Abstract:

The project described by this document involves the investigation of a new form of general purpose MIMD (multi-instruction-multi-data stream) computer architecture and programming language, in which programs are regarded as implicitly parallel and serialism has to be indicated explicitly. This investigation centres on two forms of program representation, the machine level "generalised control flow" (GCF) computer organisation and the language level "structured directed graph" (SDG) user programming language. The document illustrates these two forms of program representation by presenting a description of a GCF computer architecture and a SDG programming language. A status report and objectives for the project are also briefly presented.

The Design of
Highly Concurrent Computing Systems

By

Philip C. Treleaven, Edward P. Farrell, Noordin Ghani,
Simon B. Jones, Brian Handell and Peter J. Smith

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 126
July, 1978

© 1978 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne,
Computing Laboratory, Claremont Tower, Claremont Road,
Newcastle upon Tyne, NE1 7RU, England.
The design of highly concurrent computing systems.

[By] Philip C. Treleaven [and others].

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1978.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 126.)

The project described by this document involves the investigation of a new form of general purpose MIMD (multi-instruction-multi-data stream) computer architecture and programming language, in which programs are regarded as implicitly parallel and serialism has to be indicated explicitly. This investigation centres on two forms of program representation, the machine level "generalised control flow" (GCF) computer organisation and the language level "structured directed graph" (SDG) user programming language. The document illustrates these two forms of program representation by presenting a description of a GCF computer architecture and a SDG programming language. A status report and objectives for the project are also briefly presented.

About the author

All the authors of this report are working in the Computing Laboratory of the University of Newcastle upon Tyne.
1. INTRODUCTION

Conventional programs, even on computers which are capable of concurrent activity, are implicitly sequential, and require any desired concurrency to be indicated explicitly. We believe, like our colleagues working in the area of data flow computation, in the potential value of the opposite approach, i.e. of programs which are to be regarded as implicitly concurrent, and whose sequentiality has to be indicated explicitly. However it would be quite misleading to think of our work as being solely within the realm of data flow computing; rather it is based on what we term "generalised control flow" architectures and "structured directed graph" program representations. In fact while most work on data flow can be seen as having the Computational Graph model [1] of Karp and Miller as its theoretical basis, our work is related to the Parallel Program Schemata model [2] which these workers developed later.

This document presents the machine level "generalised control flow" organisation and the user language level "structured directed graph" program representation which we are using to investigate the design of highly concurrent general purpose computing systems. Our primary objective is to identify programming language structures and mechanisms and a corresponding "multi-instruction-multi-data" stream (MIMD) processor architecture that support concurrent computation. The desire to design such MIMD computing systems, which can utilise a high degree of concurrency, is motivated by at least two interlinked requirements: firstly, the need to increase the performance of computers, and secondly the wish to obtain a more efficient utilisation of system resources. In the former case, for example, there are still problems such as wind tunnel and weather forecasting models that cannot be effectively simulated by computer. The resulting class of concurrent systems would overcome problems such as these by being able to obtain their workload either by executing a number of distinct programs concurrently or by utilising the natural structure of individual highly concurrent programs. It is also hoped that these systems will be inherently more reliable because of their replicated resources, and that their structure, particularly that of the software, will facilitate correctness proofs.

Consideration of our previous design work [3-6], and of what we view as the inadequacies of current MIMD computing systems has led us to attempt to synthesise what we believe to be the better features of control flow and data flow computing systems. Thus, for example, we have adopted the concepts of storage and the use of additional sequencing constraints (i.e. flows of control) inherent in control flow systems and combined these with concepts such as "activation by availability" from data flow. This approach has resulted in two forms of program representation, namely a machine level embodied in the MIMD computer architecture, which we refer to as the "generalised control flow" organisation, and a new style of high-level concurrent programming language, based on what we have called the "structured directed graph" representation. We are using the machine level and language level representation of computation as the basis of an empirical study of both the computer architectures and programming languages for MIMD computing systems. This document describes the generalised control flow organisation and how it can be used to support a spectrum of program representations which includes conventional control flow and data flow as special cases.
Subsequent sections discuss a possible implementation of such a generalised control flow computer, the structured directed graph representation, and the user language that this representation has led us to propose. A final brief section indicates how the generalised control flow architecture might be extended to provide a more direct implementation of our proposed user language.
2. GENERALISED CONTROL FLOW ORGANISATION

