction, which he was writing. So John and I co-authored the introduction and I wrote the discrete maths chapter, one of sixty three contributed chapters. The Software Engineer's Reference Book, now perhaps a little dated from a technical point of view, was a thoroughly informative and useful volume. I felt that if one read a chapter each evening one could get an excellent understanding of software engineering in less than three months. I found several of the other chapters most instructive myself.

One of the more interesting opportunities we explored at Praxis was with the EFTPOS consortium. EFTPOS is Electronic Funds Transfer at Point Of Sale, the system now familiar to us all, where one can pay for goods at a shop or supermarket with a credit or debit card. To start it all off a UK consortium was formed consisting of four banks and the national debit card company. They were to begin with a pilot study. Hardware for the trial was to be supplied by IBM, terminals by Ericsson and Ormeron, a Japanese firm with an outlet in Chessington, and IMI. Cryptography was to be supplied by Plessey, their Base 24 standard networking product. Praxis was never involved, but the ubiquitous presence of these systems today shows that the trials and studies all took place successfully back in 1989.

Lucas developed and supplied the control systems, including the control software, for Rolls Royce aero engines that powered many civil passenger aircraft. The target embedded microprocessor was the Motorola 68000. The software was developed on a micro-VAX development environment. At that time (1989) Lucas were using a local language, LUCOL, for the code development phase and a method called Auto-G for the design. Auto-G was

35

based on Yourdon and data flow diagrams. Lucas were under contract from Rolls Royce to build the control software for the next generation engines to be produced in the late 1990s, with new control systems using the Z8002 as target machines. Lucas had promised to produce a new controller within the next eighteen months, delivering the first prototype within three to four months. They were expecting to write about 10,000 lines of code. LUCOL was an autocode-level language, but I recognised that it was designed in the tradition of analogue computers, a thirty-five year old technology. They were in effect using digital computing techniques to simulate analogue computing, even electronic servo systems. The code structure was linear with modules being repetitively executed every so many microseconds. The modules were equivalent to macros in a more conventional language. Within each module there were determinate loops. About 40-90 modules had to be produced. Previous versions had to be rewritten for the Z8002. Rolls Royce had commissioned Lucas to do a static analysis of the code for the 68000 using Spade. Both parties proposed to do the same for the modules designed for the Z8002. I could see that translating the LUCOL code into FDL would be feasible, which would enable using the more powerful SPARK proof system.

All in all, an interesting project, and encouraging to some degree, that new techniques were being explored to increase the confidence in the safety of this kind of critical software. Although some of the methodology in the project's technology was a little retrograde (analogue computing traditions, and little separation of concerns in the design), the team were now taking a considerable leap forward. A concern I had was that, with different organisations being responsible for the controllers, the engines and the plane, Lucas, Rolls Royce and the aircraft builder, no-one seemed to have responsibility for the overall cohesion of the architecture of the control system from the pilot to the engine. This continues to be a feature of today's urge to unbundle responsibilities, for example on the railways, with different firms taking charge of the rail network, the stations, the trains and their schedules: who takes care of the coherence of passenger safety from entering the station, alighting from and to the platform, having enough time to board and disembark from the train safely, and safety throughout the whole journey?

A short time afterwards we spent time with Marconi considering another consultancy opportunity.

Those of us who gave courses also attended them, both technical ones and "management" courses. I personally attended management courses on appraisal interviewing, leadership and recruitment interviewing. As well as the many FACS events that I went to in my rôle as a FACS committee member, I attended courses on Type Theory, Theorem Proving and Frame Theory. Type Theory is the mathematical theory which can form a model of the data types

that one can use in programming languages. This is especially tricky where the language allows the programmer to define their own types. Theorem proving and logic is one of the most abstruse topics in mathematics, left only until part III of the Cambridge mathematics tripos when I took it in 1962, and hence a post-graduate course. To design and even use an automatic theorem prover or proof assistant, which is an aid to proving programs correct, one needs to know about mathematical logic, which underpins formal proofs. The course was held at Leeds University.

The Ministry of Defence standard DS0055 is related to BS5750, to which Praxis had recently been certified. With our propagation of the use of formal methods, we investigated how these standards should be applied to projects using FMs. There are in DS0055 a number of rôles, organisational structures, records, notions, documents and life-cycle activities. It was possible to relate most of these features to the activities specific to a safety critical project, in which formal methods were most likely to be used.

In the context of safety critical systems, whether or not they have an IT component, a technique called “Hazard Analysis” was developed, became a mature, well defined approach and the subject of standards itself. It would cover an initial review of the contract for the project under scrutiny, its technical validity, a review of system safety, operating and support analysis, review of system safety during maintenance, and more.

The IEE was becoming more interested in computing, seeing it as a branch of electrical engineering. It began to rival the BCS as the professional society that supports software engineers, and does so to this day. In 1989 they got together with the NCC and awarded certificates in software engineering. Ten polytechnics and higher education colleges offered courses that would lead to accreditation. Today ET, the fortnightly Engineering and Technology magazine of the IET, and the frequent regional meetings of the IET, are often dominated by articles and lectures on information technology matters.

Maybe there is a parallel between this movement in the IEE/IET and in the software engineering industry. One could see it as a movement away from the coalface of hard engineering work, the technology of materials and their manipulation, to the more arm’s length control, where software prevails. At about this time I went to the Praxis library and looked up the archives of the twenty most recent projects. (The practice of keeping an archive of completed projects can lead to many useful insights afterwards and is much to be commended). Only five out of the twenty projects were required to deliver an implemented program. It seems that customers needed most help in the early stages of the development cycle. Three delivered project plans, two requirements analysis, four produced designs, five were more general consultancy projects and two were software or system audits. Having said

that, the implementation projects were generally among the longest in terms of effort and duration. I personally came across customers who wanted to retain control over the implementation of their projects, and they perceived that they could do so most easily by "doing it themselves", after help with the tricky early stages.

The BSI standardisation of VDM continued to progress. ISO voting was positive, STC copyrights of certain key documents were expected to be released, a contribution on good style of writing VDM was included, and the committee considered issues relating to polymorphic and recursive types. The process of defining the standard uncovered quite a few areas where different uses and meanings had been assumed by different VDM writers. We decided that the standard VDM should not permit any language extensions and only admit changes necessitated by problems in the semantic definition. At the same time, funding was needed for the whole process of standardisation, and we pursued funding opportunities from the DTI. The DTI had an initiative called TickIT for certifying software quality. The DTI used BSI as a certification agency for TickIT, and so funding for IT standards could be obtained through this route. Some eight intensive meetings over twenty months later and the standard went to the BSI and ISO in "draft" status.

FACS held the Third Refinement Workshop in Oxford in January 1990. The topics included the refinement calculus, originated by Carrol Morgan, refinement applied to CCS, CSP, Action systems, the proof of a compiler, ML, RAISE, and VDM. I continued to be heavily involved in BCS FACS. The committee meetings generally took place in the evenings, or occasionally over a weekend. So on the whole this did not impact my work at Praxis very much, although responding to emails, organising further meetings and the like took up a little time during "normal" working hours. This time had to be recorded as an overhead, and I found it quite difficult to keep this to an acceptable minimum. Staying late at work was frequently necessary. FACS got close to the London Mathematical Society, as well as the Institute for Mathematics and its Applications, the IMA. We held events on functional programming, concurrency, LOTOS and the formal definition of specification languages. We decided to include a tutorial event once a year on a subject such as denotational semantics. During those years, 1990 – 1991, we often held committee meetings in a booked room in the City Pipe, a Wetherspoon pub in the City of London. There was no charge for the room, the proprietors reckoning that hosting a meeting on their premises would stimulate sales. They were probably right! Various industrial and other organisations supported or sponsored some of the FACS events, such as BT for LOTOS, IBM for the Third Refinement Workshop and the Logic for IT initiative of the SERC for the Concurrency workshop. The FAC Journal was in its second year of publication (1990) and a line-up of future issues in the pipeline. I edited a special edition on the modularisation of specifications. Almost all formal specification languages were "flat", that is, had few facilities for expressing a specification of a system in

terms of the specifications of its parts, unlike most modern programming languages. This was a deficiency. So we invited papers on the subject for FACJ.

There was a tradition that had grown up in FACS to hold an event over a day or two in late December, which we called the Christmas meeting. On 20th December 1990 the topic of the Christmas meeting was formal aspects of databases. Databases were normally associated with commercial applications, where formal methods were less in evidence. Although this meeting went off well enough, I felt that it was used by some of the database devotees as an opportunistic platform for advocating database techniques rather than formal ones. We held a weekend committee meeting in January 1991 and did a lot of forward planning, plotting out events for the next two years into 1993, with proposed events on B, Measurement, Databases again, RAISE, Domain theory, and Process Algebras. We drew up guidelines for organising events and established a working arrangement with Springer in London to publish the proceedings of the more significant events in a dedicated workshop series. One of these was on the topic of formal methods applied to measurement. There had always been a tension between formal approaches to software development, where a solution is derived progressively by logical steps from an initial specification, and a by then burgeoning school of thought, software metrics. The latter regarded the software development process as something subject to experimentation, measurement and theorisation about resources and error rates. On the face of it, these two schools are in opposition: formal development asserts that the use of proof must eliminate errors, whereas metrics asserts that errors will always happen, so the way to control them is by experimental observation of trends and thence prediction of reliability. Robin Whitty, professor at South Bank University, London, and I believed that these two positions were too stark and there had to be common ground. We started to plan a FACS event on Formal Methods applied to Measurement. The resulting event in May 1991 at South Bank University was particularly successful, avoiding the traditional clichés of the opposing positions, and with several speakers (Austin Melton and Horst Zuse in particular) putting forward thought-provoking ideas on abstract models and measurement theory.

Meanwhile we were planning the Fifth Refinement Workshop, to take place in London in June 1992. Lloyd's Register, Program Verification Ltd. and the DTI had agreed to sponsor the event. The Refinement Workshops had by now become the major annual British formal methods event.

Towards the end of 1991 the BCS came into some temporary financial problems, owing to its difficulty in selling its London headquarters in Mansfield Street. FACS was in a very good financial state, having its own bank account and facilities, but constitutionally we were part of BCS and so there was a possible danger to our own financial autonomy. We seriously considered setting up a company owned by the committee and at the committee's risk. In fact, this never happened. BCS recovered extremely well largely by mounting a very

effective recruiting campaign, holding out benefits of status, qualifications and facilities for new members. The Mansfield Street headquarters were sold, BCS moved to Swindon where costs were lower, but in due course moved back to London to substantially superior premises in Southampton Street. BCS is in 2011 a financially and scientifically thriving professional society.

Of course, I was working for Praxis all this while, and supporting FACS and the standardisation of VDM had to be done mostly in my spare time. It was acceptable to use some of the firm's resources, secretarial and communications, for these "professional" purposes, since the involvement enhanced the company's technical reputation and image. One Praxis project I carried out was for a London finance company client. Their business was with real estate, mortgaging, insurance, valuations and the like. The project was named "Abacus". I was chosen to do this project because I lived in London; Praxis was based in Bath, an hour and a half's train ride away. Anthony Hall at Praxis wrote the bid for the work. Anthony was experienced with Z, the formal method developed at Oxford University and used in projects at IBM and INMOS. In his bid he proposed using Z in the Abacus project. I had not used Z before, although I was familiar with it; I had concentrated on VDM, which had broadly the same characteristics. So I had to get up to speed rapidly with Z, which I did with little difficulty. In the project I used a simple support tool for Z written by Mike Spivey at Oxford University, called *f*UZZ. The firm wanted to produce a database which reflected the structures of their clients, properties and users, with their rules built in to the database. The firm were extremely concerned with security and commercial confidentiality, and wanted a secure database for these reasons. They were adamant that their name should not be recorded in any of our documents, for fear of their competitors learning of their use of formal methods in the project. Although I spent some weeks on the customer's premises, constructing the specification in Z using *f*UZZ, I'm to this date not sure what the firm's name was.

The usual process with any project of this kind is to start with eliciting the customer's requirements. They had a large rule book, which was full of a mixture of technical and other requirements. By making experimental formal fragments of aspects of these rules in Z, I unveiled numerous uncertainties and produced lists of questions for them. My main contact within the firm was two software engineers, who were the two most IT technically oriented people in the company. We had a string of meetings where I asked questions. Do the users form a hierarchy? Is it the case that no property can belong to more than one property group? And so on. Within a few weeks I was writing copious quantities of Z, checking its consistency with the *f*UZZ tool, and giving small presentations to my contacts in the firm. The project was delivered after six weeks, with apparent satisfaction on the part of the customer.

My manager at Praxis, John Thornton, spoke to me about a prospect, some consultancy work for a firm called Headland. They had bought out four other companies and wanted to unify the five different accountancy packages. All were product -oriented. The R & D functions of three of the companies had been brought together under one roof near Basingstoke. Research was separate from development, but all were innovative enterprises. The packages in question were wide-ranging, covering variously Planning, QA, Methods, Configuration Management and Documentation. The customer offered a fifteen day contract to analyse what was required, leading to a feasibility study, an action and cost plan. They wanted a 10-15 page report with a plan for further work, including networks and bar charts etc. The company had 480 employees. I could do the work from home. But it was another prospect that never matured into a contract. Once again, I suspect that the company issued an invitation to tender to a number of consultancy companies like Praxis, surveyed the resulting bids and used them as the starting point for the feasibility study, which they then carried out in-house. That way they got several considered top-level analyses for free.

John Thornton, who was in charge of the Consultancy group in Praxis, searched for possible contracts that I could do. I still lived in London and still worked under an arrangement where I was considered to be based at home, charging my fares to Praxis headquarters in Bath and taking any London based contracts I could. John had been negotiating with the DTI for me to be seconded there, in the Information Technology directorate, for a lengthy period. I went to the DTI in London for an interview with the directorate's head, Professor John Buxton, with whom I had had several earlier associations. John Buxton was also on secondment; indeed, it was a policy of the DTI to bring in specialist talent from academe and industry to help operate their more technical directorates. I would be given a job title of Assistant Director of the IT Directorate, along with about four others. The Directorate used to initiate programmes of work, such as software engineering or speech and language technologies, offer grants to proposed projects, usually carried out by industry, but frequently in mandatory collaboration with academe, judge the contending grant applications, and fund the most promising of them. I had already been in the rôle of Monitoring Officer for a couple of projects, notably Analyst Assist, on contract with the DTI. Within the DTI, I would be Project Officer for a list of projects, in a number of areas. One of these would be Speech and Language technology, taking over from Nicolas Ostler, who was expected to leave in a few months. Projects in this area related to the analysis and understanding of speech, and translation and processing of natural languages. I would also be Project Officer for European and ESPRIT projects. There were several other European initiatives besides ESPRIT, some of which worked under the arrangement that a collaborative European project would be funded by the national government of each partner, taking responsibility for that partner's share of the expenditure. So in a collaborative project with industrial partners from France, Spain and the UK, for

example, the UK firm would be funded by the DTI, the French and Spanish companies would be funded by the French and Spanish equivalents of the DTI, and the European Commission would oversee the whole project in some way.

John Buxton advised me that at the end of my secondment I was likely to find that the "wound" that I left by my absence from my employer would have healed by the time I returned, and that many secondees found themselves out of a job when their secondment terminated. This did not entirely surprise me.

The SERC was running an initiative called Logic for IT, and under that was holding a course on Frame Theory. Frame Theory is a topic in mathematics related to lattices and topology, and a generalisation of the latter: that is, any topology is a frame. Frame theory and topology can be used to model computations. Having been engrossed in the rather mundane, less academic work of Praxis, I felt a need to stretch my brain a bit on some demanding intellectual computer science. Shortly before the start of my secondment to the DTI I asked to be released to go on the course, which was three days long and held in Oxford. Praxis agreed to let me have the time, recorded as training, but said I had to pay the course fee myself. There was one price for academics and a higher price for industrial participants. I asked the course administrators if I could be granted the academic fee, since I was paying it out of my own pocket. They silently agreed. I was the only industrial attendee and I suspect that I was the only one paying myself. The course, given by Harold Simmons of Aberdeen University, was demanding, enjoyable and stimulating.

Chapter 11 Civic Duties

I started my secondment at the DTI in April 1990. Nicolas Ostler was, indeed is, a supreme expert in languages and the technology for processing languages. It was unwise for the DTI to end his assignment as a secondee, but this move arose from their policy and outlook that no-one needs to be a specialist; generalists were the order of the day. I felt embarrassed at stepping into Nick's shoes when I was so clearly less qualified to do this work, but he explained the ins and outs of the tasks to me with the utmost care, to make my learning curve as easy as possible. In the event, he stayed on at one day a week for the next six months, which was immensely useful.

