Quaestiones Informaticae

Vol. 3 No. 1 February, 1984

Quaestiones Informaticae

An official publication of the Computer Society of South Africa and of the South African Institute of Computer Scientists

'n Amptelike tydskrif van die Rekenaarvereniging van Suid-Afrika en van die Suid-Afrikaanse Instituut van Rekenaarwetenskaplikes

Editor: Prof G. Wiechers
Department of Computer Science and Information Systems
University of South Africa
P.O. Box 392, Pretoria 0001

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R6	$7	£3,0
Institutions	R12	$14	£6,0

Circulation manager
Mr E Anderssen
Department of Computer Science
Rand Afrikaanse Universiteit
P O Box 524
Johannesburg 2000
Tel.: (011) 726-5000

Quaestiones Informaticae is prepared for publication by Thomson Publications South Africa (Pty) Ltd for the Computer Society of South Africa and the South African Institute of Computer Scientists.

Concurrency: An Easier Way to Program

C.S.M. Mueller
University of the Witwatersrand, Johannesburg

Abstract
Why concurrency in itself should be seen as a method of programming, is discussed. Current approaches to concurrent programming are evaluated in terms of ease of programming. The properties required to decompose the program functionally, using concurrency, are then developed.

Introduction

This paper does not profess to be the first to suggest that concurrency can aid programming [11, 17]. Some languages like data flow [2] and functional programming [13] languages have already introduced concurrency and experience with them seem to indicate that it is an advantage for the programmer [11]. However with these languages the improvement was a biproduct of some other objective such as trying to produce faster programs. This paper advocates that concurrency in itself should be seen as desirable within a programming language from the programming point of view and for no other reason.

Background

The idea of concurrency has been around for some time. Dijkstra [8,9] in his "guard" statement became aware that the order in which statements get executed is not always important and that it may be better to leave the order unspecified. Other areas of the program have also been looked at, such as the evaluation of a condition for the "if" statement [18] and the lazy i/o [19]. What is significant about the above examples is that the primary purpose was to improve programming; however these ideas relate only to certain very localized aspects of the program.

Other very early work was that done in compiler optimization in evaluating data dependencies [3,7,12]. This work showed that in studying data dependencies one could reorder the statements without altering the meaning of the program, illustrating that strict ordering is not required in a program which implies that programs have inherent concurrent properties.

More recent work which is based on the data flow analysis is that of the data flow languages [10]. The primary objective was to break away from the von Neumann style of architecture to move into "high-speed" computing [2]. In designing the data flow languages, the language designers incorporated functional programming aspects. Similarly functional programming is moving towards parallelism [23]. However, not for the purpose of making them better from a programming point of view, but rather to make functional programming viable in terms of execution speeds.

Concurrency

In this paper the intention is to extend the idea of concurrency to the limit. This has already been taken a long way with data flow languages. In order for this idea to be carried further it is important to consider how concurrency is identified by data analysis. Any statement within a program can proceed as long as the data used by it is defined and ready to use. Thus, at any point in time, a number of statements may be ready to proceed. These statements can then all proceed in parallel. Data analysis is able to determine if the data for the statement is defined or not. The analysis is greatly facilitated by applicative programming [13] where a variable cannot have the value it is defined with, changed. Using the more functional programming language concept one is able to construct a direct acyclic graph [3] which determines the data dependencies. Once the data dependencies have been established, then parts which can run concurrently can be established.

Turner [23] has some intersting definitions which apply to concurrency. A program in which the order of execution of the statements is fully specified he calles TOTALLY ORDERED — this being the case in most of the more conventional languages. The program in which the order is not fully specified he calls PARTIALLY ORDERED. The partial ordering implies some concurrency. This concurrency is made use of in the new approaches of data flow and some functional programming. What this paper wishes to explore, is whether the programmer has to explicitly define ANY ordering in his program and how can it aid him in the task of programming.

Functional Relationship

One of the biggest problems with programming languages was raised some years back, one to which no one as yet seems to have come up with the answer. Stevens et al [22] analysed the properties that a program should have. In particular they looked at what properties statements closed together should have, such as in a procedure. They come up with the fairly obvious answer that they should be closely related functionally. This is not feasible in a totally ordered program because the position of the statements is determined by the order of execution of the statements. Two statements following one another will indicate the order in which they will execute. Therefore the relationship between them will be one of time (i.e. temporal). In most languages the need to specify the order overrides the functional relationship between statements.