The "generalised control flow" (GCF) organisation is, at least in the abstract, somewhat conventional in appearance consisting of (i) memory locations that are used to store data, (ii) data instructions that read and write data in the memory, and (iii) control instructions which synchronise the order in which the data instructions of a program are executed. The major difference, in simple terms, between the conventional control flow organisation, with its sequential flow of control, and the proposed organisation is that control instructions in the latter may be viewed as forming a directed control graph. This is illustrated by the program fragment for complex multiply in Figure 1. Each control instruction has a sequence of data instructions associated with it and is responsible for synchronising the execution of these data instructions. In terms of the control graph, a control instruction is activated by the availability of partial controls (i.e. control tokens) on its input arcs. Activation of a control instruction causes the activation of the associated data instructions, which read their input data from memory, perform the data functions and write their results back to memory. Upon completion, a data instruction either passes control to the next data instruction in the sequence or, if it is the last in the sequence, returns control to the control instruction, which then terminates by outputting partial controls that signal the availability of the results.

It is the responsibility of the programming language compilers for a computing system based on the GCF organisation to optimise the style of control and data instructions that are generated, depending both on the resources of the target computer and on the style of language being translated. For example, for a computer of a limited store size, a compiler for a conventional programming language might generate highly sequential code which reused storage, as illustrated by Figure 2, in which a single partial control simulates the operation of a program counter in a von Neumann computer.

For appropriate problems where high performance is the essential aim and a "pure" data flow calculation is adequate, such as Mesh calculations or Fast Fourier Transforms, then programs could be encoded in some form of concurrent programming language (such as presented in section 5), probably based on a "single assignment" rule [7,8], and would be translated into a data flow style of representation, as shown in Figure 3, to take advantage of any inherent concurrency. In such cases the control graph will correspond to the flow of data. However, for other more general types of calculation where additional sequencing constraints are needed [6] extra control nodes and arcs can be provided in the control graph to supply just the necessary synchronisation, without incurring the over-specification of sequence which is characteristic of von Neumann computers.

Our research project, since it is concerned with general-purpose computing, concentrates on questions relating to such highly concurrent (implicitly concurrent, explicitly sequential) computations, and on the design and implementation of a programming language which facilitates the generation of highly concurrent GCF programs. The programming language that we are developing is termed the SDG language, since we visualise programs in the language as "structured directed graphs". Section 4 of this document sketches the present
control instructions

Partial controls from the instructions which produced a, b, c and d

control node

\[ e := a \times c - b \times d \]
\[ f := a \times d + b \times c \]

Partial controls for e and f

memory

a: □
b: □
c: □
d: □
e: □
f: □

Figure 1: A Program Fragment for Complex Multiply

data instructions

A single partial control

control node

e := a \times c

t := b \times d
e := e - t
f := a \times d
t := b \times c
f := f + t

Figure 2: Sequential Code for Complex Multiply
Figure 3: Highly Parallel (data flow-like) Node for Complex Multiplication

The diagram shows a control network for a complex multiplication operation, where each node represents a control point and the links represent data flow. The network is designed to efficiently parallelize the multiplication process, minimizing the overall computation time. The nodes labeled with equations illustrate the computation steps involved in the multiplication process, with each node performing a specific operation on the data flow.
state of our ideas concerning such programs. We anticipate that it will later prove feasible to extend the GCF ideas so as to simplify the process of compiling SDG programs greatly - this is discussed briefly in section 6. However we will firstly discuss the present state of our ideas on structured directed graphs.