I was made extremely welcome on my arrival at the DTI. Their care for employees was without parallel in all the different types of organisation where I had worked. However, working there was stressful, on account of the enormous importance attached to the results and timeliness of the work one had to do. A Project Officer's duties related to past projects that were at completion, projects currently running, and new programmes of work with applications for grants from prospective projects.

One requirement for those working in the DTI was that one should not have any interest in any particular company. This included owning stocks and shares. I had inherited a few shares from my mother, and had to declare these. It was decided that the quantity was so small that there was no issue.

Working for the civil service, one has grades. So-called Management grades are numbered 1 to 7, 1 being the highest. This was in contrast to Praxis grades, where the higher the number, the higher the grade. At first, the DTI proposed that I should be grade 7, but John Thornton negotiated that I should be grade 6. This would bring more money into Praxis from the DTI for the secondment. So I was a grade 6; comparing myself with the other grade 6 staff, not many of them, I felt that this was perhaps higher than I deserved. The grade was also one's job title; instead of saying "my department manager" or "my division manager", people would talk about "my Grade 5" etc. I was one of three grade sixes in the directorate. Nicolas Ostler and Graham Mackenzie-Washington were the other two. Graham was a regular DTI employee, and something of a guru of the directorate's ropes. He was a tremendous support in guiding me through the mores of DTI's ways of working.

Every DTI project, whether it was a large scale initiative, support for a project under an initiative, or even engaging a secondee such as myself, had to have a ROAME reviewed and accepted. ROAME was an acronym for a case for the required funding, consisting of Rationale, Objectives, Appraisal, Monitoring and Exploitation. Nick Ostler was in the process of writing a ROAME for speech and language technology projects. The plan was for me to take over this work once he had got the ROAME through the lengthy approval process. A ROAME of this kind had to go before Government Ministers, but not the Treasury (some did, however). I was to concentrate on formal methods projects and their European connections. Besides ESPRIT, there was a separate European initiative called Eureka. Unlike ESPRIT, Eureka was not funded through the CEC (Commission of European Communities), and a Eureka project could include partners from any of 39 specific European countries, not just those from the EC, although the 39 included those.

SALT was an association of organisations active and interested in speech and language technology, the UK members funded by the DTI and the SERC. It too needed an approved ROAME to secure the DTI part of the funding. It funded some projects, and encouraged exchange of ideas and information between active parties through meetings and conferences. A considerable number of the ITD were involved in SALT.

There was a plethora of committees and subcommittees within or including the ITD. ITAB, the Information Technology Advisory Board, oversaw funded IT projects. It had two subcommittees, A and B. I found myself most often attending meetings of subcommittee B. An extract from some private notes I took at a meeting of subcommittee B where a certain funded project was being discussed, may give a flavour of its way of working.

> There seem to be several factors contributing to the lack of success of this project. None of them are disastrous or overwhelming in themselves; none of them are such that some party is obviously to blame; all of them are easy to recognise in retrospect, but easy to have been passed over at the time; none of them alone would have definitely jeopardised the project to the extent that the they would have been adequate reason for suspending the project.
>
> One major factor I think is that there was not a sufficient coherent unified technical vision for the project as a whole that could lead to plans for technical integration of the work. Hence the disjointed results, and the collection of deliverables which are difficult to conceive as a united whole. But I want to look at other parts of the file to get a better idea of the initial planning.
>
> ...

There was sometimes a temptation for Monitoring Officers to identify too strongly with a project, especially if they were subcontracted from outside the DTI, so that they began to "defend" the project against the DTI instead of acting as an objective observer, reporting on the project to inform the DTI. Seasoned DTI staff referred to this phenomenon as "going native", recalling an imperial past!

Two important programmes in the ITD were Systems Design and Safety Critical Systems. Both were overseen by Subcommittee B, but the projects in each had their own project officers. I was project officer for Systems Design projects, Simon Attwood was PO for Safety Critical Systems projects with Bob Malcolm, another secondee whom I had known for many years, technical coordinator. The immense level of scrutiny of these funded projects was a surprise to me; all of them had Monitoring Officers too, a rôle I had carried out for the Analyst Assist project. In addition, there were some more ad hoc groupings of projects. Metrics for example was not a programme, but consisted of a few pieces of work and individuals, possibly in different programmes. Great emphasis was put on attempting to coordinate and cross fertilise between different projects, to the desirable advantage and advancement of UK industrial firms. In some ways, one could say that inter-company rivalry within the UK was discouraged in order to promote international competitiveness.

Eclipse was a knowledge engineering collaborative project, which has now grown to a strong technological community. A further project was set up to evaluate the Eclipse project. This project in turn had to have a ROAME, and I was asked to provide a view to contribute to the evaluation of this project. I found it a bit odd evaluating an evaluation project, especially with respect to the aspects of its ROAME: how could one evaluate its exploitation, for example? I thought that the only way an evaluation project could be expected to be exploited was by informing the original project under study, Eclipse in this case, and observe whether such information had been taken on board and whether Eclipse in turn had been exploited, a second order exploitation if you will. All these were indeed positive outcomes.

All these activities came under the umbrella of the Advanced Technology Programme in the DTI. This covered more than just the activities within the ITD, thus more than just IT related

work. I was sent on a two-day course on the ATP. We were told about the general aims, Support For Innovation (SFI), the Small firms Merit Awards for Research and Technology (SMART); acronyms abounded in the DTI and often seasoned DTI staff had forgotten what were the original expansions of many of them. Government support had moved away from being near-market in recent years. The thinking was that government should not take on industry's initiative. On the other hand, the DTI at any rate was not going to support blue-sky research (however, the EPSRC and parts of the EC would do just that). EC rules prevented governments from giving grants that would make for unfair competition, hence funded projects had to be some distance from "market". Much of the course was devoted to explaining terminology. A Scheme is a broad heading of government funding. Programmes were technical areas within a Scheme. Projects could be funded within a programme. (In the CEC context, programmes were grouped into a Framework. ESPRIT was one of 37 programmes forming, in 1990, Framework III.) There were various criteria which applied to both programmes and projects. Additionality was one such: research and development had to provide some additional advantage to a wider audience than just the participants.

The ROAME approach was explained, including the desirable form of a ROAME, down to the recommended number of pages for each section. The Rationale should explain why the project/programme was to be carried out, in terms of the benefit for the UK. Policy, Rationale and Objectives form a hierarchy of abstraction, each being a reification of the former. Objectives could be commercial, economic, operational, technical or relating to dissemination. Much store was set by Market. Market was supposed to provide the ultimate value of any endeavour. But if through some explained quirk of mechanism, market failed, then that market failure could be used as a reason for government funding. Personally, I was never convinced by this emphasis on the mechanism of market. It seemed to me to smack of bias towards a particular kind of political-economic theory.

Eureka covered 39 European countries including but extending beyond the European Community. There were at the time 297 current projects under Eureka, with another two to three hundred further proposals in the pipeline. Each country had a national coordination office. In the UK this was a section within the DTI. Criteria for funding under Eureka were much the same as in other schemes, but a collaboration could involve just one UK partner, projects could be nearer to market and industry-led, they need not involve government funding (about 30 were unfunded but were still Eureka projects), funding was a maximum of 50% of eligible costs, projects could co-opt more companies after starting up, and the IPR could be negotiated between the participants, with the respective national coordinating offices putting in their bids for national interests.

Some programmes could incorporate "Uncle" projects. These were academic projects that had an industrial "Uncle", an individual from industry who would visit the project at intervals, typically every three months, to provide industrial input and try to keep the project

of ultimate practical utility, even if that was long-term. This was a very light form of academic-industrial collaboration, but it gave the academic partner some ratification for government funding. Even these projects required a monitoring officer. For Uncle projects, these MOs were from another academic institution, and were paid a standard rate of £200 per day, working out at about £25 per hour in 1990, modest even then.

The DTI was involved in PCTE and PCTE+, already mentioned, since Praxis took part in several PCTE-related projects. The MoD and ECMA were highly interested in PCTE+, and ECMA proposed to adopt it as a standard: ECMA PCTE would be PCTE+ re-badged. The DTI proposed to host a workshop on the industrial use of PCTE, publicised within JFIT and the BCS Computer Bulletin (a newsletter of the BCS, which is now renamed as IT Now). A PCTE newsletter was being published by the French R&D firm Emeraude.

The European Software Factory, ESF, was an international endeavour to define and produce an environment for developing and supporting software: a PSE in other words, but wider-ranging than most. BT were participants in the project, assisted with funding from their internal customers. BT saw ESF as near market, and were not interested in its commercial exploitation as such. Their interest was to encourage their software suppliers such as ICL and SEMA to use ESF and thereby ensure a uniformity of quality and direction of their supplied products. Having BSI registration, BT was motivated to follow a quality route in its policies. They considered that DTI funding would help to sell the concept and wanted to know whether such DTI support was forthcoming, to what percentage and scale, the appropriate details. A possible spin-off downstream of ESF itself might be common components and general experience. STC, ICL and Logica were among the other UK participants, and SEMA in France. A Council drawn from the participating members steered the work and direction of ESF. One technical concept within ESF was a Software Bus. (This nomenclature suggested an analogy with a hardware bus, which is a general term for an information highway carrying data between a large variety of components). The Software Bus was a medium by which the ESF process model was integrated with the processes in ESF, for example Reuse, Project Management, etc. It was sometimes described as a discipline of integration. There was a degree of compatibility between ESF and PCTE, and parts of PCTE were reused in ESF.

There was some slight tension between the need for the ESF consortium to align its partners to the project's objectives and a temptation for an individual partner to use the project merely to further their proprietary developments. But mechanisms were in place to control these difficulties, all reinforced by contract.

Three programmes within the DTI were approved or in progress: Safety Critical Systems, Knowledge based systems and Speech and Language technologies, whose ROAME was

being written. A future programme was to gather ideas from these three existing ones and carry them forward, with some developments of emphasis. This new programme was to be called Systems Design and Productivity. I was to be the "link man" to write its ROAME and run it. I would have to collect ideas from interested parties and formulate viable arguments for its approval. The programme would have to be worth doing, of benefit to the UK; the arguments would have to be primarily economic. For example, if one could speed up the route from requirement to delivery of systems, this would lead to competitive ability. Computer supported cooperative work could lead to large, complex long-lived systems built by groups of people. One might promote compatibility of tools across different vendors. Government funding could overcome sectoral specialisation especially across diverse user populations, leading to reuse across sectors. Productivity thus became the primary argument: productivity can lead to interdisciplinary working and better quality. The concept of reuse needed to be interpreted widely, as reuse of designs and concepts for example, not just of software, which was in any case rare. Sectors were of various kinds, products, markets, technologies and others.

I attended a seminar for Project Officers, to which Monitoring Officers were also invited. Project Officers worked within the DTI and had oversight of a group of projects within one or more programmes. Each project had a Monitoring Officer engaged from outside the DTI on contract, who reported in principle to the Project Officer. Having been a Monitoring Officer, and now a Project Officer, I had some suggestions for how the activity could be improved. Monitoring Officers could be given feedback on whether their reports were at a useful level of detail. More knowledge of the structure of staff and their rôles in ITD would help, as would an understanding of what was done with their reports; currently, Monitoring Officers received communications from half a dozen different people within the ITD, which I had found perplexing. One reason for this was that the DTI was excellent at using junior staff to the limit of their abilities, in taking minutes of meetings, chasing up due reports from Monitoring Officers and deliverables from projects, and all manner of clerical activities, which nonetheless required a measure of intellectual ability. A result of this is that people who had a relationship with the DTI, managers of funded projects, Monitoring Officers and others, would receive communications from many different DTI staff, and this could be confusing. An initial one or half day seminar for monitoring officers would have been a help. As a monitoring officer I never received the Project Monitoring manual, or knew of its existence. Being kept informed of changes of Project Officer would have been helpful! The monitoring officer needs a copy of the project proposal, so as to know what deliverables, activities and objectives were expected of the project, preferably before signing the contract. That way a prospective MO could make a better judgement about whether he/she was capable of monitoring that project. I had to ask my project for a copy. I would have liked a clearer understanding of what, as an MO, I could demand of the project: e.g., to see a list of what

personnel they are charging to the project month by month and the specific contributions of each one. I felt that monitoring officers should be advised on how proactive/reactive they could be – I thought they should be encouraged to be as proactive as their skill allowed. On the Project Monitoring Manual, under "Rôle of the MO", nothing was said about whether the MO should try to influence the project to be successful! I thought that this was both desirable and feasible. As an MO I required to see copies of all working documents that were communicated between the partners, so that I could keep track of the technical progress etc. I suggested that this should be a standard "right" of Monitoring Officers. There were various other discrepancies between the duties that the MO had to carry out, according to the Project Monitoring Manual, and the information provided enabling him/her to do so.

All these criticisms of mine might suggest that the process was a bit of a shambles, but in practice an intelligent MO could deliver meaningful and helpful reports without too much difficulty. But as a PO I found that some Monitoring Officers were surprisingly lacking in initiative.

The DTI kept a watching brief over all the European Framework projects having UK partners, and the ones funded under ESPRIT would feature in the annual JFIT conference. A wide view would generally be taken of ESPRIT projects: their context within other Framework projects, their position in the context of ITD, and the general shape of funded initiatives in Europe. There were, in 1990, 19 UK leaders of ESPRIT projects, and 24 projects having at least one UK partner. These ESPRIT projects were within Frameworks I and II; Framework III had been approved but was at the time yet to start.

I was taking over the PO duties of projects for which Nick Ostler currently had responsibility, as well as other items. There were three "Clubs" within Nick's domain. These were largely unfunded associations of organisations having a specific common interest. SALT was one. The others were Logic Programming, and Advanced Databases and Knowledge Bases. Also, there was Eurotra.

Eurotra was a programme comprising a collection of projects researching the machine translation of natural languages. Eurotra had started in 1982. There was a steering committee and I would be taking over from Nick as UK representative on this. Eurotra was financed by the CEC and member states. The project originated from a perceived need of the CEC: there would be economic savings for the CEC if the numerous translations of documents into the languages of all the member states could be done automatically. This, in 1990, was seen as a forlorn hope, but the project was a medium for considerable and wide-ranging research. This led to the CEC letting subcontracts to member state organisations to do the work. The UK participants were UMIST and Essex University. Within the DTI the funding source for Eurotra seemed always to be passed around from one budget to another.

In July 1990 I attended an interesting seminar on the Japanese software industry. The most notable contributor was perhaps Alan Benjamin, who was the first director general of the Computing Services Association and founder of the Worshipful Company of Information Technologists. He noted that long term high investment and patient marketing were characteristic of the Japanese software industry. He recommended using the British embassy to find a partner in order to gain entry to the Japanese market. No "NIH" (Not Invented Here) prejudice operated in Japan. Alan Benjamin recommended that an Anglo-Japanese industrial club be formed.

My own personal observations from the range of published Japanese research papers were that there was a stronger link between research directions and long-term industrial aspirations in Japan than in western countries. Research focussed on very large scale parallelism and sophisticated (mathematical) logics. From this I predicted that one long term industrial aim was the development of very clever human-machine interfaces, possibly using natural language and spoken words. If this was the case, we have yet to see it emerge, but some work along the way might be discerned.

Although I was seconded to the DTI, I still spent occasional days at Praxis in Bath, reporting briefly on my secondment, which was viewed as a Praxis contract, and taking part in some of the indirect Praxis activities. I remember, for example, participating in a committee to review the Praxis project review process, which was crucial to the in-house quality system.

There were many specific duties and fierce deadlines working in the DTI, but one had a large measure of freedom too. The approach to timekeeping was relaxed, provided the work got done, and one was free to arrange a visit to an industrial or academic institution on one's own say-so, for example. I was interested in the use of IT to assist those who were disabled in some way, blind people for instance, and was a member of the Disabled SIG of the BCS. I visited Sight and Sound Technologies, a firm that designed and manufactured equipment enabling blind people to use a computer, among other things. Being a representative of the DTI, and hence the "Government", one had considerable clout: a visit was always granted, and demonstrations arranged and provided. Sight and Sound Technologies showed me their various pieces of equipment and told how they were being used in libraries and other places. Today, such "accessible" interfaces are provided as part of the latest operating systems as built-in facilities.

Springer-Verlag London publishes a lot of computer science and software engineering books, and FACS had developed something of a special relationship with them. Springer published the FAC Journal and several proceedings of FACS conferences. After some negotiation, in 1990 Springer agreed to start a special series of volumes, called FACIT – Formal Approaches to Computing and Information Technology. An advisory board was set up and we had lengthy discussions on the content and emphasis of the series. Springer expected members of the board to procure proposals for books from our contacts, something that I suspect did not transpire as much as they would have wished. However, a healthy collection of volumes appeared in the series over the years, and it was the first port of call for many computer science authors looking for a publisher.