For a number of years there have been lobbies for structured programming and functional decomposition into modules [5]. In new languages one attempts to overcome this problem with packages in languages such as Ada [24]. These approaches are fighting a losing battle until they can place the functional requirements above temporal ones.

A valid question to put at this point is, what is wrong with the languages that do allow partial ordering? While accepting that they are a step in the right direction, it is maintained that they still take the form of the totally ordered language. Their intentions are not to give the programmer a different perspective of his program. They still present him with a very procedural-like language which requires him to think in a totally ordered way (see figure 1). The compiler then determines the partial ordering and makes use of it. The idea put forward here is that this partial ordering should be fully exploited in the language to help the programmer express his program.

If the programmer is not concerned about the ordering he can concentrate on writing truly functional decomposed programs — one in which the functional considerations take precedence over all other aspects.

How?

The problem is how the programmer can avoid specifying the order. The von Neumann architecture has influenced our way of thinking [4]. In order to break away from this line of thought, it is very useful to look at what else is being done in this line. A new area of research is that into program specification [6]. Looking at program specifications one finds that the order of the statements is not critical. Why should there not be a middle ground between specification and programming?

The program defines two aspects which one does not find in the specification. These are the algorithms to be used and the second is the order in which computations must take place.

If the order is ignored that still leaves the algorithm to be stated. The most important aspect of the algorithm is the way in which the data is manipulated. This suggests that it may be possible to define the algorithm by specifying the data and how the domain gets mapped onto this data.

Support for Arguments

People who have worked with parallelism in programs may be very sceptical about the idea that concurrency can aid the programmer. The typical program written in a conventional implicative language is not easy to understand. Even in the latest implicative languages like Ada [24], one finds that even to describe how to write a concurrent program is not simple. It is the conventional implicative language that makes programming concurrent programs complex. The problem is that the programmer is forced to deal with concurrency using a tool designed for sequential programming.

Novice Programmes

Amongst some first year students a natural tendency to think concurrently was observed. The students had a programming assignment to find the average number of letters per word in a sentence. A number of these students' immediate response was to try to define two procedures: one to find the number of words in the sentence; the other to find the number of letters in the sentence. The problem was one of making the students think sequentially. Even after having pointed out that they should scan the input data only once, some still had difficulty in understanding why they could not use two procedures to do the two functionally independent tasks. The only reason which prevented them from doing so is the shortcoming of the language they were using. They could grasp the concurrency correctly. Conceptually it is not difficult to understand that these two tasks can proceed concurrently and that they are both dependent on the input character becoming available before they can proceed.

Instead, what do these new novice programmers have to do? Immediately they must start to violate the concept of structured design. They now must deal with two different functions in one module. The reason for this is the language they used, needs them to fully define the order in which the statements get executed. Thus the programmer finds himself having to decide what the effects are after having read in a character.

The more experienced programmer though has now had this concurrent way of thinking stamped out of him. He has been trained to think in the way the von Neumann architecture executes his program. His methods of programming have taught him to check his program by simulating what the computer does. Thus the idea of not being able to determine the way in which his statements will execute is a nightmare. This fear is a reflection on the tools he uses. Better methods must be devised for program validation.

UNIX

Although UNIX is an operating system, there are some lessons that can be learnt from it in terms of programming. There have been claims that the UNIX operating system facilitates program development [17]. Studying why this is so illustrates that concurrency aids programming.

The operating system has a very interesting concept called a pipe. Experience has shown that it is often easier to construct a pipe using a number of existing programs rather than write a new program [17]. An example given below by Kernighan illustrates the point very well. The purpose of the pipe is to sort words according to their endings.

```
reverse < dictionary | sort | reverse > rhymicdictionary
```

The pipe from the programmer's point of view can be considered to consist of three concurrent parts: reversing the words in the dictionary; sorting the reversed words and then reversing them back again. Unlike an implicative language the programmer is not forced to specify the full order in which the operations are performed. Whether the words are all reversed first, before the sort starts or not is no concern of the programmer. In the same way when the process of reversing the sorted lines takes place is also not defined. The two extremes are: each process is completed before the next process starts and the three processes run in parallel.

At first glance at the example above, it looks as if the programmer is explicitly defining some order. The three processes are placed in order of the sequential processing of the pipe. This explicit ordering is only required for specifying the flow of the data. The need for explicit ordering could be done away by defining the pipe as follows:

```
sort    < unsorted    > sorted
reverse < dictionary  > unsorted
reverse < sorted      > rhymicdictionary
```

The partial ordering can now be established from the data dependencies. Thus even the partial ordering need not be explicitly stated if the minor modification is made to the way in which the pipe is defined. It is the lack of the need to explicitly order the task that is being put forward here as the reason for the UNIX pipe being an easier method of program development.