2.1 Control and Data Graphs

In a GCF computer the control flow graph, conceptually, transmits and processes single bit control tokens which indicate the presence of a control signal on an arc. A control node becomes active when one of a set of combinations of tokens arrives on its input arcs. The combinations which can cause activation are determined by the type of the node. The activated control node then invokes any associated data instructions, and waits for them to terminate before releasing control tokens onto its output arcs. Thus a control graph in the GCF organisation exhibits similarities with the data graph in a data flow organisation. However, the control tokens in the former are far simpler than the data tokens in the latter, and because control tokens are uniform it is only necessary in an implementation (as presented in section 3) to determine the number of tokens which have arrived at a node, rather than also determining on which arcs they are travelling.

The data instructions of the GCF organisation may also be viewed as forming data graphs. These data graphs merely indicate which data instructions can read and which can write particular storage locations, and are therefore similar to the data graphs used by compiler optimisation, rather than the graphs used by data flow computers. No information, either tokens or values, "flows" in the data graph of a GCF program and thus GCF programs are different from the data flow and control flow graphs of Dennis [9], in which both graphs carry tokens, and the control graph carries boolean data values.

We have chosen the above view of control and data instructions so as to obtain the maximum flexibility and influence over the transformation of data by programs. We are encouraged in this view by the fact that the GCF organisation has similarities with the LOGOS [10] system, whose representation was chosen for reasons of flexibility. There are certain minor differences between the two representations, resulting from their different applications. These differences include the format and types of operations provided by the control and data instructions, and the way in which they are implemented.

2.2 Examples

Because of the nature of the GCF organisation, in the following description control instructions will be represented by directed control graphs, while data instructions and memory locations will be shown as program fragments. We begin with GCF representations for simple programs that do not involve procedure invocations, complex data structures or input/output. In particular the GCF representations of a variety of program specifications for a simple iterative program fragment for factorial are examined.

The first program to be considered, shown in Figure 4, is expressed in a conventional Algol-like notation with an underlying sequential
control flow model of computation. Variable "n" is the input to the program fragment and "ans" is the output variable. An equivalent GCF representation for this program is also shown in Figure 4. The two types of control node introduced by this example are the A node, which is activated by a single partial control, and the OR node, which can be initiated by a partial control from either of two sources. In this example the OR node also supports a conditional capability by placing a partial control on either the "true"(T) or "false"(F) output, depending on the result of the TEST instruction.

The second factorial program, in Figure 5, is expressed in the MIT language DFL [11], and is defined in a data flow style of GCF representation. In this example it is assumed that the data identifiers "i" and "x" have been compiled to mimic the arcs of data flow graphs by providing storage for "new" and "old" values of x and i. At the end of each iteration the new values, i.e. new.x and new.i, are copied to the old locations, i.e. x and i.

Anticipating the "activation by need for data" mechanism to be proposed in section 4, the third, and final, example shows in Figure 6 the representation of a program which is intended to be activated by the need for the result value. The example, a simple extension of the representation in Figure 5, uses a special control node that checks a named memory location to see whether it contains a result (i.e. "ans"). The control node then either signals the availability of the result, or generates further activation by need signals which request the availability of the input ("n") needed to create the result.

Having examined how the GCF organisation can be used to represent simple programs, its ability to handle procedure invocations is illustrated by the recursively specified program fragment for factorial shown in Figure 7. The scheme for implementing procedure calls is based on a naming mechanism we developed for the Manchester data flow systems, which is well described in [12]. Basically, each invocation of a process is tagged with a unique name (e.g. creating what Dennis calls coloured tokens [9]) which distinguishes the calls and also the different versions of any temporary storage created by the process.

2.3 GCF contrasted with Data Flow

Although the GCF organisation was originally intended as a basis for research into a wide variety of computer architectures, rather than as a specific proposal for an MIMD architecture, it is possibly appropriate at this point to contrast the GCF with the data flow organisations, as a means of representing concurrency at the machine level.