In July 1990 the DTI held a seminar on ESPRIT Framework III. I introduced the day: the purpose of the seminar, format of the day, circulated the attendance list and introduced the first speaker, Derek Flynn. A number of speakers gave position papers and there was a fairly free-ranging discussion. ESPRIT Framework III was to run from 1990 to 1994, and had a total budget of 5,000 MECU, millions of ECU. Participation by UK organisations was allowed up to 16% of this total. Speakers from academic and industrial organisations in about equal numbers contributed positions: IPSEs, software reuse, safety critical systems, technology transfer, quality, user interfaces, foundations of software engineering, software components, measurement and metrics, reliability, testing and validation, performance, exploitation, distributed systems, neural networks, learning systems, genericity, formal methods, speech and language, and multi-media were all topics of discussion.

I had been Monitoring Officer for the Analyst Assist project, of which Robin Pyburn was the project leader. Robin and I had both worked at RADICS in 1969-1970. Now Robin Pyburn was the Monitoring Officer for the project CLARE, and I was the Project Officer. It is mildly amusing how the same people waltz around each other over the years.

John Buxton and I presented the ROAME for the proposed System Productivity programme to Subcommittee B. We received several pieces of advice: to emphasise the problem of scale, i.e. how to build large systems; to strengthen trans-sector reuse; building systems in a generic environment; to show how the objectives lead to revitalisation of British software industry. The discussions at these meetings were always in very general terms with much metaphorical hand-waving. The programme had originally been called System Design, but the name had changed to Productivity to change its emphasis from the technical to the economic. The estimated budget was some £36M with half provided by industry and half provided by Government, of which DTI would supply £14M and SERC £4M.

John appointed me as Project Officer for the applications of Imperial College and BT to take part in ESF (European Software Factory). ESF was a project under the Eureka scheme. In

many ways, ESF was more like a programme rather than a project, because of the loose associations between the work of the different partners. Another member of ITD retained the PO responsibility for ICL's participation. John Buxton and I visited BT, who already had experience in project support environments: MCHAPSE in 1983 and ISTAR in 1984 to 1988. I attended an ESF conference in Berlin at the end of 1990. Fourteen organisations were participating in ESF, including three academic. Imperial College was one of the latter. My conclusions at the end of this conference were that these kinds of massive infrastructure systems will only ever be afforded by the biggest organisations, even with European collaboration. But it was good to have a European competitor to the US efforts in support environments. The collaboration could help to achieve standardisation which would otherwise be slower. Conformance to such a standard would make such a software factory more competitive and saleable. But little UK collaboration was visible. Having said that, using a software factory could bring large companies such as BT and ICL into a modern era. Funding the collaborative overheads seemed justifiable, especially for European standards and compatibility. I still had grave doubts about the plausibility of these huge infrastructure systems. The main trend seemed to be towards user-driven and user-definable systems, not massive institutional ones. I would have liked to see more technical issues aired in the conference.

I visited SEMA who were another partner in ESF. After a discussion on their expenditure and claims to the DTI, always a necessary topic on a PO's visit, we talked about the direction the project was going. They were going to put less emphasis on tools, more on the kernel of the system. They wanted to investigate and seriously develop means of integration with other platforms. I urged them to marshal arguments to show how wider benefits would result. For example, they wanted to enable ESF to support the traditional development method, SSADM, which many organisations were still using. Other continental partners included Matra, AEG, Telesoft, Softlab and CGS.

By July 1991 BT had withdrawn from ESF, and a company sprung from the University of Durham, independent but in the university Science Park, had joined. This company was formed mainly because the French participants did not want to subcontract to a university; otherwise the company, albeit "Ltd", comprised members of the Durham computer science department.

ESF, funded through Eureka with the French government funding the French partners 100%, was an advanced generic environment for software development. The aim was to bring a new generation of integration technology to industrial scale software engineering products and practices. The watchword was integration: of software product management, developments of actual pieces of software, configuration control, production of manuals and documentation etc., a framework to support all the computer-based activities of software production. What was believed to single ESF out was its genericity: not oriented to any specific method,

application or programming language. Particular instances of these environments or "factories" can be provided which would be built according to the underlying architecture and conforming to a number of technical standards specific to ESF. That was the idea. It aimed to be the major software environment project in Europe. ESF as a Eureka project was to have a ten-year lifespan from 1986 to 1996. The partners comprised 3 German, 4 French, 1 Swedish, 1 Norwegian and 3 UK bodies. SEMA and ICL were the UK industrial partners, Imperial College the academic one. Durham University's participation was still only a proposal in August 1991.

In February 1991 I visited Imperial College to discuss their latest proposal for taking part in ESF. I had various questions, of the kind I had learned that it was necessary for a PO to ask. Since the previous version of their proposal, the collaboration seemed weaker, because there were no longer any references to sub-projects within ESF; not a good thing from a DTI perspective. One justification for government funding was that it fostered collaboration, which in turn stimulated technological advance. The proposal had notes of pessimism: "*It is widely recognised that there are difficulties with some of the strategies of ESF*". The proposal did not elaborate this any further. This would cause alarm within the DTI, even to the extent of questioning the existing funding that the UK were providing to ESF. They would think, are we being asked to throw good money after bad? IC needed to explain and itemise this, and reassure as to how the difficulties could be resolved. It conflicted sharply with the euphoric style of most ESF publicity. Later on, the proposal offered new and helpful promise of collaboration, to some extent counteracting the earlier weakness. More details of the collaboration mechanisms would help. Each major task would deliver a consultancy report. Were these the only deliverables? They needed to provide a list of all deliverables with time-scales, preferably not all coming to fruition at the end of the project's two-year time span! These deliverables were needed to assist the Monitoring Officer's task (of monitoring the project). I give this detail here to convey the flavour of the relationship between Project Officers and their projects. These are the kinds of questions a PO has to ask and issues to be understood.

Another EC funded initiative that was more of a programme than a project was Eurotra. The programme was funded by the Commission through "Contracts of Association" with the participants, the British participants being two universities, Essex and UMIST. The DTI part-funded the British participants and the equivalent national administrations likewise part-funded the participants in the other EU countries. The European Commission coordinated the whole effort, provided a proportion of the funding and periodically called meetings of project leaders and representatives from the contributing national administrations. I thus found myself attending these, essentially steering group, meetings, as well as funding and other policy meetings within the DTI. The EC chaired meetings were all held in Luxembourg, my

first visit to that country and its eponymous capital. There we discussed and agreed on matters of distribution of results and IPR, the proportions of EC and national government funding for different aspects of the projects, royalties, and matters of that kind. The Commission wanted identical clauses on IPR to be signed in the contracts of association. But also discussed were technical strategies. All participants wanted to work on translation to and from English and most also to/from German; technical matters included common grammatical issues such as morphology[1] and modifiers[2]. The EC wanted final reports on the work done from each member state to cover the whole programme period. Both scientific and management reports were proposed, the latter to relate what happened to the project and what it produced, the former to concentrate on software produced and such like. The whole programme was to cost some 10 million ECUs.

Meanwhile, within the DTI there were the usual requirements for any UK government funded programme: a ROAME had to be written, and with something of this size, the advance expenditure had to be agreed at a meeting involving representatives from several DTI divisions and the Treasury. I wrote the ROAME and then presented it to the the approval meeting. Since the programme had been running for a couple of years, I thought that provided I clearly stated the work to be done and related issues, the decision would be taken on the merits of the case, which since it had been approved for previous years would be quite apparent. In an industrial situation, of which I had had considerable experience, this would have been the case: no industrial organisation would casually put a large investment into something and then halt it before the results had arrived. I couldn't have been more wrong. This was my first time presenting in one of these committee meetings and the experience was ghastly. The process in the DTI, and probably throughout the civil service, was for the protagonist of a case to come with a band of supporters, or at least an ally or two, all of whom could contribute their views in the argument. Decisions were taken not so much on the merits of the case as on the rhetoric and debating skills of its defenders and opponents. The Treasury representatives, and some of the others, would see their rôle as trying to save government spending, of course understandable to a degree. The result was that the committee rejected my ROAME, which would mean that the UK's participation in Eurotra would be stopped. This was disastrous, considering that the teams in two universities had been working on their projects for several years and had every right to expect that funding would continue until their conclusion. One of my grade 7 colleagues in the DTI IT division heard of this calamity and helped me bring about a change in the decision. Together we wrote carefully worded memos to the grade 5 manager who chaired the committee. My colleague explained to me that challenging the committee's decision would not do; it had been taken through an entirely correct process. Essentially I had to say, cap in hand, that it was my fault for not bringing all

[1] Morphology concerns changes to the forms of words, usually endings: has, had, have for example.
[2] A modifier is a general term for an adjective, adverb and also the word change that alters its grammatical rôle such as give - giver - given.

the salient facts before the committee. The one missing fact that we hit upon was that a government minister had put his name to the funding of the British part in the programme at its inception, and to cancel it now would cause him great embarrassment. After several exchanges of memos with the grade 5, he eventually agreed to reverse the committee's decision. Phew. I could go back to Essex University and UMIST and tell them that all was well. I felt somewhat humbled by the fact that the colleague who guided me through this repair process and saved the day was a grade 7, a lower grade than mine. I often wondered if I really deserved to be grade 6.

Because the previous DTI project officer for Eurotra, Dr. Nicolas Ostler, had so much expertise in matters of languages and translation[3], I felt I should at least teach myself something of the technicalities of the subject area. I did not want to be just a government administrator. I asked Doug Arnold, who was the lead researcher at Essex University's contribution to Eurotra, for suggestions on some reading matter that might help me obtain some technical background. He recommended *Lectures on Contemporary Syntactic Theories* by Peter Sells[4], which was in their MSc reading list. I found this most illuminating, and learned that there was much more to grammar and syntax than the traditional parts of speech that those of us of a certain age were taught at the age of nine or so, and which were part and parcel of the process of learning Latin and classical Greek.

An illuminating meeting of the Language Round Table (the government funded programmes had a penchant for inventing these little committees) in Paris revealed some rather interesting statistics: 9.1 million person-years are devoted to the written word each year, which is 19% of the world total: it varies from 22.7% for the lowest level worker ("office worker") to 16.5% for the highest ("high-level executive"); 30% never use DP tools; 5% of foreign language dictionary usage is computerised; 11 people out of 242 (4.5%) used computer-aided translation. These statistics date from 1990.

In June 1991 I attended the Eurotra Advisory Committee in Luxembourg. this committee selected the project proposals that would be accepted into the Eurotra framework. There were rolling calls for proposals every few years, each call designated ET1, ET2 onwards. The deadline for ET9 was at the end of the next month. The committee comprised delegations from twelve European Union countries: Belgium, Denmark, France, Germany, Greece, Republic of Ireland, Italy, Luxembourg, the Netherlands, Portugal, Spain and the UK. There was a question about how follow-up projects resulting from previous calls, even from ET6, related to ET9. Follow-up projects could include implementing software and extending research. Not all scientific problems that previous calls addressed had been solved. ET6 was complete and its final report would be produced the following week. Then the means of selecting proposals in the forthcoming ET10 was discussed; the selection process evoked a

[3] His various recent books, see Ostler 2005, 2007 and 2010, are very accessible to an intelligent lay reader.
[4] See Sells 1985.

lot of criticism. There was a desire for more visibility in the selection process and opportunities for discussion prior to a fait accompli. Objections to the current selection process were voiced by four of the delegations. Then one delegate expressed concern about the absence of contracts of association in the projects. This led to difficulties: some groups had to reduce their size for organisational or financial reasons. Further follow-up work would include the creation of linguistic resources and applications of machine translation. One project selected focussed on reusability of grammars, dictionaries and other linguistic resources of Eurotra. The relationship between ET7 and the EC Third Framework was discussed. Using external experts to assist in the evaluation of project proposals was agreed for the future. Invitations had been issued to the national administrations to nominate experts. The committee needed the list by the end of September 1991 for ET10. Evaluators would have to declare their interests and to abstain from evaluating their own proposals, or competing ones in some areas.

By the time I came in on the act, the ROAME for the SALT (Speech and Language Technology) programme was at its seventh draft and was still being scrutinised closely by many parties. A steering group codenamed “Link” had been set up to examine the ROAME and subsequently to play a part in overseeing the programme. The forecast was for 15 projects and an expenditure of £14M over four years. After that the club was expected to be self-financing. Meanwhile the DTI and SERC would fund two workshops per year. The participants would include academics and SMEs, but enable exploitation by large corporations. The ROAME included claims of UK leadership in the field, scientific capabilities, economic interest and involvement of SMEs. At last the ROAME gained acceptance from the governmental approval committees and was ready for presentation to ministers.

In January 1991 I gave a presentation on SALT to an international audience in Versailles. The programme would include speech processing: the acoustic basis of speech and computer analysis of spoken information; assigning meaning to natural language text in computer accessible printed form; and script recognition, the analysis of hand-written text for computer input. We would aim to include international collaboration in some projects and widen awareness by encouraging new entrants to the technology.

All fourteen projects within SALT contributed to the workshop held in January 1991, and again to that held the following June. By September 1991 the role of SALT was widening. The SALT programme could fund Eureka projects. SALT projects could involve foreign, such as US, partners, funded by foreign sources. But the UK government would not fund foreign partners. More adjustments were made to the structure of the coordination committee, an assessment panel, means of reviewing, publicity and so on.

One of the most famous software engineering institutions in the world is the SEI, Software Engineering Institute at Carnegie-Mellon University in the USA. They had devised a means of assessing the capability of a software development organisation, called the Capability Maturity Model[5] - CMM. Five levels of maturity were defined, and there was a battery of questions which, if answered honestly, would place an organisation at a level in the model. This maturity model became very well-favoured and used internationally. It was part of a more general investigation into the Software Process[6], the process of building and maintaining software from inception to retirement, as opposed to the Software Product, which is the actual software and associated documents, test data and other delivered products. Part of the reason for the popularity of the CMM, I believe, was that it is extremely difficult to assess the actual quality of a given piece of software. So assessing the ability of the team that built it was the next best thing. (This may be considered a slightly cynical view). However, given this background, it was interesting to have a visit from a member of the SEI in October 1990. The SEI had a budget of $30M, and employed 150 people. They were 100% funded by the US Department of Defense.

In November 1990, in my rôle as PO for formal methods projects, I accompanied a scientific officer from the SERC to the Laboratory for Foundations of Computer Science at Edinburgh University to review the projects funded by SERC grants to Rod Burstall and Gordon Plotkin, who founded the LFCS along with Robin Milner and Matthew Hennessey. We were given a series of presentations about the research work going on there. I was extremely impressed, especially by the work being done on Extended Calculus of Constructions. The Calculus of Constructions is a formal language which can express both computer programs and mathematical proofs. It was developed in the late 1980s by Thierry Coquand and Gerard Huet[7] at the French research centre, INRIA. In CoC, types can be first-class values, that is they can be assigned to variables, passed as parameters, returned as results and constructed at run-time[8]. This theory enables the building of practical proof assistants, and was the basis of the proof tool, Coq. The LFCS were extending CoC to allow reasoning about abstract structures such as groups, topologies, etc., whereas CoC enabled the definition of concrete mathematical structures such as natural numbers, lists etc. Thus in ECC, first class objects also included theorems and proofs.

[5] See Paulk et al. 1993.
[6] See Humphrey 1988 and 1989.
[7] See Coquand and Huet 1988.
[8] The term, “first class value” was coined by Christopher Strachey in the 1960s. He advocated that functions should be “first class citizens”. See Burstall 2000.

Our review meeting was held in Edinburgh University's James Clerk Maxwell building, which then was the home of LFCS. I was embarrassed and, indeed, rather shocked that my co-visitor from SERC did not know who James Clerk Maxwell was.

A few weeks later I was again with a SERC panel reviewing the rolling SERC grant for "Foundational Structures for Computer Science" at Imperial College, led by Samson Abramsky. The Computer Science department at IC had eight sections. There were 70 PhD and MPhil students in its post-graduate programme, which included both conversion and advanced MSc courses. Sixty research projects were in progress, 20 funded by ESPRIT, 25 by SERC and other UK government sources, and 15 funded by industry and other agencies. Along with the LFCS at Edinburgh and the PRG at Oxford, the IC CS department was one of the leading academic computer science research groups in the UK.