In the example one finds that the programmer is able to break up the program very conveniently along functional criteria. He does not need to allow temporal considerations to interfere with the functionality of the pipe. The result is that there is a much weaker coupling between these components. The effect of this is considerable.

This ability to be able to decompose the pipe along more functional criteria enables the goals of modularity to be further realised. Since the links between modules is weaker the programmer is able to understand a component in isolation more easily. In the example one sees this aspect clearly. Reversing of the dictionary words is completely separated from how it is sorted. These two components require to know very little about one another. The only knowledge required is the data format. This surely is what making programming easier is all about [14].

The other major advantage that the weakening of the coupling achieves is to be able to maintain the program or pipe. The first fact is to be able to locate where the change needs to be made. This is made very easy as the programmer knows which functions require modification and he is immediately able to home in on the component. The other major problem with maintenance is that a change to one component usually has implications in the other components. Since the only dependency on each other is the data, only a change to the data structure has implications outside of the component.

Limitations of the Pipe

Before the impression is given that the pipe is the total answer to the program development, a few weaknesses of the pipe need to be highlighted. Further on in the paper how to overcome and extend these ideas will be addressed.

The biggest limitation is that of only allowing one input and one output per program which is passed via the pipe. This limitation means that a program cannot directly communicate with any other program except its predecessor and its successor in the pipe. Thus if a component of the pipe generates data for another component other than its successor, it must be passed via this successor. This immediately requires the programmer to introduce logic into a component where it does not belong functionally. This violates the biggest advantage of the pipe, of being able to functionally decompose the pipe.

Another problem is the power of the pipe. Information cannot be looped back into a pipe. The output of a later component cannot form the input to an earlier component within the pipe. This disallows finding the power of a component.

The pipe is really a function of the operating system and as such cannot fully overcome all the problems of program development. The programmer is still forced to write the low

level components using full ordering in usually an implicative language. This has the unfortunate consequence that the programmer must think in a very ordered sequential non-concurrent way when designing the components. However when it comes to putting the components together, a completely different approach is used. These two ways of thinking places an extra burden on the programmer.

Evaluation of Data Flow Languages

Unfortunately there has not been too much experience in this new area. It is very much in the experimental stage. All that one can base one's ideas on are the personal experience of people working in the area. In the case of this paper the views of Th. Lalive d'Epinay have been used [11].

A summary of some of the advantages of the language are
— no side effects
— no risk in having all values globally known
— facilitates incremental construction of software systen s
— facilitates extension of running system software
— identification of faults

Although some of these advantages can be credited to the benefits derived from an applicative language, the rest are due to being able to partially order the program. The interesting point to note is that although authors in this area are aware of partial ordering (or data dependencies) and of possible improved programming advantages they do not relate these two together. This fact is illustrated by these languages not moving away dramatically from the procedural type of format (see below example [12,21]).

```
fac = [n where n> = 1]
  {n=1 : 1, OTHERWISE : n*fac(n-1)}
  a=b+c
  WITH
    b=fac(3)
    c=15
  WEND
```

Figure 1

Extending the Unix Pipe Concept to Programming

There is a very interesting observation to make about the UNIX pipe which suggests a method of extending the ideas of the pipe to programming. Consider the role of the program and the files within the pipe. The programs can be regarded as the definition of a mapping which maps the input file (domain) onto the output file (range). A file is nothing other than a data structure. Thus the pipe can be considered as a set of mappings onto data structures. This concept can now be extended to a program and define the program as a set of mappings onto data structures.

Each of these mappings within the program would have the same properties as that of the program in the pipe. The result of this is that the program would consist of a number of concurrent mappings. This means that all the advantages of the pipe now apply within the program.

The idea is that the design of the program is based on the data structures used. This idea has already been put forward by Jackson [16]. He maintained that the structure of the data should determine the structure of the program. However his method requires the merging of all the data structures into one which has been shown to have major limitations [15]. The flaw in his approach was trying to merge the data structures. A program consists of many data structures and should reflect this. In order to be able to do this, the language requires the concept of concurrency as shown by Schach [20].