Firstly the case against the GCF organisation. Compared to data flow, the GCF organisation incurs the following penalties (i) extra storage, because of the replication of partial controls (i.e. control tokens), and because both control and data instructions are needed to represent a program; and (ii) an additional processing time overhead, to execute the additional code as well as to move data operands between a processor and store. These criticisms are clearly true, particularly for simple processes in which a processor must wait for partial results to be accessed from store. However,
x:=1; i:=1;
WHILE i≤n DO
BEGIN
  x:=x*i;
  i:=i+1;
END;
ans:=x;

a) Algol-like, sequential control flow, representation

b) GCF representation

Figure 4: Algol-like Representation of Iterative Factorial.
for \( x := 1, i := 1 \)
\[ \text{if } i > n \text{ then } x \]
\[ \text{else iter } x \times i, i+1 \]

a) data flow notation

b) GCF "activation by availability" representation.

Figure 5: Data Flow-like Representation of Iterative Factorial.
for $x := 1, i := 1$

if $i > n$ then $x$

else iter $x^i, i+1$

a) data flow notation

b) GCF "activation by need" representation

Figure 6: Activation by Need Representation of Factorial
factorial (n) = IF n=1 THEN 1 ELSE n * factorial (n-1)

a) recursive specification of Factorial

b) GCF representation

Figure 7: Representation of Recursive Factorial
the increase in the size of code will be significantly less than double, as illustrated by the previous examples, and for processes using data structures, the execution time should be similar to a data flow computer.

On the positive side, the GCF organisation is believed to gain from (i) its ability to support a spectrum of other organisations, including conventional control flow; (ii) the use of a concept of storage, which means that GCF computers do not have to replicate partial results, and that partial results and retained data are treated in an equivalent way; (iii) the use of partial controls to activate a computation, which reduces the information (e.g. data tokens) moving round the computer, and the complexity of the mechanism required to group together sets of controls (an example is described in the next section); and (iv) the possibility of sophisticated program control (i.e. synchronisation) structures, which may also be used for optimising the usage of the computer's resources.

The above description of the GCF organisation has been somewhat abstract, and has not discussed how such programs, and their directed graph constituents, might actually be represented and executed. There are a number of ways in which this could be done — one scheme which shows promise is presented in the next section. This particular scheme, which has its roots in [5], is aimed at environments where high degrees of concurrency can be exploited.

At a superficial level this GCF computer architecture may appear similar to either conventional control flow architectures or data flow architectures, depending on how the GCF is described, which is only to be expected since the GCF is an attempt to synthesise these two organisations. In practice the GCF is radically different from conventional control flow architectures. For example, the GCF is based on partial controls which can "fan-out" to activate multiple streams of code, or can "fan-in" (i.e. be grouped together) to synchronise the streams. In keeping with this "implicit concurrency, explicit serialism" philosophy a partial control can also activate a process which can run concurrently with the calling process. These independent instruction streams generate "heaps of work" (in a similar way to a data flow organisation) which can be serviced asynchronously by the replicated resources in the computer. However, the GCF scheme has equally radical differences from conventional data flow, e.g. the use of passive storage facilities, rather than continuously regenerating partial results and the use of control tokens rather than data tokens to signal the availability of data.
3. GENERALISED CONTROL FLOW COMPUTER ARCHITECTURE

The style of program representation employed by the GCF computer architecture varies slightly from the conceptual GCF notation used above, because conventional practice has been followed by interspersing control instructions within sequences of data instructions. The resulting program representation is illustrated by Figure 8, which corresponds to the data flow program fragment of Figure 5. In this example JOIN instructions are activated by the arrival of two partial controls and release a single control, whereas the FORK is activated by a single partial control and releases two controls.

3.1 The Computer Organisation

Partial controls in the GCF computer define the names (e.g. addresses) of independent tasks (which can be a single instruction or a procedure) that are ready to be processed by the computer. These names are grouped together in what may be viewed as an unordered set of tasks that provides a "pool of work" for the computer. In simple terms the GCF computer (see Figure 9) consists of 3 units, namely (i) the Task Unit which stores the names (e.g. instruction addresses) of tasks requiring processing, (ii) the Processing Unit, comprising a possibly large number of processing elements and (iii) the Memory Unit providing the program and data storage.