We were given presentations by Tom Maibaum, Samson Abramsky, Steve Vickers and Mike Smyth. Their investigations into the mathematical foundations of CS focussed on "deep structures", the basis of effective analytical methods. They were building on work done in the previous two years, which had studied domain theory, models of polymorphism and information systems where infinitary objects are defined as limits of of finite objects, after the methods of Scott. The results promised to unify areas in which the current research efforts were largely separate. There was also a deal of work in progress on the connections between topology and domains, topology and logic. Once more, it was clear that radical new theoretical research was being effectively pursued at Imperial College.

The Alvey project *Poetic* intended to establish a dynamic database about road conditions in the UK, linked to police mobile radios. The partners were the AA (Automobile Association), the Independent Broadcasting Authority, the University of Sussex and Racal. There was liaison between the project and the Home Office, (who were in effect the authority for the police then). The principal planned output from the project was to be a demonstrator version of the software system; certain Alvey projects were known as Alvey demonstrators. A prototype that could demonstrate the feasibility or otherwise of an idea which could then be taken forward was a legitimate justification for Alvey funding.

Sussex University were to carry out all the software implementation, using Poplog. Poplog is a multi-language software development environment supporting, among other languages, ProLog and LISP. Sussex University had also developed Poplog. The software was to support natural language processing, in particular, police traffic reports. One difficulty they had discovered was that police in different areas used different vocabularies, although in a restricted domain. Other developers could be brought in to help develop the software in the future. Sussex University would undertake the technology transfer, dissemination being a

required facet of Alvey projects. Cartographic data would form part of the system; supplying this would be the AA's rôle. The IBA would be responsible for providing access to radio data systems.

In December 1990 I received a proposal for a Eureka project from a UK company that none of us had heard of. The proposal was interesting if only because it was utterly infeasible. There were to be two partners, the other one an Italian company, but it was not clear from the proposal whether the two were in the same group: if so, it would not have been truly collaborative and therefore not eligible. The proposal was for a 32-bit microprocessor designed using formal methods, and therefore suitable for safety critical applications. Some work at the time had been done on formal methods applied to hardware design, but the proposers made no reference to such previous work or how they would build on it. The really stunningly ambitious part of the proposal was that the proposed machine would execute high-level real-time languages directly. This would have moved enormously more complexity to the hardware and to run-time: lexical and syntax analysis, name reference resolution, etc. They proposed to "expand formal design and verification methods to system engineering, production, manufacturing testing...". all in a two-year project. Furthermore, they planned to build a CAE environment to support it all. What mainly worried me was that they proposed to develop a product, yet there were a whole string of research questions needing to be answered on the way. We had never heard of the company, and if they were capable of doing this kind of work I suspect we would have done.

That was one request for government funding that did not get off the ground.

Although the Monitoring Officer kept watch over a DTI funded project at least quarterly, the Project Officers were also advised to visit their projects once a year or thereabouts if possible. I was Project Officer of some twenty projects and at this beginning of my stint at the DTI, I thought it wise to visit as many of them as I could, to get some familiarity with the individual projects and to get a general view of their landscape and context. The next one I visited was the ReForm project, run by IBM, Durham University and CSM Ltd. The purpose of the project was to produce a system of tools that could take software written in assembler language and generate from it a formal functional specification that was an abstract model of it, in the Z notation. IBM wanted to develop this system in order to assist the maintenance of CICS. CICS is a transaction manager that oversees multi-part transactions that must not be interrupted, such as financial transactions initiated by banks, and manages restore actions in case of breakdowns. It has many other features. IBM would share this tool with the community of CICS users, customers of theirs. CICS was first released in 1969 and has undergone further development ever since. It was an example of *legacy code*: software dating

from possibly many years in the past, that may work well and be useful, but whose details of design have long been lost. The process of taking legacy code and deriving design and specification documents from it is known as *Reverse Engineering*. In the normal, or perhaps ideal, development process, one works from a statement of requirements, produces a specification, then a design and finally an implementation, code that runs on a computer (I am simplifying to an extreme degree here). Reverse Engineering in a sense reverses this normal engineering process, by deriving a design or a specification from the final product. This can be a notoriously difficult process, because one has to find the structure of the wood when only being able to see the trees.

Thus, I paid a visit to the project at IBM Hursley research laboratory in January 1991. This gave rise to a slightly amusing event. In welcoming me, as we were settling ourselves in the meeting room, my hosts asked me if I had ever visited IBM Hursley before. As it happened I had done a vacation job there while an undergraduate. Yes, I said, in 1961. There was a moment's stunned silence and I realised that this date was probably before some members of the team had been born!

The project was going well, the collaboration agreement was being signed and arrangements for approving the publication of results were well in hand, something the DTI wanted to happen so that the benefits would be spread as wide as possible. Collaboration agreements between the partners were also needed so that the intellectual property could be shared, but the agreements could take a very long time to be negotiated and sometimes were signed only at the end of all the work. The project control was the most superb I had seen so far, with computer aids that actually seemed to be a help and not get in the way. The project was now in the last quarter of its term and the team was making preparations for post-project evaluation and exploitation.

The DTI maintained a bipolar relationship with its industrial and academic funding beneficiaries. It could not be seen to direct their technical policies: that would be unwarranted and unpopular government interference. On the other hand, any funding had to encourage national economic and technical advancement. The DTI guidelines for eligible projects and programmes were therefore broad but strictly applied. Within those guidelines the funding beneficiaries (we used to call them the "punters") could make their own technical decisions without interference. To maintain a transparent profile, the DTI would hold conferences for projects in progress within a programme, and workshops for establishing programmes at their initiation. One of the latter was a workshop introducing the Systems Engineering programme, held in March 1990. The purpose of this was both to gain views from technological leaders in companies and academe, and to communicate to them the objectives and emphases of the forthcoming programme. The Systems Engineering programme was to integrate two previous

programmes, Systems Engineering and User Enhanceable Systems. The driving emphases were to be productivity, technology integration and benefits for the user. Increasing the users' productivity would benefit the national competitive edge. This was the flavour, rather than technological advancement for its own sake.

After debating this, we reached a form of words which summarised the policy of the programme: "To enhance the productivity of IT systems users in UK business by establishing requirements for and researching and delivering the right technologically integrated scientifically-based advances, in both the medium and long term". There was further debate on this. The programme should include the enabling of users, and the link between productivity and effectiveness. A few more sentences were agreed which gave the definition of the programme. These would be incorporated into a call for proposals and proposals would be assessed on their conformance to those objectives. These workshops were a part of the process of formulating a new programme of funded R&D.

The DTI's Information Technology Directorate was quite relaxed about enabling its officers to attend conferences and other technical events. This may have been partly due to the head of ITD being John Buxton, an academic also on secondment. ICALP 90 took place on the campus of Warwick University. ICALP stands for International Colloquium on Automata, Languages and Programming. ICALP is an annual international colloquium sponsored by the European Association for Theoretical Computer Science. It first took place in Paris in 1972, and in 1990 twelve European countries had hosted it. ICALP 90 was an intensive but very interesting five days incorporating 57 papers grouped into 17 sessions. Parallel sessions were, of course, inevitable. ICALP is one of the most prestigious international computer science events and it was a privilege for me to attend it.

Another interesting conference that year was a workshop on Concurrency, held at Leicester University. There is in practice little difference between a workshop and a conference, although workshops are intended to present more recent work that may be in a less final state, and be opportunities for discussion amongst researchers working in the same areas. Concurrency is that research area which studies and develops the theories behind processes that proceed in parallel, at the same time, and is motivated by modern digital electronics involving multiple processors, and software working under the control of time-sharing operating systems, both of which are now ubiquitous. At this workshop Robin Milner introduced his idea of "mobile processes", which was a development of his CCS[9]. In Mobile Processes, processes themselves can be created, destroyed and passed as arguments to other

[9] See Milner 1980.

processes[10]. Bill Roscoe described a theory of Tony Hoare's CSP, and gave an account of the difference between CSP and CCS. CSP, Communicating Sequential Processes, is defined in terms of a mathematical model, whereas CCS is defined as a calculus. (A calculus is a set of rules for transforming terms or formulae, without particularly prejudicing what those formulae denote, whereas a mathematical model will be a specific algebra, for example). Other papers dealt with temporal properties (Colin Stirling), causal semantics (Samson Abrasky), multi-traces (Antoni Mazurkiewicz) and much else besides.

These various theories of concurrency, process algebras like CCS and CSP, Petri Nets and Temporal Logic, are principally concerned with the mathematical modelling of concurrent processes, so that, if a concurrent system conforms to an interpretation of the theory, then one may reason about it and draw deductions about its behaviour. None of the theories are specifically designed as a specification language to be used while developing a concurrent computing or electronic system, although in principle most of them could be, at least to some extent. A specification language was developed, based on CCS, CSP and algebraic specifications of abstract data types, called LOTOS[11]. FACS held a one-day meeting on LOTOS in September 1990, soon after the language was published.

VDM Europe, ongoing then for four years, was planning its next symposium, VDM '91. The organisation was still being funded by the European Commission, with project officer Karel de Vriendt. He was the group's "champion" in the EC. Every funded initiative needs a champion in the governmental funding organisation, EC or DTI; the same probably applies in local government. However, he warned us that this funding could not continue indefinitely. We were technically classified as an "advisory group", i.e. giving advice to the EC. Further funding could only be justified if VDM Europe widened its scope. Proselytising, communicating the technology to a wider audience and therefore benefiting European industry and academia, would make the funding easier to justify. We discussed whether to widen our scope to include other formal methods such as RAISE, CSP, LOTOS. For the time being at least we would continue to focus on VDM. But VDM '91 should contain two tutorial tracks, one advanced and the other more "elementary".

Alexander Moya took over the rôle of project officer in January 1991. In that year we decided to broaden our scope to include other formal methods, but limited to other model-based methods for the time being: these in practice included RAISE, Z and B. A draft charter for the organisation was distributed. A more umbrella-like structure was adopted, with a controlling body and other subcommittees dealing with various activities, some permanent,

[10] See Milner et al 1989 and 1992.
[11] See ISO 8807:1989.

others temporary. Examples were a newsletter, technical events, the symposia planned at 18 month intervals, and perhaps other activities such as standardisation. This committee meeting in January 1991 was to be the last meeting of VDM Europe; henceforward it was to become Formal Methods Europe.

My secondment to the DTI had another eight months to run. I knew that at the end of that, my manager at Praxis, John Thornton, would want to end my six-year long arrangement of being notionally based at my home in London, and I would have to be a normal employee, based at the firm's location in Bath. I didn't want to pay the expensive commuting costs myself, and with my family ensconced in London; with my wife in a London based job and my teenage children at a crucial stage in their schooling, I did not consider moving to Bath. So I started to plan working for myself as an independent consultant when the time came. I thought there was a good chance I could secure another secondment contract with the DTI. At the same time, at the suggestion of my ex-colleague Roger Shaw, who was already working there, I approached Lloyd's Register.

Lloyd's Register, previously called Lloyd's Register of Shipping, originated from Lloyd's Insurance. Its original purpose was to assess the insurance risks of seagoing vessels. LR would maintain a list, known as Lloyd's List, of seaworthy vessels. Now, an independent organisation, it covers the safety of lives and property more generally and actively promotes safety. First, LR developed rules of design and construction. Then surveyors would check vessels on a five-year cycle. Fees provide 50% of their income. Over the last 50 years LR have expanded into more industrial areas such as power generation and civil engineering, and they now have 300 offices throughout the world.

Over the previous few years software had become embedded into engineering control systems. LR recognised that they needed to be able to certify and assess software to ensure that it was fit for purpose. They wanted to devise guidelines for the development of dependable systems containing software. They wanted to establish their competence in expert systems and formal methods. A software department had been set up and had grown from 30 to 40 over the previous six months. We discussed a consultancy contract for me in which I would work two days per week to help them devise a software certification procedure. My ultimate embarkation into independent consultancy in eight months' time was beginning to look almost secure.

In April 1991 BCS FACS ran a three-day tutorial on B[12]. B is a formal system for describing abstract machines that can be used as abstract models of computer systems, and hence as formal specifications of systems. J-R Abrial developed B while working as an independent

[12] See Abrial 1996.

consultant and as a researcher at the Oxford University PRG. An abstract machine is defined as a formal system consisting of data types and operations with state: that is, the machine contains a collection of variables of specific types whose values can be altered by the operations. Operations can have input values and output values, but can also alter the state of the machine; they are therefore not like mathematical functions, which do not have "state", and an abstract machine is for that reason not quite the same as a program in a functional language. A program in a functional language does not contain a state.

The B language is a formal language, which technically consists of an alphabet with rules of formation of well-formed formulae (wff). Along with the formal language there is a deductive system consisting of axioms and rules of inference that determine what wff may be deduced from some given wff(s). There are automated proof assistants and other tools for B; the one demonstrated at this tutorial was called the B-Tool. (Another was subsequently developed in France under Abrial's guidance called Atelier B).

In this three-day tutorial, David Till of City University presented logic and deductive system for B and J-R Abrial presented the B language and B Proof Assistant. B, he declared, consists of the B Tool, the Theory of Abstract Machines, and Abstract Machine Tool-kit. David Nielsen presented more details of the Abstract Machine tools, the type-checker, proof obligation generation, refinement and other features.

This course was a hard, stimulating three days. I found the notion of abstract machines very reminiscent of VDM; indeed VDM, Z and RAISE are all examples of approaches based on the concept of abstract machines, in addition to B. All these languages, based on abstract machines, that is data types, operations and a state, became known as model-based specification languages.

Over the subsequent years, B has been used in important safety-critical projects. The Paris Metro line 14 was started in 1993 and opened to the public in 1998. It is completely automated with driverless trains. The critical parts of the control and signalling software were specified and designed using the B notation and method, which allowed for stages of refinement down to a version that can be translated into a conventional programming language. The improved confidence in the safety of the overall system meant that trains can be scheduled more closely, 85 seconds apart with trains arriving at a platform just as the rear of the previous one is leaving.

The DTI Subcommittee B would review funded projects, along with its other duties. One of the speech and language projects was "Aviator", which was developing text searching and retrieval software with facilities for lexical clustering and filters that interfaced with the retrieval mechanisms. The academic partner was the University of Birmingham, with the publisher Collins (who produce continuous editions of a collections of dictionaries), Nimbus

Records and BRS (British Road Services). This was one of a long list of other projects: PEBA, headed by SEMA, aiming to build an environment supporting SSADM; FERESA, again headed by SEMA for an environment supporting JSD; the Eureka project ECMA PCTE already mentioned; the Eureka project Europicon (European Process Intelligent Control) which aimed to make advances in process control using knowledge-based systems; and the British National Corpus.

The British National Corpus was led by Oxford University Press, that part which published the renowned Oxford Dictionaries. The project aimed to build an ongoing corpus of English text, a "bank of English". Again, to be useful, such a corpus of work needs search, retrieval and related software. Another project planned to enable computer analysis of the content of spoken discourse.

One of the concerns the DTI had when monitoring Eureka projects, which had British and European partners, was to ensure that the benefits after the project, such as royalties, were equitably distributed, not biassed in favour of another participating country. The ROAMEs for these projects had to justify government funding. Distance from market of the proposed research would often be cited, yet there had to be an eventual market for the work. The proposal had to demonstrate that the aims were technically viable. The work had to be innovative and "pre-competitive", that is, at an early enough stage that there was no competition, for the DTI could not favour one firm over another. While collaboration was a requirement, it also had to have merits for the project: usually a case would be made that no one organisation had all the requisite skills or experience. The proposal would have to show knowledge of previous related work so that it could build upon it, and the ROAME would in turn bring such points out. High risks, long time-scales to market, a generic approach, wider eventual benefits could all add justification to the case. A very detailed project plan would be required with milestones and deliverables allowing the DTI to monitor the progress of the project effectively. Such was the lengthy and meticulous work needed to construct not only a proposal by the partners but a ROAME by the DTI project officer.

In July 1991 I took part in the JFIT conference, which the DTI organised and hosted every year. Some dozen industrial and academic representatives presented their views and experiences, and the conference finished with a panel session and subsequent discussion. Some, to me, surprising views were expressed: one academic asserted that IBM had saved a seven figure sum by using Z to specify CICS, and that PROLOG and Type Theory originated in the UK. I think that some French academics might vigorously disagree with that asserted origin of PROLOG, and I wasn't sure he would have had access to enough detail of IBM's finances to know the extent of saving due to the use of Z. The ADJ group in the USA might reasonably claim the application of Type Theory to computer science, although I suppose one

might refer to Russell and Whitehead's theory of types in their Principia Mathematica which long pre-dated computers. Nonetheless, the JFIT conferences were always a fruitful networking event.