The next point to consider is how the mapping is defined. The ideal way would be in the form of a specification. The new data item could be defined in terms of the domain in very much the same way as a component in the pipe is defined. For example the output file of the first component is defined as the dictionary with the words reversed.

Conclusion

Concurrency introduces a new way in which to perceive programming. It enables a method of program production that can become truly functional: something that all the programming methodologies have been striving for. This new concept releases the programmer from much of the complexities of programming by relying on the system to coordinate between parts of the program. This could be a breakthrough in dramatically improving productivity of programmers, thus enabling the industry to meet the ever increasing demands placed on it.

Bibliographical References

[1] Ackerman, W.B. Data Flow Languages. In: *Computer,* Volume 15, Number 2, 1982.

[2] Agerwala, T. and Arvind. Data Flow Systems. In: *Computer,* Volume 15, Number 2, 1982.

[3] Aho, A.V. and Ullman, J.P. Principles of Compiler Design. *Addison-Wesley,* 1979.

[4] Backus, J. Can Programming be Liberated from the von Neumann Style? A functional Style and its Algebra of Programs. In: *Communications of the ACM,* Volume 21, Number 8, 1978.

[5] Bergland, G.D. A Guided Tour of Program Design Methodologies. In: *Computer,* Volume 14, Number 10, 1981.

[6] Cleaveland, J.C. Mathematical Specification. In: *Sigplan Notices,* Volume 15, Number 12, 1980.

[7] Davis, A.L. and Keller, R.M. Data Flow Program Graphs. In: *Computer,* Volume 15, Number 2, 1982.

[8] Dijkstra, E.W. A Discipline of Programming. *Prentice-Hall, 1979.*

[9] Dijkstra, E.W. Guarded Commands, Nondeterminacy and Formal Derivations of Programs. In: *Communications of the ACM,* Volume 18, Number 8, 1975.

[10] Gajski, D.D. et al. A Second Opinion on Data Flow Machines and Languages. In: *Computer,* Volume 15, Number 2, 1982.

[11] Guth, J. and Lalive d'Epinay, Th. The Distribution Data Flow Aspect of Industrial Computer Systems.

[12] Gustafson, D.A. Control Flow, Data Flow and Data Independence. In: *Sigplan Notices,* Volume 16, Number 10, 1981.

[13] Henderson, P. Functional Programming. Application and Implementation. *Prentice-Hall,* 1980.

[14] Hoare, C.A.R. The Emperor's Old Clothes. In: *Communications of the ACM,* Volume 24, Number 2, 1981.

[15] Hughes, J.M. Formalization and Explication of Michael Jackson's Method of Program Design. In: *Software Practice and Experience,* Volume 9, Number 3, 1979.

[16] Jackson, M. The Principles of Program Design. *Academic Press,* 1975.

[17] Kernighan, B.W. et al. The UNIX Programming Environment. In: *Software Practice and Experience,* Volume 9, Number 1, 1979.

[18] Meek, B. Serial Attitudes, Parallel Attitudes. In: *SigplanNotices,* Volume 15, Number 6, 1980.

[19] Perkins, Lazy I/O is not the Answer. In: *Sigplan Notices,* Volume 16, Number 4, 1981.

[20] Schach, S.R. Jackson Structured Programming. *2nd South African Computer Symposium.* October 1981.

[21] Sharp, J. Data Oriented Program Design. In: *Sigplan Notices,* Volume 15, Number 9, 1980.

[22] Stevens, W.P. et al. Structured Design. In: *IBM System J.,* Number 2, 1974.

[23] Turner, D.A. Recursion Equations as a Programming Language. In: *Functional Programming and its Applications — an Advanced Course.* Edited by Darlington, J. et al., Cambridge University Press, 1982.

[24] United States of America — Department of Defence. Ada Programming Language, January 1983.

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be submitted in triplicate to: Prof. G. Wiechers at:

Department of Computer Science
University of South Africa
P.O. Box 392
Pretoria 0001
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae

Contents/Inhoud

Very High Speed Graphics Work-Station* 1
J Jablonski and H J Dijkman

Documenting a Database System with a Database*.....5
D A Hunter

Program Design as the Tool of Preference in Computer Literacy Education, Experience, Incentives, Implications*.....9
J M Richfield

A Virtual Multiprocessor Computer Design*.....15
T Turton

A Context Sensitive Metalanguage for Intelligent Editors*.....21
S M Kaplan

Concurrency: An Easier Way to Program*..... 25
C S M Mueller

*Presented at the Third South African Computer Symposium held on 14th to 16th September, 1983.