When a processing element in the Processing Unit becomes idle through the completion of a task it selects a next instruction address from the Task Unit. Using the address, the processing element follows a conventional instruction execution cycle by fetching the instruction from the Memory Unit and decoding it to obtain: (i) the operation required, (ii) the addresses of the input and output operands and (iii) the address(es) of the following instruction(s). The processing element then extracts the input operands from the Memory Unit, performs the operation, placing the results back in memory, and then places the next instruction address(es) in the Task Unit, before returning to the idle state. Next instruction addresses need to be included in instructions because there is no concept of centralised registers in the machine.

3.2 Instruction Formats

Within the GCF computer there are basically two classes of instruction, namely data instructions, which perform arithmetic and logical operations, and control instructions which organise the execution of a computation. Figure 10 illustrates the format of these two classes of instruction and Figure 11 shows the style of machine code that could be generated from a program fragment for iterative factorial.

If the GCF instruction formats are examined with regard to the way instructions are activated by partial controls then three types of control flow are of interest: (i) "single-input-single-output" (e.g. all data instructions), (ii) "single-input-multi-output" (e.g. FORK instructions) and (iii) "multi-input-single-output" (e.g. JOIN instructions).
for $x := 1$, $i := 1$

if $i > n$ then $n$
else iter $x * i$, $i + 1$

a) data flow notation

b) GCF "activation by availability" representation

Figure 8: GCF Style of Program Representation
Figure 9: The Generalised Control Flow Computer Organisation
a) general instruction format

<table>
<thead>
<tr>
<th>operator</th>
<th>input operand addresses</th>
<th>output operand addresses</th>
<th>next instruction addresses</th>
</tr>
</thead>
</table>

b) data instructions

i) one input operand

<table>
<thead>
<tr>
<th>operator</th>
<th>input operand address(p1)</th>
<th>output operand address(p2)</th>
<th>next instruction address(p3)</th>
</tr>
</thead>
</table>

ex. COPY p1 INTO p2 THEN p3
NOT p1 INTO p2 THEN p3

ii) two input operands

<table>
<thead>
<tr>
<th>operator</th>
<th>input operand addresses(p1,p2)</th>
<th>output operand address(p3)</th>
<th>next instruction address(p4)</th>
</tr>
</thead>
</table>

ex. ADD p1,p2 INTO p3 THEN p4
COMPARE_EQ p1,p2 INTO p3 THEN p4

c) control instructions

i) alternative next instruction address

<table>
<thead>
<tr>
<th>operator</th>
<th>input operand address(p1)</th>
<th>next instruction address(p2)</th>
<th>next instruction address(p3)</th>
</tr>
</thead>
</table>

ex. IF TRUE p1 THEN p2 ELSE p3

ii) JOIN

<table>
<thead>
<tr>
<th>JOIN</th>
<th>no. of inputs required(p1)</th>
<th>next instruction address(p2)</th>
</tr>
</thead>
</table>

iii) FORK

<table>
<thead>
<tr>
<th>FORK</th>
<th>no. next instr. addresses(p1)</th>
<th>next instruction addresses(p2-pn)</th>
</tr>
</thead>
</table>

Figure 10: GCF Instruction Formats.
\[ y - x = 8 \]
\[ A = B + C \]

\[ S1 \quad S2 \quad S3 \]

Begin

\[ S2 \quad S1 \quad S3 \]

End

at some (call)
load

\[ \text{op} \]

Next line
Figure 11. GCF "machine code" representation of Figure 8.

Activation of a data instruction, for example, is caused by a single partial control, and having been processed the computation instruction passes control to a single following instruction via its address. Activation of a "single-input-multi-output" instruction such as FORK, usually signifies the end of a sequence of data instructions and hence the availability of one or more partial results. The processing element executing a FORK instruction extracts a number of next instruction addresses and places them in the Task Unit, thus activating or contributing to the activation of a number of instruction streams.