The DTI did not have a technical policy in IT themselves: it would not do for government to influence industry's market-driven R&D. On the other hand, the DTI tried to predict what would be the future needs for IT, so that they could foster the developments that would meet those needs and thereby boost the UK's IT industry. So papers were written, studies conducted and meetings held to try to determine the future needs for IT: this was called variously the "Forward Look" and "Strategic Overview of IT". How does one predict future trends in IT needs? Straightforward extrapolation of current trends does not work; for example, the sixties and seventies saw an ever-increasing use of main-frame number crunching computers. Yet in the two next decades, these fell by the wayside and networked workstations and desk-top computers were coming into their own. Advances in AI had almost halted, but were beginning to move on again, more slowly. The surge of formal methods had been checked, but still had a continuing demand. The decreasing price of chips had many repercussions that were hard to foresee. The need for *integration* in general was prevalent in these DTI papers and discussions. Users could easily identify their specific problems, but less easily see how to combine solutions: for example, how can I integrate my point-of-sale transactions with my stock control programs, my stock control programs with financial management and forecasting? How can I integrate my sales processing and invoicing with my raw materials ordering, financial forecasting, throughput forecasts, process control, day to day staff planning? My new product design with longer term financial, site requisition forecasting? It could be highly advantageous if the IT systems supporting all these activities were integrated and worked together.

Another approach to predicting the needs for IT is to observe social trends: they follow each other. IT systems support social structures and activities, such as processes between people in a work-place, or processes used to develop engineering systems, including software intensive systems. The architecture of the IT systems supporting people at work depends on current social trends: one can envisage two scenarios, one community-oriented and the other individual-oriented. In a community-oriented scenario, users form a large team who identify with the objectives and products of the team. These are embodied in the information structures and processes which are held centrally on (in 1991) a mainframe computer. All the users have access to these central resources via, again in 1991, a terminal. They agree to comply with the centralised procedures, access rules etc. and cooperatively construct powerful, large information structures and processes. The knowledge that they have contributed to this effective powerful system, in which they are in direct contact, gives the individuals in the team a personal feeling of gratification, protected because of being part of

something large, but not being individually exposed. The sense of community is enhanced by communication facilities such as electronic news-boards, email etc. The user feels as if logged into a corporate consciousness, losing individual identity but gaining an identification with a more powerful and authoritative composite.

In an individual-oriented scenario, users develop their own, more individual, information structures and processes, sharing them through the communications facilities. The items may be integrated later into the "official" system. Each individual feels identified with the items for which they are specifically responsible. Centrally available facilities are presented to the user as personal tools that assist and add to the power of the individual: spreadsheets, word processors, fourth-generation languages etc. were typical in an office system. Through these the user can build complex information structures. This power gives the user a sense of fulfilment and individuation.

Hybrid systems are possible and indeed frequently occur. Organisations that are building software systems or systems that require a high degree of discipline, for example where quality or reliability is paramount, will tend to the community-oriented scenario. Where the systems developed are largely for use within the organisation, the individual-oriented scenario is more likely. Within the DTI itself one could observe that we used a hybrid system, with common structures such as divisional and departmental files to which access was disciplined, but with a heavy emphasis on the individual-oriented system: each user can use a battery of common facilities ― database packages, word-processors, calculator, spreadsheet, graphics and drawing packages. But other facilities such as data management were community-oriented.

In principle a system can be produced by either of these social structures, or by any which lies in the spectrum between them. The resulting system will have facets that depend on which social structure was used to construct it. In particular, high reliability requires a discipline which fits more easily into the community-oriented model. Otherwise, the choice of what type of system will be popular depends on the preferred social structure. In the 1990s the trend was far more towards the individual, and will probably continue for a considerable time.

In the 1990s one could perceive a broad trend towards constructivism: the individual as authority, owner of rights, creator, and the demise of the institution as regulator of people's lives and values. This was exemplified by the then recent political changes; trends in intellectual thought (philosophy, religion, psychology); social attitudes and public behaviour; a general atrophy of public deference; economic trends (more people than ever before owning cars, personal bank accounts, houses etc.) at least in the West. A small symptom of this is that people tend to ignore instruction manuals: they don't want to be told what to do. People prefer to try out an artefact themselves, straight off. The result is that instruction manuals are

ignored and become deficient, written in awkward unpolished language. There is almost a tolerance of faulty engineering provided that useful features are provided. By contrast, the Japanese had a tendency more towards community-oriented modes, and as a result they showed a capacity to produce goods of high quality and reliability.

A two-pronged approach was proposed: a large prong and a small one. Both were essential. The large prong is an enhanced user-machine relationship: interfaces, speech and language processing, computer-vision, integration of facilities and function, which required research into the abstraction of functions in order to coordinate them. These were reflected in parts of the DTI ITD programme. The small prong was that part of systems that users want to feel secure about, so that they can ignore it, so that their personal power is facilitated: for example, engine control systems in cars and other vehicles. The end-user does not want to be concerned with them, but wants them to work transparently.

The foregoing was a perception and opinion summarised from papers circulating in the DTI's "Forward Look" cogitations. Much of it still applies today, although some of the supporting technology has changed. Networked PCs with a central server would now replace a mainframe and terminals. News-boards have been replaced by blogs, although they are in essence a new name for much the same thing. There are today other more sophisticated, often web-based, tools and facilities. Over the subsequent decade, during the noughties, that trend in the UK began to reverse: the individual became more regulated by the community's rules. But now, in the 2010s, we are seeing a societal rebound back to individuation.

The Fifth Refinement Workshop, sponsored by Lloyd's Register, Program Validation Limited and the DTI, and organised by BCS FACS, took place in the imposing board room of Lloyd's Register's offices in London from 8th to 10th January 1992. The proceedings were published by Springer-Verlag in their Workshops in Computing series[13]. There were 19 papers presented, including an opening address from Mr. Patrick O'Ferrall, the Deputy Chairman of Lloyd's Register. There were also demonstrations of eight support tools for formal methods: the Genesis Z Tool from Imperial Software Technology; μral from Manchester University; Specbox from Adelard; RED (Refinement Editor) from Oxford University; the RAISE Toolset from CRI; Cadiz from York Software Engineering; the SPADE Theorem Prover from Program Validation Limited; and a refinement tool from the Victoria University of Wellington, New Zealand. Program Validation Ltd. hosted a most enjoyable social evening in the London Transport Museum, at which there were refreshments and a delightful recital from a string quartet. These Refinement Workshops run by FACS were becoming the major regular formal methods events in the UK, and the Fifth was a particularly memorable one.

[13] See Jones, Shaw and Denvir (Eds.) 1992.

As I have said, my secondment from Praxis was coming to an end, and my manager at Praxis did not want to renew it: secondment to the DTI was not very profitable for the company. I decided to leave Praxis and try to renew my secondment myself. This required writing – a ROAME! All proposals needed a ROAME, and my own secondment was no exception. I would need to outline the scope of the problem that my leaving would create, the Eureka and other projects for which I had project officer responsibility, why I was uniquely qualified to fulfil the rôle and why a "normal" civil servant in the department was not so well qualified to perform it. There would have to be "deliverables" and the normal criteria for an acceptable ROAME, monitoring arrangements and all.

So my ROAME proposed that the DTI engage a consultant to provide the technical expertise necessary to appraise proposals for Eureka projects in IT and to monitor them when approved and in progress. I estimated the support required would be one day of consultancy per week, with some allowance for travel, over three years in the first place, subject to review after that time. I referred to the DTI policy that required it to have responsibility for Eureka projects (these ROAMEs had to go before committees, not all with DTI staff, so contextual explanations were always necessary), the activities that this would involve and the skills that were necessary. Those skills I identified as knowledge of DTI procedures, knowledge of Eureka criteria, good knowledge of IT and software engineering so as to be able to assess the technical aspects of proposals and the competence of proposers, familiarity with UK IT companies and academe, familiarity with European IT and software engineering companies and academe, an understanding of the differences and nuances between UK and European IT culture, and some understanding of European administration: Eureka, ESPRIT and the EC. Then a statement that this mixture of skills was not available in ITD and the ROAME therefore proposed engaging a consultant to fill the rôle.

Next I wrote some words about the scale of the activity, visits to industrial and academic sites, the need to attend some international events and conferences. I was, of course, uniquely qualified to do the job, my ROAME was approved and I was re-engaged.

Chapter 12 Independence Days

<to be writ>

References

A. E. Abdallah, C. B. Jones, J. W. Saunders (Eds.): Communicating Sequential Processes: The First 25 years. LNCS 3525, Springer 2005.

J-R Abrial, The B Book, Cambridge University Press, 1996.

K. R. Apt: Obituary: Edsger Wybe Dijkstra (1930 – 2002): A Portrait of a Genius, Formal Aspects of Computing, Vol. 14, no. 2, pp. 92-98, 2002.

J. W. Backus: The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference, ICIP Proceedings, Paris 1959, Butterworths, London, pp. 125-132, 1960.

D.W. Barron, J.N. Buxton, D.F. Hartley, E. Nixon, and C. Strachey: The main features of CPL, The Computer Journal 6:2:134-143, 1963.

D. Bjørner, C. B. Jones, M. Mac an Airchinnigh, E. Neuhold (eds.): VDM '87; VDM – A Formal Method at Work, Springer-Verlag LNCS 252, 1987.

R. S. Boyer, J. S. Moore: A Computational Logic, Academic Press, New York, 1979.

Grady Booch: Object Oriented Design, USAF Academy, Colorado, 1980.

Taylor Booth: Sequential Machines and Automata Theory, John Wiley and Sons, New York, 1967.

BS 5750-8:1991, EN 29004-2:1993, ISO 9004-2:1991 Quality systems. Guide to quality management and quality systems elements for services; British Standards Institution, 1991.

R. Burstall and J. Goguen: Putting Theories together to make Specifications, in Reddy (ed.) Proceedings, Fifth International Joint Conference on Artificial intelligence, pp. 1045-1058. Department of Computer Science, Carnegie-Mellon University, 1977.

R. Burstall and J. Goguen: The Semantics of Clear, a Specification Language, in D. Bjørner (ed.) Proceedings, 1979 Copenhagen Winter School on Abstract Software Specifications, pp. 292-332, Springer-Verlag LNCS 86, 1980.

Rod Burstall, "Christopher Strachey—Understanding Programming Languages", Higher-Order and Symbolic Computation 13:52, 2000.

A. Church: The Calculi of Lambda Conversion, Annals of Math. Studies no. 6, Princeton University Press, 1941.

E.F. Codd: A Relational Model of Data for Large Shared Data Banks, Communications of the ACM 13 (No. 6): pp. 377-387. Association for Computing Machinery, 1970.

P. Cohn: Universal Algebra, Mathematics and its Applications vol. 6, Harper & Row 1965, revised edition D. Reidel Publishing Company 1981.

S. D. Conte, H. E. Dunsmore, V. Y. Shen: Software Engineering Metrics and Models, Benjamin Cummings, 1986.

D. J. Cooke, H. E. Bez: Computer Mathematics, Cambridge Computer Science Texts vol. 18, Cambridge University Press 1984.

D. J. Cooke: Constructing Correct Software, Springer -Verlag, 1998.

Thierry Coquand and Gerard Huet: The Calculus of Constructions, Information and Computation, Vol. 76, Issue 2-3, 1988.

B. T. Denvir, W. T. Harwood, M. I. Jackson, M. J. Wray (eds.) The Analysis of Concurrent Systems, Proceedings, Cambridge September 1983, Springer Verlag LNCS 207, 1985.

Tim Denvir: Introduction to Discrete Mathematics for Software Engineering, Macmillan, 1986.

Tim Denvir: System Specifications, in Software Engineering, Blenheim Online Publications, pp.11-18, 1988.

Tim Denvir: The Rôles of Mathematics in Software Engineering, in Mathematical Structures for Software Engineering, Clarendon Press, 1991.

K. Devlin: The Joy of Sets: Fundamentals of Contemporary Set Theory, Second Edition, Springer-Verlag, 1994.

E. W. Dijkstra: “Cooperating Sequential Processes” in Programming Languages, F. Genuys (ed.), 1968.

E. W. Dijkstra: Goto Statement Considered Harmful, Letter to the Editor, Comm. ACM, Vol. 11, pp.147-8, 1968.

E. W. Dijkstra: Guarded Commands, Non-determinacy and the Formal Derivation of Programs, Comm. ACM, Vol. 18, no. 8, pp. 453-7, 1975.

E. W. Dijkstra: A Discipline of Programming, Prentice-Hall, 1976.

ECMA: Standard ECMA 149: PCTE Abstract Specification, December 1990.

H. Ehrig, B. Mahr: Fundamentals of Algebraic Specification I, Springer-Verlag 1985.

J. R. Ennals: Star Wars: A Question Of Initiative, Wiley, 1986.

Erwin Fehlberg: Low-order classical Runge-Kutta formulas with step size control and their application to some heat transfer problems. NASA Technical Report 315, 1969.

Robert L. Glass: Software Runaways, Prentice Hall PTR, 1998.

M. J. Gordon, A. J. R. G. Milner, C. P. Wadsworth, Edinburgh LCF: a Mechanised Logic of Computation, Lecture Notes in Computer Science 78, Springer-Verlag 1979.

Anthony Hall, Seven Myths of Formal Methods, IEEE Software, September 1990, pp 11-19.

M. Halstead: Software Physics – Basic Principles, IBM Research Journal 1582, May 1982.

C. A. R. Hoare: An Axiomatic Basis for Computer Programming, Comm. ACM, vol. 12, no. 10, pp. 576-80, 583, Oct. 1969.

C. A. R. Hoare: Communicating Sequential Processes, Comm. ACM, vol. 21, no. 8, pp. 666-777, Aug. 1978.

A. Horn: On sentences which are true of direct unions of algebras, *Journal of Symbolic Logic*, 16, 14-21, 1951.

W. Humphrey, "Characterizing the software process: a maturity framework". *IEEE Software* **5** (2): 73–79, March 1988.

W. Humphrey: Managing the Software Process, Addison Wesley, 1989.

ISO/IEC 13719-1: Information Technology - Portable Common Tool Environment (PCTE) - Part 1: Abstract Specification. October 1998.

ISO/IEC 13817-1: Information technology. Vienna Development Method. Specification language. 1996.

ISO/IEC 13568: Information technology. Z formal specification notation. Syntax, type system and semantics. 2002.

ISO/IEC 14977 Information Technology – Syntactic metalanguage – Extended BNF, 1996(E).

ISO/IEC 19501:2005 Information technology — Open Distributed Processing — Unified Modelling Language (UML) Version 1.4.2. 2005.

ISO 8807:1989 Information processing systems – Open Systems Interconnection – LOTOS – A formal description technique based on the temporal ordering of observational behaviour. 1989.

ISTAG report on the Grand Challenges in the Evolution of the Information Society, ftp://ftp.cordis.europa.eu/pub/ist/docs/2004_grand_challenges_web_en.pdf, EU, 2004.

M.A. Jackson: Principles of Program Design. Academic Press. 1975.

J. H. Johnson, M. J. Loomes (eds.): The Mathematical Revolution Inspired by Computing, Clarendon Press, 1991.

G. Khan, ed. Semantics of Concurrent Computation, Proceedings Evian, 1979, Springer-Verlag 1979.

D. Knuth and P. Bendix. "Simple word problems in universal algebras." Computational Problems in Abstract Algebra (Ed. J. Leech) pages 263--297, 1970.

T. S. Kuhn: The Structure of Scientific Revolutions, 1962, Second edition, University of Chicago Press, 1970.

M.A. Jackson: Principles of Program Design. Academic Press. 1975.

M. A. Jackson: System Development, Prentice-Hall, 1983.

M. I. Jackson, B. T. Denvir, R. C. Shaw: Experience of Introducing the Vienna Development Method into an Industrial Organisation, in Formal Methods and Software Development, Lecture Notes in Computer Science 186, Springer-Verlag, 1985.

C. B. Jones: Software Development, a Rigorous Approach, Prentice Hall 1980

C. B. Jones: Systematic Software Development Using VDM, Prentice Hall 1986 (Second edition, 1990)

Cliff B. Jones, Roger C. Shaw, Tim Denvir (Eds.) 5th Refinement Workshop, Springer-Verlag, 1992.

I. Lakatos: Proofs and Refutations, Cambridge University Press, 1976.

P. J. Landin: The Correspondence between Algol60 and Church's Lambda-Notation, Comm. ACM, vol. 8, nos. 2-3, pp. 89-101 & 158-166, Feb. – March 1965.