Finally, activation of a JOIN instruction (whose task is little more than to facilitate the grouping of a set of partial controls) signifies that one of a set of controls has been generated for the instruction. The processing element executing the JOIN must initially determine how many other partial controls have already been generated for this instruction. To determine when a particular JOIN instruction can be activated (i.e. has reached its threshold) two input parameters are supplied, one specifies the number of controls required and the other defines a store location, in the Memory Unit, in which a count of controls received is built up. (Owing to the asynchronous nature of the GCF computer it is clearly necessary for the processing element to "increment and read" the contents of this memory location as an indivisible operation.) If the number of controls received is still less than the number required, the processing element merely discards the JOIN instruction. If however the correct number of controls have been received, the element resets the memory location and places the next instruction address in the Task Unit.

3.3 The Naming Mechanism

Central to the concurrent operation of the GCF computer architecture is the "naming mechanism", that uniquely identifies each object (i.e. element of data or instruction) stored in the Memory Unit, as well as distinguishing between concurrent activations (invocations) of a specific procedure either within a single program or within separate programs, thus implementing a procedure calling mechanism.
(We use the term "name", as opposed to the more traditional terms of "address" or "pointer", because (i) the name concept is intended to be implementation independent, hiding such details as indirect addressing, multi-length address fields, and linearly addressed stores; and (ii) a name is used to identify both passive objects such as data, as well as active objects such as processes.) The name attached to a stored object or process is also used to route data, and code, round the GCF computer and to units (e.g. Memory Units) of other concurrent systems.

The naming mechanism used in the GCF architecture represents a merging, and we believe a simplification, of the name mechanism of the Manchester data flow systems [4, 12] and Miranker's procedure mechanism [13]. It is possible to shorten the name used, for example, because partial controls "flow" round the GCF computer, as opposed to partial results in a data flow computer, and it is unnecessary to identify which partial control is conceptually travelling on which particular input arc of a control node. A name, whether referring to code or data, consists of two fields: a <user name> which is allocated at compile time to a particular object (e.g. an internal system name for a function such as factorial or the relative address of an instruction) and an <environment name> which is allocated at run time to identify a particular environment (e.g. a particular invocation of a process or a particular data segment). Thus each next instruction address placed in the Task Unit is a two part name where the <environment name> defines, say, the process and the <user name> specifies a relative address within the environment.

The use of the naming mechanism for such tasks as process activation and parameter passing has yet to be finalised, however one promising scheme, influenced by the B6700 Tasking Mechanism [14] and based on descriptor-like names is being studied. If for example a process was executing in an environment "P" then the majority of the addresses "i" specified in the code (either for instructions or data) would be relative to P (e.g. P.i). However, certain addresses within P could reference objects in other environments using an appropriate descriptor (e.g. R.i). Parameters, including operand addresses in P, input operands and return addresses (possibly not in P), could therefore be written to a data segment corresponding to R. This new data segment would be dynamically created by the system at the first access from P. The R process could then be activated simply by placing a next instruction address such as "R.i" in the Task Unit.

3.4 Implementing the GCF Architecture

The above, admittedly vague, description of the GCF computer has concentrated on the information flow (e.g. logical organisation) within the architecture while largely ignoring the details of implementing such a machine. One of the key implementation difficulties is how to design the architecture in such a way that the units which make up the computer can interact asynchronously, allowing the processing elements to access the information in the Task Unit and the Memory Unit concurrently without getting extensive memory clashes.

The solution currently under investigation centres on a ring-based architecture in which all the asynchronous units making up the GCF computer communicate via packets of information placed on one or
more "slotted rings". (A slotted ring may be viewed as a circular conveyor belt subdivided into a number of individual segments, each capable of holding a packet.) Thus the Task Unit could be represented by a ring in which each slot contained a next instruction address. This ring-based approach also allows concurrency to be exploited in the execution of individual instructions by decomposing (i.e. pipelining) the operations performed in the normal instruction fetch-decode-execute cycle and allocating them to separate units around the ring.