M. M. Lehman, L. A. Belady: Program Evolution – Processes of Software Change, Academic Press, New York, 1985.

E. H. Mamdani, B. R. Gaines (Eds.): Fuzzy Reasoning and its Applications. Academic Press, 1981.

J. McArthy: Towards a Mathematical Science of Computation, in Information Processing, North Holland, 1963.

John A. McDermid (Ed.): Software Engineer's Reference Book, Butterworth-Heinemann, 1991.

R. E. Milne, C. Strachey: Theory of Programming Language Semantics. Chapman & Hall, 1977.

Robin Milner, Logic for Computable Functions: description of a machine implementation. Stanford University, 1972.

Robin Milner, A Calculus of Communicating Systems, Lecture Notes in Computer Science 92, Springer-Verlag, 1980.

Robin Milner, Jachim Parrow, David Walker: A Calculus of Mobile Processes, Parts I and II; LFCS Report Series ECS-LFCS-89-85 and -86, University of Edinburgh LFCS 1989.

A. J. R. G. Milner, M. Tofte, R. Harper: The Definition of Standard ML, MIT Press, 1990.

R. Milner, J. Parrow, D. Walker: A Calculus of Mobile Processes, in Information and Computation 100, pp. 1-40, September 1992.

A. J. R. G. Milner, M. Tofte, R. Harper, D. MacQueen: The Definition of Standard ML (Revised), MIT Press, 1997.

E. Nagel and J. R. Newman: Gödel's Proof, Routledge and Kegan Paul, 1959.

Nicolas Ostler: Empires of the Word: A Language History of the World, Harper Collins 2005.

Nicolas Ostler: Ad Infinitum: A Biography of Latin, Harper Collins 2007.

Nicolas Ostler: The Last Lingua Franca: English until the Return of Babel, Allen Lane 2010.

D. L. Parnas: On the Criteria to be Used in Decomposing Systems into Modules, Communications of the ACM, 15, 12, December 1972, pp. 1053-1058; (Reprinted as Chapter 7 in Parnas 2001).

D. L. Parnas, "Software Aspects of Strategic Defense Systems", American Scientist, Vol. 73, No. 5, Sept.-Oct. 1985, pp. 432-440. (revised version of University of Victoria Report No. DCS-47-IR). (Reprinted as Chapter 6 in Parnas 2001).

David L Parnas: Software Fundamentals, Ed. D. M. Hoffman, D. M. Weiss, Addison Wesley, 2001.

Mark C. Paulk, Charles V. Weber, Bill Curtis, Mary Beth Chrissis, "Capability Maturity Model for Software, Version 1.1". *Technical Report* CMU/SEI-93-TR-024 ESC-TR-93-177. CMU/SEI, February 1993.

C. A. Petri. Concepts of Net Theory, Proc. Mathematical Foundations of Computer Science, pp. 137-166, 1973.

C. A. Petri. Concurrency, in Net theory and Applications, Lecture Notes in Computer Science 84, pp. 251-260, Springer-Verlag 1980.

K. R. Popper: Conjectures and Refutations, Routledge and Kegan Paul, 1963, revised 1972.

Laurence H. Putnam: Software Cost Estimating and Life-cycle Control: Getting the Software Numbers, IEEE Computer Society, 1980.

C. L. N. Ruggles (Ed.): Formal Methods in Standards, A Report from the BCS Working Group, Springer-Verlag/BCS 1990.

D. S. Scott. Outline of a mathematical theory of computation; Technical monograph PRG-2, Programming research Group, University of Oxford, 1971.

D. S. Scott. Data Types as Lattices. SIAM Journal of Computing, Vol. 5, pp.522-587, 1976.

D. S. Scott. Lectures on a Mathematical Theory of Computation. Technical monograph PRG-19, Programming research Group, University of Oxford, 1980.

D. S. Scott. Domains for Denotational Semantics, in LNCS 140: Proc. 9th ICALP, pp.577-613, Springer, Berlin, 1982.

Peter Sells, Lectures on Contemporary Syntactic Theories, CSLI (Center for Study of Language and Information) 1985.

Michael W. Shields. Semantics of Parallelism, Springer-Verlag London Ltd, 1997.

Herbert A. Simon, The Science of the Artificial, MIT Press, 1996.

C. Strachey: Towards a Formal Semantics, in T. B. Steel (ed.): Formal Language Description Languages, North Holland, 1966.

H. Thompson: Why Scientists are Speaking Out, in New Scientist, November 1985.

UK Computing Research Committee: http://www.ukcrc.org.uk/about/index.cfm, 2002.

Von Neumann, John, Arthur W. Burks and Herman H. Goldstine. Preliminary discussion of the logical design of an electronic computing instrument. [Princeton, N.J.: Institute for Advanced Study,] 1947.

L. Wakeman and J. Jowett, for the PIMB Association: PCTE, The Standard for Open Repositories, Prentice-Hall, 1993.

Glossary

Abstract Machine	A term used much in the B literature: an abstract machine comprises a set of data types, operations and a state consisting of a set of variables. Operations can change the values of the state variables, unlike the functions in a functional programming language.
ACL	Atlas Commercial Language: an imperative programming language used on the London Atlas machine, geared to commercial applications, dating from the 1960s. It had similar features to COBOL but a less verbose syntax.
ACM	Association for Computing Machinery: the USA professional scientific and educational computing society founded in 1947.
ACT1	A specification language based on initial semantics of universal algebras with equational axioms, often called equational algebras[1].

[1] See Ehrig and Mahr, 1985.

Ada	A comprehensive programming language commissioned by the US DoD to replace the plethora of languages previously used in defence projects. Ada was chosen from four contenders codenamed Green (the winner), Blue, Red and Yellow in 1979.
Additionality	A piece of government jargon: in a R&D project, additionality is the property of producing additional benefit to an audience wider than just is participants. Additionality is usually a requirement when seeking government funding for a project.
ADJ	A group of researchers (J. A. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright) who focussed on category theoretical interpretations of algebraic specification during the 1970s and 1980s. ADJ is a reference to "adjoint", a concept in category theory.
ADT	Abstract Data Type: the mathematical model of a data type found in computer languages. The latter can be integers, real numbers, characters, lists of data of another type, records of mixed type etc. An ADT in addition is bundled with the operations that can operate upon data of that type. An ADT is *abstract* because it makes no prescription about how the type is implemented or represented in the computer. The concept[2] is inspired by the study of universal algebra[3].
AEI	Associated Electrical Industries, see GEC.
AFFIRM	An experimental specification language based on abstract algebra; the effect of actions within a specification are defined by axioms.
AI	Artificial Intelligence: a broad category of computer science research in which attempts are made to mimic human intellectual activity. Typical areas are the understanding of natural language and logical reasoning.
ALGOL	A family of imperative programming languages, Algol58, Algol60, Algol68 and AlgolW, of which Algol60[4] was the most implemented and used. ALGOL stands for Algorithmic Language.
Alvey Directorate	The team within the DTI that managed the Alvey Programme, a UK collaborative research programme in information technology involving industry, academia and government.
ANSI	The American National Standards Institute.

[2] See Ehrig and Mahr, 1985, for a thorough account.
[3] See Cohn 1981.
[4] See Backus 1960 for the official definition of Algol60.

Applicative Programming Languages	See Functional Programming Languages.
APSE	Ada Program Support Environment; a proposed integrated set of tools to assist the development of programs in Ada.
ARPANET	US Department of Defense Advanced Research Projects Agency Network in the 1960s, the first ever email network.
Assertion	In an algorithmic programming language, a statement having no effect on the state of the program but which asserts that some proposition about the state should be true at that point in the execution.
ASTG	Advanced Software Techniques Group, a group within STC.
Atlas	A computer developed by Ferranti in conjunction with Manchester University and Plessey in the early 1960s. Four machines were manufactured, the London Atlas (see ULICS), the Manchester Atlas, the Chilton Atlas in the Atlas Computer Laboratory at AERE Harwell, and the Cambridge Titan. There were substantial differences in architecture, although the machine code was the same and enabled a good degree of program portability between them.
ATP	Advanced Technology Programme, a programme of work within the remit of the DTI in 1990.
Auto-G	A software design method based on Yourdon and DFDs.
AXES	An experimental axiomatic specification language dating from the 1970s.
B	A formal method for specifying and designing software, using set theory and logic as a modelling medium. Designed by J.R.Abrial[5], B is named after the French mathematicians who published under the pseudonym of Bourbaki.
BCL	A language proprietary to RADICS designed for writing compilers. It was an early example of a compiler-compiler, easier to use than that designed by Brooker and Morris.
BCS	The British Computer Society.

[5]See Abrial, 1996.

Bison	A compiler-writing tool that inputs a grammar written in a format upwards-compatible with yacc, and converts it to a program in an imperative language (C etc.) that can parse a script that conforms to the grammar. Bison is now available from the Free Software Foundation under the GNU conventions.
BNF	Backus Normal Form, later Backus-Naur Form. The description language devised by John Backus for the syntactic definition of *Algol58*, later extended by Peter Naur for the definition of *Algol60*.
Bootstrap	To bootstrap: a technique of writing a compiler for a language by writing the compiler in its own language. The first version of the compiler is usually very restricted and its implementation helped along by other means. Then a series of versions can progressively build up to the full implementation.
BoT	Board of Trade, a British Government Department, predecessor of the DTI and its successors.
BSI	British Standards Institution.
BT	British Telecommunications, the successor to the Post Office as the nationalised UK telephone and telecommunications supplier, now privatised and with several competitors.
BTH	British Thompson Houston, which merged with Hollerith to become the British Tabulating Machine Company, BTM. See also GEC.
BTM	The British Tabulating Machine Company, see BTH. BTM merged with Powers Samas to become ICT, q.v.
C	An imperative programming language developed in 1972 at Bell Telephone Laboratories by Dennis Ritchie. It was designed for implementing systems software, although it is of general purpose and has also been used for developing portable applications.
CAA (UK)	Civil Aviation Authority (UK).
CADES	Computer Aided Design and Evaluation of Software, an ICL CASE tool.
CASE	Computer Aided Software Engineering.
Category Theory	A branch of mathematics which generalises and studies the relationships between mathematical structures. It is useful for modelling certain phenomena of programming languages, such as type polymorphism.

CAVIAR	Computer Aided Visitor Information And Retrieval: a piece of software developed at STL for administering visitors to the company and their needs. It was developed as a case study in the use of Z, but was a program of practical use with a non-technical person providing the requirements.
CCS	Calculus of Communicating Systems: a formalism devised by Robin Milner at the University of Edinburgh for modelling dynamic systems which can consist of communicating components. It is an example of what later became known as a Process Algebra.
CCITT	**C**omité **C**onsultatif **I**nternational **T**éléphonique et **T**élégraphique, the international standards committee for telecommunications. Its parent body is the ITU with which, in 1992, it became identified and ceased to be a separate entity.
CCTA	Central Computer and Telecommunications Agency, a one-time agency of the UK Government.
CCTV	Closed Circuit Television
CEC	The Computer Engineering Centre, a group within ITT based in Brussels and later in Versailles.
CEC	The Commission of the European Communities, frequently referred to as the European Commission.
CEGB	The Central Electricity Generating Board, the UK electricity generating utility before privatisation.
CHILL	The CCITT High Level Language: a language used for telecommunications systems with similar features to Ada (q,v.) and developed about the same time.
CHILL IF	The CHILL Implementers Forum. A committee of implementers of compilers for CHILL that met to agree language features, syntax and semantics.
CICS	Customer Information Control System, a transaction server developed by IBM, mainly for the business sector. The first release was in 1969 but CICS has been developed further ever since.
CLEAR	A language based on abstract algebra for expressing models or specifications.

CMM	Capability Maturity Model: developed by the SEI at Carnegie-Mellon University, a means of determining the maturity of an organisation's capability in developing software and performing software engineering. Five levels of maturity were defined from chaotic to highly defined with procedures for assessment and improvement. The model became very popular despite very few organisations reaching the desired level 5.
CMOS	Complementary metal–oxide–semiconductor: a technology for constructing integrated circuits, patented in 1967. It uses the principle of field effect transistors, having a sandwich of a metal electrode, a metal-oxide insulator and semiconductor (a semi-metal). It has practical advantages of low noise and low power consumption.
COBOL	Common Business Oriented Language. One of the first high-level languages, contemporaneous with Fortran. COBOL was designed in 1959 by Grace Hopper, a Commander in the US Navy[6]. The first compilers were written in 1960 and portability of programs, an initial advantage of high-level languages, demonstrated that year.
CoC	Calculus of Constructions, a formal language which can express both computer programs and mathematical proofs, developed at INRIA by Thierry Coquand and Gerard Huet in the late 1980s.
COCOM	Coordinating Committee for Multilateral Export Controls. A committee of 17 member states established in 1947 during the Cold War which held a list of embargoed products considered to be of potential assistance to the Soviet Union's military capability. COCOM was disbanded in 1994, to be replaced by the considerably less proscriptive Wassenaar Arrangement.
Common LISP (CL)	A dialect of the LISP programming language, developed in 1984 and standardised by ANSI in 1994. In addition to list processing, Common LISP supports functional, procedural and object oriented programming paradigms.
Coq	A proof assistant developed at INRIA, based on CoC, the Calculus of Constructions, q.v.

[6] To paraphrase Gilbert and Sullivan, "If you stick to your computer and never go to sea, you'll soon be a Commander in the US Navy"

CORAL — Computer On-line Real-time Applications Language: a programming language developed in 1964 at RSRE and stabilised as CORAL-66 in 1966. Intended for military applications, CORAL was a high-level language in the Pascal style but with low-level facilities designed for space and time efficiency. A version, PO-CORAL, was adopted for industrial use by the UK Post Office in the 1970s.

CORE — Controlled Requirements Expression: a method of capturing and expressing requirements of a system, developed and used at British Aerospace in the early 1980s. A simple diagrammatic notation was used to show data flow within and between different system viewpoints and shown to the customer to provide a picture of the proposed system, as well as providing a basis for implementation.

CPL — Cambridge/Combined Programming Language: a programming language developed during the 1960s first at the Cambridge University Computing Laboratory, then in conjunction with the University of London Institute of Computer Science. It survived into the early 1970s. See Barron et al, 1963.

CPU — Central Processor Unit, the electronics within a computer that executes programs held n the computer's memory.

CSP — Communicating Sequential Processes: a process algebra and mathematical theory consisting of a notation and language for modelling concurrent, communicating processes., devised by Tony Hoare in 1978.

DARPA — Defence Advanced Research Projects Agency (USA).

Database — A file or files stored on a computer that consist of potentially many records of information, all of the same structure. Examples could be names, numbers and addresses in a telephone directory, or names, addresses, account numbers and balances of customers of a bank or a department store. The first research project database systems appeared in the 1960s, and became commercially available in the late 1970s.

DEC — The Digital Equipment Corporation, a US computer manufacturer, noted for the VAX (q.v.) range in the 1970s, which displayed some innovative technology.

DERA — Defence Evaluation and Research Agency: previously RSRE, now privatised as QinetiQ, q.v.

DFD	Data Flow Diagram: a graphical technique for displaying the flow of data through a system, without specifying the order of processing, devised by Larry Constantine in the 1970s. Data Flow Diagrams assist a top-down approach to design.
DoI, DTI	Department of Industry, Department of Trade and Industry, previously the Board of Trade, now (in 2009) the department of Business, Innovation and Skills (BIS): the UK government department with the objective of stimulating UK industry.
Domain Theory	A branch of mathematics comprising partially ordered sets in which such a set can include its own (partial) function space. Such partial functions include the computable functions and can therefore model certain kinds of recursive types, which cannot be modelled in more traditional set theory. The application of Domain Theory to computer science was pioneered by Dana Scott in the 1960s[7].
DP	Data Processing, the commercial application of computers to large scale information.
DPSS	A proprietary operating system for the ITT 3200 machine developed at LCT, part of ITT Europe, in the early 1970s.
Dumb terminal	A user console consisting of a keyboard and display device with little or no processing power of its own, linked to a central computer.
EATCS	European Association for Theoretical Computer Science.
EC	The European Community, comprising six countries (Belgium, France, Germany, Italy, Luxembourg, the Netherlands) at its foundation in 1957, became nine-strong when the UK joined in 1973, numbered 15 from 1995 until 2004 when ten more countries joined making the total 25. With Bulgaria and Romania arriving in 2007, the number in 2010 stands at 27.
ECC	Extended Calculus of Constructions, an extended form of CoC developed at the LFCS at Edinburgh University in the early 1990s.
ECMA	The European Computer Manufacturers' Association.
ECU	European Currency Unit, the predecessor of the Euro. It was used from March 1979 until the adoption of the Euro on 1st January 1999. There was never any hard currency for the ECU, but it was possible in most European countries, including the UK, to open a bank account in ECU. On 1st January 1999 these accounts magically turned into Euro accounts, on a one-for-one basis.