The operation of a single ring GCF computer can be illustrated using Figure 12, in which 3 types of functional unit, namely program stores, data stores and processing elements, are connected to the ring and communicate using information packets. Four types of packet traverse the ring: (i) packets containing next instruction addresses ("nia" packets), (ii) incomplete packets comprising instructions without their input operands, (iii) complete packets containing instructions and their inputs and (iv) result packets consisting of output addresses and operands, and next instruction addresses.

For example, to activate a simple dyadic add instruction a next instruction address (e.g. %100) might be placed on the ring:

\[
\text{"nia" %100}
\]

This nia packet will be read by the relevant unit in the Program Store and the corresponding instruction will be output on to the ring in the next vacant slot:

\[
\text{"incomplete" + %a, %b, %c then %201}
\]

This instruction packet is labelled "incomplete" as it does not contain input operand values, but only their addresses %a and %b. The result operand address is %c, and %201 is the address of the subsequent instruction to be activated.

Units in the Data Store will process this incomplete packet replacing the input addresses by their corresponding operands and place an executable instruction on the ring.

\[
\text{"complete" + a, b, %c then %201}
\]

Next, an idle processing element will extract this complete packet and execute the instruction, returning the results to the ring.

\[
\text{"result" (a+b), %c then %201}
\]

Finally the Data Store will process this result and place the next instruction address on the ring.

\[
\text{"nia" %201}
\]

It should be noted that in traditional pipelined computer architectures each unit must be able to start on a new piece of work (i.e. another packet) as soon as it has completed the previous piece of work. Thus each type of packet must contain all the information needed to process it. The simple model described above, while illustrating the basic concept of this ring-based architecture, runs
Figure 12: The Ring-based GGF Computer Architecture
into severe performance problems in practice as the instruction rate will be limited by the ring traversal time. We are currently investigating two ways of reducing traversal time - the most obvious being to combine data stores and processing elements into single units. The other is to have more than one ring, each dedicated to carry one type of packet.

In terms of performance there is clearly a limit to the number of units which can be connected to a ring or even a set of rings. For this reason, the ring-based computer is considered to form one module (e.g. board) in some larger GCF computer in which a number of such modules would be interconnected into a tightly-coupled network. The problem of interconnecting these modules, whether to form a high-speed machine or a desk-top computer, is interlinked with the problems of reallocating units on the various rings and scheduling work amongst the modules.

Our current ideas for overcoming problems, such as these, of routing information around a GCF network are based on the naming mechanism described above, and equating names in a particular address range, say, to a Program Store in a particular module. In this case the name of an object is viewed as being used for routing data in a similar way to the number dialled in a telephone system.
4. STRUCTURED DIRECTED GRAPH REPRESENTATION

The SDG language which we are currently in the process of defining constitutes an attempt at producing a convenient high-level user language which could, by a straightforward process of compilation, be used to program a GCF computer in such a fashion as to make good use of its facilities for highly concurrent computation. It is intended as a general-purpose language, capable of coping with such problems as data retention and input/output, as well as arithmetic computations.

As a support for the SDG language description we are also developing a virtual, high level SDG machine on which, conceptually, SDG language programs could execute. Ultimately code generated for the GCF machine is modelled on the behaviour of the abstract SDG machine. (See however the discussion in Section 6).

The SDG language and the abstract SDG machine are together referred to as the "SDG representation" of computation. The role of the SDG representation is to define all of the required computation and data retention constraints at an instant within the concurrent computing system, and thus ensure, for example, that the required input/output relationships between the sub-components of the "total" computation are achieved. One can visualise the SDG representation as produced from programs offered to the system, some initially, others introduced during execution.