[7] See Scott 1971 et seq.

ELLA	A computer language proprietary to Praxis for programming simulations of digital electronic hardware.
ELIZA	A computer program designed in 1966 by Joseph Weizenbaum, which simulated a psychotherapist of the Rogerian school. It worked mainly by rephrasing many of the client's responses as questions and posing them back to the client. The program was named after Eliza Doolittle, the heroine in George Bernard Shaw's play Pygmalion.
Environment	Development Environment or Operating Environment: the development environment of a piece of software is the collection of computerised support tools that aid the developer to design and produce the software through its development. Examples are the APSE and IPSE. The operating environment is the characteristics and rate of arrival of the data presented to the software, the expectations placed upon its output by the technical system of which it is a part, and any other software which interacts with it.
EPSRC	The UK Engineering and Physical Sciences Research Council.
Equational Reasoning	The techniques of formal deduction in a formal system in which the axioms are expressed as equations.
ESA	The European Space Agency.
ESF	The European Software Factory: a European international endeavour to define and produce a PSE for developing and supporting software through its life-cycle.
ESPL1	Electronic Switching PL/1, a programming language devised for telecommunications proprietary to ITT in the 1970s. Although named after the language PL/1 it had little of the sophistication of the latter and was not much more advanced than an autocode. However, it was a great advance on the symbolic assembler language, which was otherwise used for telecoms applications in ITT.
ESPRIT	The European Strategic Programme for Research in Information Technology, an initiative of the European Commission.
EST	A 1984 research project in STL investigating the construction of an automated proof system that was generic, i.e. that could be parametrised by a codification of the desired logic. The intention was to start with equational reasoning and proceed to other logics. The project led on to the NIMBUS project.
ETOL	An interpreted language for testing software produced and used by the ESA in 1985.

Eureka	A European initiative, not funded through the CEC, launched in 1985, to support close-to-market R&D in all technological sectors carried out by industry, research institutions and universities. It comprises 39 national members, including those in the European Community.
FACIT	Formal Approaches to Computing and Information Technology, a series of books within Springer-Verlag London's volumes on computing, initiated in 1990.
FACJ	The Formal Aspects of Computing Journal.
FACS	Formal Aspects of Computer Science, a special interest group of the British Computer Society founded in 1978 and still flourishing at the time of writing.
FCO	Foreign and Commonwealth Office, a U.K. Government department.
FDL	Functional Description Language, an algorithmic language with provision for assertions, being an intrinsic part of PVL's SPADE proof checker.
Finite Automata	A finite automaton is a theoretical machine defined by a simple mathematical formulation. A finite automaton consists of a finite set of states, of which one is an initial state, an input alphabet consisting of a finite set of symbols, a state transition function and one or more final states. Finite automata have been proved equivalent to Turing machines, and can parse sentences belonging to a grammar defined in certain standard forms. The pioneering work was done on finite automata in the 1950s.
Finite State Machines	A formulation of finite automata conducive to implementation in imperative computer programming languages.
Flagship	A parallel computing research and development project funded by the Alvey Directorate in 1985.
FM	Formal Method(s): mathematical approaches to software and system development which support the rigorous specification, design and verification of computer systems, based on mathematical logic and set theory.
FME	Formal Methods Europe, instituted in 1987 with support from ESPRIT, but for many years now self-financing.
FORMAP	Formal Methods Applied to Protocols, an Alvey funded UK collaborative project in 1984-1985; STL, BT, GEC, ICL and IDEC took part.
ForTIA	Formal Techniques Industrial Association, a club of industrial users and suppliers of formal techniques, initiated in 2003 under the auspices of FME.

Fortran	An early high-level imperative programming language, developed by John Backus at IBM in 1953. The name is an abbreviation of Formula Translation system. Numerous versions have flourished and Fortran is still in use today, the latest version (at the time of writing) being Fortran 2008.
Functional Programming Language	In contrast to imperative programming languages, a functional language has no variables, no state and no instructions that are obeyed. A program is a collection of functions which may be defined in terms of each other, very similar to mathematical functions. The program works by applying the main function to one or more input values. For that reason FPs are also known as applicative programming languages.
*f*UZZ	A simple but effective support tool for Z, written by Mike Spivey at Oxford University in 1988.
Framework Six	The sixth in a series of "Framework" programmes, funding initiatives of the European Commission to stimulate research and development in a number of technical areas, including "Information Society" technologies. Framework Six covered the period 2002 to 2006.
Gamma	A software design technique developed by Mike Falla at Software Sciences Ltd. in the 1970s. It was a tool that could support the use of a software development method and had been used by Barclays Bank with JSD.
GCSE	General Certificate of Secondary Education. The standard collection of examinations taken by school students at about age 16 in England and Wales. It succeeded the GCE which in turn replaced the O-level exams.
GEC	The General Electric Company, a British company founded in 1886 manufacturing electrical apparatus. Through numerous mergers GEC acquired AEI, Metropolitan Vickers, BTH, Edison Swan, Hotpoint, English Electric (which included Elliott Bros.), Marconi, Plessey and many others.
GYPSY	A program verification environment developed by Donald I. Good.
HNC	Higher National Certificate, a higher education qualification in the UK, approximately equivalent to one or two years' of a university course.
HOL	Higher Order Logic, distinguished from first order logic. HOL is also the name of a theorem proving system; the homonym is deliberate.

HOPE	An experimental applicative (functional) language developed by Rod Burstall, Dave McQueen and Don Sanella at Edinburgh University in the 1970s.
Host	A computer on which software runs that assists the development of other programs, such as compilers. Contrast with target machine.
IBM 360, 370	A range of mainframe computers produced by IBM (International Business Machines) from 1964 to 1977. The success of the series may be attributed to the continuing compatibility of the numerous models in the series, achieved by using the same instruction set. IBM spent a lot of effort and money on marketing to remove the expanded form of their name from the public consciousness.
IC	Imperial College: Imperial College of Science, Technology and Medicine, London.
ICALP	International Colloquium on Automata, Languages and Programming. ICALP is an international colloquium sponsored by EATCS, which has been held annually since 1972.
ICL	International Computers Limited, a leading British computer manufacturer formed by a series of mergers from Elliotts, English Electric, International Computers and Tabulators (ICT), Leo Computers, Marconi, and others.
ICT	International Computers and Tabulators, see ICL.
IEE	The Institution of Electrical Engineers, now the IET.
IEC	International Electro-technical Commission. An international standards organisation for electro-technological products.
IET	The Institution of Engineering and Technology, the UK professional body for engineers and technologists.
IFIP	International Federation for Information Processing, established by UNESCO in 1960.
IKBS	Intelligent Knowledge Based Systems: systems based on the techniques of Artificial Intelligence (AI).
IMA	The Institute of Mathematics and its Applications.
IMI	Imperial Metal Industries, a UK engineering firm founded in 1862.

Imperative Programming Languages	Programming languages which perform actions by executing a series of coded instructions (also called commands or statements) in sequence. The canonical imperative instruction is an assignment, in which a variable is assigned a new value. The "state" of a program is the set of values held by all the accessible variables at a point in the execution.
Ina Jo	A language for formal specification and verification of software systems, developed by J. D. Guttman at the Mitre Corporation in the 1980s.
INRIA	Institut National de Recherche en Informatique et en Automatique (National Institute for Research in Computer Science and Control): a French national research institution of high repute founded in 1967.
Intel 8086	A 16-bit microprocessor chip designed by the company Intel, in the late 1970s.
Invariant	A logical condition which must remain true throughout the execution of a program loop.
IPR	Intellectual Property Rights.
IPSE	Integrated Project Support Environment; see PSE. Also used for a particular UK DTI and MoD initiative to develop an IPSE: see IPSE2.5.
IPSE2.5	A particular IPSE project within the Alvey programme. Three generations of the IPSE were envisaged: the specification of IPSE2.5 was positioned between those of the second and third generations.
ISO	International Standards Organisation
ISTAR	A project support environment developed by BT in the 1980s.
ITAB	The Information Technology Advisory Board, a committee within the DTI that oversaw funded IT projects.
ITD	Information Technology Directorate, a directorate in the DTI.
ITT 1600	A 16-bit minicomputer manufactured by ITT and used within that corporation mainly as an embedded computer in telecommunications (telephony and telex) systems in the 1970s.
ITT 3200	A 32-bit minicomputer manufactured by ITT and used within that corporation mainly as an embedded computer in telecommunications (telephony and telex) systems, but also for software development, in the 1970s and 1980s.
ITU	International Telecommunication Union, founded in 1865 and made a United Nations Agency in 1947.

INRIA	Institut National de Recherche en Informatique et en Automatique (National Institute for Research in Computing and Control); the French national research institution, which concentrates on computer science, control theory and applied mathematics. It was founded in 1967.
Java	An Object-Oriented programming language released in 1995, designed to eliminate as far as possible the capability of programs to corrupt the operating system and other essential software.
JFIT	Joint Framework for Information Technology: a UK government supported advisory body for IT with members from industry and academe.
JSD	Jackson System Development: a system development methodology developed by Michael A. Jackson and John Cameron in the 1980s. Great emphasis is placed on modelling the real world environment of the system to be developed, before specifying the system itself.[8]
JSP	Jackson Structured Programming, a structured program development method developed by Michael A. Jackson in the 1970s, based on the principle that the structure of a program should reflect the structure of the data that its processes.[9]
KDF9	A mainframe computer designed and manufactured by English Electric, in service from 1964 to 1980. Its logic was based on germanium solid state circuitry.
Lambda Calculus	In mathematical logic, a formal system that can be interpreted as a means of function definition and application, devised by Alonzo Church[10] in the late 1930s and early 1940s. It inspired the design and semantics of Algol 60[11]. Lambda Calculus (or λ-calculus) has been used as a component of the mathematical foundations of the formal semantics of programming languages.
LCD	Liquid Crystal Display.
LCF	Logic of Computable Functions: an interactive theorem prover developed by Robin Milner at Edinburgh and Stanford Universities in 1972[12].

[8]See Jackson 1983.
[9]See Jackson 1975.
[10]See Church 1941.
[11]See Landin 1965.
[12]See Milner 1972.

Legacy Code	Software dating from possibly many years in the past, that may work well and be useful, but whose details of design have long been lost. Maintaining such systems, in the absence of detailed knowledge of its design, how it works, and what are all its features is problematic, and is itself the subject of much study and research.
LFCS	The Laboratory for the Foundations of Computer Science was founded by Rod Burstall, Gordon Plotkin, Robin Milner and Matthew Hennessey in 1986. It is part of the Computer Science Department at Edinburgh University, now called the School of Informatics.
LISP	List Processing language: a programming language designed by John McCarthy in 1958. The design of LISP was influenced by Church's Lambda Calculus and has been widely used for AI. Linked lists are a principal data structure of LISP, and source programs in LISP consist of linked lists. A considerable number of dialects and developments of LISP have been produced since its inception, notably Common LISP.
LMS	The London Mathematical Society.
Logic Programming	The use of logical statements as a programming language. One of the earliest examples was the language Prolog.
LOTOS	Language Of Temporal Ordering Specification. A formal specification language designed for protocol specification in telecommunication systems, but applicable to many applications involving concurrent and temporal behaviour. LOTOS is built upon concepts from CCS, CSP and data types. Originated in 1989[13].
LPF	Logic of Partial Functions: a logic which allows undefined values in addition to the propositional values True and False.
LR	Lloyd's Register, previously Lloyd's Register of Shipping, a British organisation with charitable status, which certifies the safety of engineering systems.
LTS	A language, proprietary to STL, for defining and simulating digital electronics.
LUCOL	An autocode-level language proprietary to Lucas for engine control software, dating from the 1980s.
Macro	A textual sequence within a programming language, especially an assembly language, which stands for a defined longer sequence. Macros avoid repetition and can be parametrised.

[13] See ISO 8807:1989.

Martin Löf type theory	A theory of types based on intuitionistic logic developed by Per Martin Löf in the 1980s.
Mascot	A system design method using a graphical notation and a set of building blocks for expressing real-time system designs. It was sponsored by the MoD, had a substantial users' association and its use was mandatory in some MoD contracts. Originated in 1978.
MCHAPSE	A project support environment, based on the APSE, for software written in CHILL or Ada, developed by a consortium of telecommunications institutions including BT in the 1980s.
Mercury	A computer designed and built by Ferranti in the early 1950s. the first version used a magnetic drum memory and its electronics used valves (thermionic vacuum tubes). Subsequent versions of the machine used transistor electronics and magnetic core store random access memory, again developed in the 1950s. First customers were Manchester University, CERN in Geneva, AERE at Harwell and the UK Meteorological Office.
Metaconta L	A computer-based telephone exchange system developed by the ITT R&D laboratory LCT in Versailles in the early 1970s.
ML	Metalanguage: a functional programming language developed by Robin Milner and others at Edinburgh University in the early 1970s. It was designed for programming proof tactics for the LCF theorem prover.
MMI	Man-Machine Interfaces: perceived by the Alvey programme as an enabling technology. The interfaces between a computers and their users, these days less gender-specifically referred to as user interfaces.
MoD	The UK Ministry of Defence.
Model-based languages	A model-based specification language is one in which the data types are defined in terms of set theory and thence whose values and state variables are models of types and values in a programming language. See also Abstract Machines.
Modular One	A 16-bit mini-computer built with emitter coupled logic, which first appeared in 1969, manufactured by the British company Computer Technology Ltd.

Modula-2	An imperative programming language developed in the late 1970s by Niklaus Wirth at the Swiss Federal Institute of Technology in Zurich. An emphasis was separately compilable modules. Its predecessor was Modula, in turn based on Pascal, both of these also being developed by Wirth.
Monitors	A program language structure invented by C. A. R. Hoare and Per Brinch-Hansen, facilitating concurrent programming. Monitors are sections of code that can be executed safely by more than one execution thread through a mechanism of mutual exclusion.
Motorola 68000	A microprocessor manufactured by Motorola from 1979 to 1996 and used in embedded systems. Other manufacturers continue to produce the design using later technologies to this day.
MULE	Manchester University Language Environment, a collection of tools being developed by Manchester University to support rigorous software and language development in the 1980s.
Nassi-Shneiderman diagrams	A graphical method of representing top-down, structured program design, devised by Isaac Nassi and Ben Shneiderman in 1972.
NBG	A system of axioms for set theory proposed by John von Neumann, Paul Bernays and Kurt Gödel in the mid-1920s.
NCC	The National Computing Centre, an organisation supporting UK IT industry.
NewSpeak	A language devised by RSRE primarily for programming the Viper high reliability computer. Special features included finite types enabling compile-time bound checking and a limited form of recursion.
NIH	"Not Invented Here", a syndrome in which one regards anything invented outside one's home territory (institution, country etc.) with mistrust. It leads, for example, to firms selling to major European and north American countries setting up local national sales offices and deliberately giving the impression that the company is based in the country of targeted sales.
NIMBUS	A 1984 STL research project in constructing generic automated proof systems, successor to the EST project q.v.
NPL	The National Physical Laboratory, a UK Government research establishment.
NuPRL	A higher order proof development system originated by Joseph Bates and Robert Constable in 1979 and further developed by many others at Cornell University.

OBJ	A family of programming languages introduced by the late Joseph Goguen in 1976. OBJ languages are based on algebraic principles, in particular order-sorted algebras, enabling users to define their own abstract data types. OBJ incorporated many object-oriented ideas.
Object Oriented Design/Programming	An approach to software design that springs from Abstract Data Types, and consists of packaging together data classes and their operations to construct the "Objects" of a system. Modularity, Encapsulation, Polymorphism and Inheritance are underlying principles of OOD/OOP.
OCCAM	A concurrent programming language based on CSP – one could say an executable version of CSP – designed by David May of INMOS in conjunction with the Oxford University PRG, in 1983.
Orion	A computer designed and built by Ferranti in 1959-61, contemporary with the Atlas (q.v.) but smaller.
OS	Operating System: on a computer, the basic software that is necessary for enabling the operation of application software such as compilers, spreadsheet packages, browsers, email programs, database packages and the like. An operating system will invariably include software handlers for the peripheral devices that are attached to the computer, the keyboard, monitor, hard disc and so on. It will also include the software that enables the running of several different applications at the same time together with input/output (time sharing). On present-day PCs, the operating systems are programs such as MS Windows, Unix, Linux, Mac OS, etc. Such operating systems today also have bundled in with them applications such as web browsers and email clients, but these are not strictly part of the OS although commonly regarded as such.
Pascal	A high level imperative programming language created by Niklaus Wirth in 1970. It was based on Algol60 but designed to be simpler and more efficient, and to encourage structured programming.
PC	Personal Computer. At first called microcomputers, small computers constructed around the microprocessors that were developed in the mid-seventies, designed to be used by one person at a time, interactively. At first there were a proliferation of designs. The IBM PC became dominant in the 1980s, but competition became rife with many firms manufacturing PC clones.
PC	Process Controller: a small computer designed to be embedded in an engineering system and programmed to control an engineering process, often used in automated manufacturing systems.

PCTE — Portable Common Tools Environment: a specified collection of tools supporting software development that can be transported from one host operating system to another[14]. See PSE.

PDP11 — A minicomputer manufactured by the Digital Equipment Corporation from 1970 to the 1990s.

Pegasus — A computer developed by Ferranti, using thermionic vacuum tubes for its electronics, in 1956 (Pegasus 1) and 1959 (Pegasus 2).

PERT — Project Evaluation and Review Technique, a system using charts for determining the time resources needed for a project and the activities on its critical path.

Petri Nets — A mathematical formalism consisting of places, transitions and arcs, for describing concurrent processes, devised by Carl Adam Petri. Other approaches include process algebras (q.v.). The first documented reference appears to be in 1962 in Petri's university mathematics dissertation.

PIMB — The PCTE Interface Management Board.

Platform — The combination of hardware and software architecture that enables other software, usually applications software, to run.

PL/1 — A high level programming language devised by IBM at their Hursley laboratories in the UK in the early 1960s. It was designed for both business and scientific use and supported some structured programming concepts.

PLC — Programmable Logic Controller. This is an electronic device which is essentially a computer, typically without input or output devices, or backing storage, and with limited memory. They are normally wired into engineering systems such as controllers for industrial processes, engine management systems and the like. They are of similar computing power to a 1960-70s minicomputer, but usually occupy a single circuit board.

PLM — "Programming Language for Microprocessors", an autocode level programming language designed by David Wright's team at STL, ITT in the 1970s.

PML — Process Modelling Language: a language defined in the IPSE 2.5 project for modelling the rôles and activities in a software development.

[14] See ISO/IEC 13719-1, 1998 for the standard defining PCTE, and Wakeman and Jowett, 1993.

PO-CORAL	See CORAL.
Poplog	A multi-language software development environment, developed by the University of Sussex from 1983. It supports ProLog, Common LISP and Standard ML, among other languages. Later co-developed and distributed by SDL, Poplog is now available as an open-source system hosted by the University of Birmingham.
Portable, Portability	Software is portable if it is designed to be easily transferred across different platforms and operating systems.
Postcondition	A logical predicate which is required to be true after a statement or sequence of statements in a program has been executed.
Precondition	A logical predicate which needs to be true before the execution of a statement or sequence of statements in a program in order that the postcondition (q.v.) is satisfied. The weakest precondition (q.v.) is the weakest such predicate, and hence is the precondition which is implied by all other pre-conditions.
PRG	The Oxford University Programming Research Group founded in 1965.
Process Algebra	A general term for mathematical formulations which can define processes, where a process consists of events, including communication events enabling process to communicate with each other. Examples are CCS and CSP.
ProLog	A language for Logic Programming, (in French, Programation Logique) developed in 1972 by Alain Colmerauer of the University of Grenoble, from a collaboration with Robert Kowalski at Edinburgh University.
ProofPower	A proof development system for specifications written in HOL – Higher Order Logic. Developed originally by Roger Bishop Jones at ICL from 1989, now an open source system with further development continued since 1997 by Lemma 1 Ltd.
PSE	Project Support Environment; a general term for an integrated set of software tools for assisting the project management and development of software. See IPSE.
PSL/PSA	Problem Statement Language/Problem Statement Analyser: a computer-based toolset intended to describe system requirements and designs, developed by Daniel Teichrow at the University of Michigan, through the 1970s. PSL/PSA is still in use and being developed.

PVL	Program Validation Limited, a UK firm founded by Bernard Carré and dedicated to providing facilities for program proving. PVL later merged with Praxis.
QinetiQ	A UK technology company with a tradition of supplying to UK and US defence and government. It arose from the privatisation of DERA in 2001.
RAISE	Rigorous Approach to Industrial Software Engineering, a formal method with a set of tools developed in a ESPRIT funded project led by Dines Bjørner. The RAISE specification language (RSL) contains elements of VDM-SL and Process Algebra.
RAM	Random Access Memory: a hardware memory medium in which the addressing and access to contents is equally direct for all content addresses. In practice this means that the content is accessible by direct electronic means, without recourse to mechanical or other searching, so no rotating medium etc. The memories mounted on the motherboards of PCs, and USB sticks are examples. By contrast, hard and floppy discs are not RAM, not being "Random" access.
Refinement	The process and result of deriving from a formal description a more concrete one which satisfies all its properties and prescriptions.
Reverse Engineering	The process of deriving a design or a specification from an implementation of some software, i.e. from the code. To put it crudely, one takes some program code and discovers (with computer-aided assistance) what it does, how it works, and possibly what it is for. It is useful for analysing legacy code (q.v.). Reverse engineering can also be applied to electronic and electromechanical systems, and can be used in less legitimate contexts such as industrial and military espionage for examining stolen or captured equipment and designs.
ROAME	A DTI-specific acronym for the case made for funding support, an activity or an initiative, consisting of Rationale, Objectives, Appraisal, Monitoring and Exploitation.
RRE	Royal Radar Establishment, a research arm of the British army regiment Royal Signals; see RSRE.
RSL	The RAISE (q.v.) Specification Language.

RSRE	Royal Signals Research Establishment, later RRE, DERA, finally privatised as QineticQ.
SADT	Structured Analysis and Design Technique: a graphical notation for describing software systems, developed by Douglas T. Ross at SofTech Inc. in 1969 – 1973.
SALT	Speech and Language Technology, a club funded by the DTI and SERC for institutions with an interest and active in the topic.
SDI	Strategic Defence Initiative. Launched in 1983 by President Ronald Reagan, SDI was a research and development project whose ultimate aim was to build a network of satellites that would detect incoming hostile intercontinental missiles and shoot them down with laser weapons. The initiative was highly controversial from many points of view: technological feasibility, safety, policy, ethics and the wisdom of UK participation. See Ennals 1986.
SDL	System Designers Limited, a British software and systems development company, subsequently merging with Scicon to become SD-Scicon, now owned by EDS.
SDL	System Design Language: a language for expressing the design of telecommunication systems, based on finite state machines, defined and standardised by the CCITT in the 1970s.
SDSS	A proprietary (ITT) software development platform hosted on the IBM 370 and having the ITT3200 as target machine, first released in 1978.
SEI	Software Engineering Institute, at Carnegie-Mellon University, who were responsible for developing the Capability Maturity Model (q.v.).
SEMA	The SEMA Group plc was a joint British-French IT services company formed in 1988 by the merger of CAP (UK) and Sema-Metra (France). After a succession of acquisitions Sema Group plc was itself acquired by Slumberger in 2001, to be mostly sold on again in 2004.
Semaphore	A mechanism for enabling parallel process to access shared data or other resources in a controlled way, without mutual interference. Semaphores were invented by Edsger W. Dijkstra in 1965 and have been widely used in operating systems since.
SERC	The Science and Education Research Council, previously the Science Research Council (SRC), now the EPSRC.

SFI	Support For Innovation: a guiding principle in the DTI's Advanced Technology Programme (ATP).
SIG	Special Interest Group.
Simula67	A computer language designed for simulation problems by Ole-Johan Dahl in 1967, Simula67 is a superset of Algol60 with object-oriented features. It is considered to be the first object-oriented language.
SLANG	Simulation Language: a language for simulating analogue computer programs, designed by ULACS for Elliott Bros. in 1968.
Smalltalk	An Object Oriented programming language, developed in the 1970s at Xerox PARC and released publicly in 1980.
SMART	Small firms merit Awards for Research and Technology: an annual competition for single UK companies run by the DTI.
SME	Small and Medium sized Enterprises. Both the DTI and the EC use the term, each with specific but slightly differing definitions in terms of number of employees, turnover etc.
SML	Standard ML (q.v.): a further development of ML, with contributions from several academic institutions. The first definition was published in 1990, revised in 1997[15].
Spade	A static analysis tool, developed by PVL, which analyses a program without running it, and detects certain anomalies such as unreachable code, variables not initialised and so on. See also Spark.
Spark	A program analysis tool, developed by PVL, which, in addition to the facilities of Spade, carries out proofs of correctness and facilitates correctness by construction.
SRC	The UK Science Research Council, now the EPSRC.
SSADM	Structured Systems Analysis and Design Method, an approach to analysis and design of information systems developed in the early 1980s by the UK CCTA.
Star Wars	A popular name for the Strategic Defence Initiative, SDI, q.v.
State Diagrams	See State Transition Diagrams.
State Transition Diagrams	A graphical representation of a finite state machine, which defines the abstract behaviour of a system. State Transition Diagrams have been attributed to Taylor Booth[16]

[15] See Milner et al, 1990 and 1997.
[16] See Taylor Booth, 1967.

STC	Standard Telephones and Cables, a leading UK telephony company.
Steelman	See Strawman.
STL	Standard Telecommunications Laboratories, a research laboratory of ITT in Harlow, Essex. STL was part of STC, Standard Telephones and Cables.
Strawman	In commissioning the Ada language definition, the US DoD issued a series of documents outlining the requirements for the language. A straw-man is a common term denoting an initial suggestion put for to be criticised and to invite further ideas; also known as an aunt Sally. In developing the Ada requirements, the series of requirements documents were named to reflect their increasing solidity: Strawman (issued April 1975), Woodenman (August 1975), Tinman (January 1976), Ironman (January 1977, revised July 1977), Steelman (June 1978), Pebbleman (July 1978, revised January 1979), Stoneman (February 1980).
SQL	Structured Query Language, a language used to query a database. The first commercial versions of SQL were released by Oracle and IBM in 1979. By 1986 the American National Standards Institute adopted SQL as a standard and ISO followed suit a year later.
System X	The first public digital telephone exchange system in the UK, first installed as a local exchange in Woodbridge, Suffolk in 1981.
S3	A systems programming language proprietary to ICL, dating from the early 1970s, designed for writing operating systems, language compilers and other applications packages, S3 was a subset of ALGOL68, and had a number of its characteristic features such as reference variables: variables which could hold the “abstract address” or information leading to the whereabouts of another variable or piece of data.
Target	A target machine is a computer on which the end product software runs. For embedded software, it will be the machine residing in the engineering system that the software drives or monitors.

Temporal Logic	A logic which can accommodate statements with a time-dependent element, such as "...will be until...", "...is always...", "...will eventually be...". Research into temporal logic in a computer science context started in the 1960s, with notable contributions from Amir Pnueli and later, Willem-P. de Roever.
TickIT	A scheme for certifying software quality set up by the DTI IT division's Software Quality Unit.
Titan	See Atlas.
Transputer	A microprocessor architecture developed in the 1960s by Inmos, a UK electronics company, which was designed to support parallel computation.
Typed Lambda Calculus	A development of Lambda Calculus (q.v.) enabling types to be ascribed to λ-expressions. There are a number of varieties of typed lambda calculus, which can be regarded as extensions of the simply typed lambda calculus; on the other hand, the latter can be regarded as a special case of typed lambda calculus with only one type. Typed lambda calculi are the foundational mathematical underpinning of functional programming languages such as ML (q.v.).
Type Theory	The mathematical theory which can form a model of the data types that one can use in programming languages, especially where the language allows the programmers to define their own types. Type theory has a wide ranging mathematical pedigree dating back to the beginning of the twentieth century.
ULACS	The University of London Atlas Computing Service. A company set up by London University to use 50% of the London Atlas computer time to generate commercial income to recover the initial investment. It was in effect a computer bureau and a software house dedicated to the commercial exploitation of the Atlas.
ULICS	The University of London Institute of Computer Science. Initially, the institute was independent of any specific college within London University, and was centred on the use of the London Atlas machine (q.v.).
UMIST	University of Manchester Institute of Science and Technology.

UML	Unified Modelling Language: a language for modelling processes and systems based on Object Oriented concepts and defined by the Object Management Group, a consortium for setting standards in Object Oriented software engineering. UML is subject to an international standard[17].
Uncle	Some SERC-funded academic projects had an industrial "Uncle", an individual from industry who would visit the project at intervals, typically every three months, to provide industrial input and try to keep the project of ultimate practical utility, even if that was long-term. This was a very light form of academic-industrial collaboration, but it gave the academic partner some ratification for government funding.
Unification	A technique used in automated theorem proving and automated deduction, in which two terms are made equal or unified by means of a syntactic substitution of component variables to other terms.
UNIX	An operating system for personal computers developed originally between 1969 and 1973 at Bell Labs, AT&T, the later versions written in C, facilitating its portability. UNIX has since been adopted and further developed by many other academic and industrial institutions and is a serious rival to IBM's proprietary operating systems for PCs and the various versions of MS Windows.
VAX	A family of computers developed by DEC in the 1970s, which used a 32-bit word length and instruction set. VAX was originally an acronym for Virtual Address Extension.
VAX11/780	The first model in the VAX computer series, q.v.
VDL	The Vienna Definition Language: a language developed at IBM's research laboratory in Vienna for formulating the semantics of PL/1, dating from the early 1970s.
VDM	The Vienna Development Method, developed at the IBM Vienna Laboratories during the late 1970s. Defining data types through the elements of set theory and operations by means of pre and post-conditions and logic, it assists the development of software through proof by construction.
VDM-SL	The VDM Specification Language: the language used in VDM and standardised in 1996 as BS ISO/IEC 13817-1:1996.

[17] See ISO/IEC 19501:2005 Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2.

VIP — VDM Interfaces to PCTE, an ESPRIT project for formally defining those interfaces.

VLSI — Very Large Scale Integration: The technique of etching thousands or millions of transistors and other semiconductor devices onto a single semiconductor wafer, known as a chip. The technique was initiated in the 1970s and has developed apace and increased dramatically in miniaturisation ever since.

VME — An operating system proprietary to ICL developed in the 1970s for the 2900 computer series.

Weakest Precondition — A concept invented by E. W. Dijkstra in 1975. If one has a machine with state *P* (see Abstract Machine in this glossary), and a post-condition *R* which one desires to hold after the execution of *P*, there may be a set of preconditions which, if satisfied before the execution of *P*, will guarantee that *R* will hold. These are called preconditions and the weakest of them, that is the one which is implied by all the others (and which therefore holds for all values of the state for which the others hold), is called the weakest precondition and is denoted wp(*P*,*R*). Dijkstra gave formulae for the weakest preconditions of the normal statements in a traditional imperative programming language, and rules for combining them into compound statements and programs[18]. This led to a method for proving programs correct given a specification of the required pre and postcondition.

wff — Well-Formed Formula: in a formal language a wff is an allowable term of the language. For example, in a language of algebraic expressions, a wff might include constants, variables and expressions containing brackets, operators and other wffs.

Word — A word-processing package produced by Microsoft. First issued in 1983, many versions have been and continue to be released.

Wordwright — A proprietary early (1973) word processor, consisting of a personal computer dedicated solely to word processing, and its resident word processing software package. WordWright is also the name of a more recent (last updated 1999) free downloadable word processing package, but it is not clear whether there is a historic or commercial link between the two.

Wordwise — A word processing software package for the BBC Microcomputer dating from the early 1970s. (The term is also now the name of a patented predictive text software product used in mobile phones).

[18] See Dijkstra 1975, 1976.

Workstation	A more powerful personal computer designed for engineering applications, with greater processing power and graphics.
X.25 protocol	A packet switching protocol standardised by the CCITT in 1976, defined in a publication called the “Orange Book”.
YACC	“Yet Another Compiler-Compiler”: a compiler-compiler (parser-generator) with a defined format for expressing computer language syntax.
Yourdon	A flavour of structured programming, as promulgated by Edward Yourdon.
Z	A formal notation for specifying the functions of a program, based on ZF set theory and logic, developed at the University of Oxford by J-R Abrial and others in 1977, standardised in 2002 as BS ISO/IEC 13568:2002.
ZF	An axiom system for set theory developed by Ernst Zemelo in 1901 and extended by Abraham Fraenkel in 1922.
Z8000, Z8002	The Z8000 was a 16-bit microprocessor manufactured by Zilog from 1979 to the mid-nineties. The Z8002 was a smaller memory version. The Z8000 was used as the CPU for many popular desk-top personal computers of that era.

w