The SDG language is so named because we describe its semantics using what we term a "structured directed graph", a directed graph in which a nested set of subgraphs are identified. In fact we describe the computations which a given SDG program would define in terms of flows of data and control in a structured directed graph being processed by the virtual SDG machine. In the SDG representation of computational requirements, the directed graph defines the cause-effect relationships (i.e. flows of control and data) in a computation, while the structural information (i.e. the subgraphs and their boundaries) specifies how a computation was constructed from its constituent sub-components. These points are illustrated by Figure 13 which shows three subgraphs defining a program fragment for calculating the roots of a quadratic equation.

Our aim is that the SDG language should involve the notions of "single assignment" [7,8] and "atomic actions" [15], concepts which have a direct representation in such graphs. Although we have identified two basic definitions of the single assignment rule [6], they can be generalised in the form "an identifier may only be assigned data by one statement in a program". Such a style of programming is illustrated by Figure 14, which shows the programming language statements (corresponding to Figure 13) for three subgraph declarations, one defining the function "quadratic", another the function "sqrt" and the third specifying the use of "quadratic" for obtaining the two roots. The SDG programming language used to specify programs would define them as hierarchies of functions represented by subgraphs, where each subgraph, when activated would give rise to a non-sequential process.

The concepts and syntax of the language we have been developing for MIMD systems, although not fully worked out, are briefly described
Quadratic:

Program:

sqrt:

Figure 13: Three Subgraphs defining a Quadratic Roots Program.
PN quadratic = INPUT(a,b,c:REAL)
    t:REAL = sqrt(b**2-4*a*c);
    x1:REAL = (-b+t)/(2*a);
    x2:REAL = (-b-t)/(2*a);
    OUTPUT(x1,x2:REAL);

PN sqrt = INPUT x:REAL
   . . .
   y:REAL = . . .
   . . .
    OUTPUT y:REAL;

(r1,r2:REAL) = quadratic(i1,i2,i3);

Figure 14: A possible style of Concurrent Programming

queue of requests → function 1
   . . .

queue of requests → function n

Figure 15: A user's view of the handling of multiple requests for service of a function
in section 5.

The major differences between the SDG representation and other representations, used by related work known to us (e.g. data flow), lies in the interpretation of the structured directed graph form of a program: (i) the implications this structure has on concepts such as the appropriate method of activating a sub-program component (i.e. a function) represented by a subgraph, on data retention and on input/output (between subgraphs and also between the entire graph and the outside world), and (ii) the combined use of two distinct evaluation schemes, namely "activation by availability" and "activation by need". For example, the choice of whether to activate a subgraph by the "availability of data" or by the "need for data" could be determined from an examination of the structure and boundary of a subgraph.

4.1 Name and Object Model

In the SDG representation each sub-component (code or data) of a computation, subsequently referred to as an object, is uniquely named. To quote Saltzer [16]: "In a general sense the way a computing system names and manages objects such as data largely determines its ease of use and its range of applicability". Names for objects are required so that computations can refer to objects, communicate or share them, and possibly locate the objects at some future time.

In the SDG representation, an object conceptually is created once and tagged with a unique name, which allows this single copy to be referenced by a number of destination processes (as opposed to the automatic replication of partial results implicit in a data flow representation). Unlike data flow where a partial result, represented as a data token, can only be used once before it is deleted, in the SDG representation, because of this concept of storage, an object can be read any number of times by any number of destination processes. The concept of storage also means that partial results, which can be viewed as temporary objects, are accessed in the same way as files, which are thought of as retained objects (see section 5.6), in the sense that they are not deleted at the end of execution of a process.

Objects may also contain sub-components, and these may be accessed individually, either by "name" or by "position", in a similar way to conventional parameter passing mechanisms where parameters are defined by position or by tagging each one with a unique identifier. The names attached to sub-components would be either user-specified or supplied by the system, based on the order in which the components were defined.

4.2 The creation and deletion of objects

The order of activation of operations within a SDG program graph is determined by the binding of unique names to objects. The creation of such a uniquely named object, by a process representing a subgraph, can be brought about in either of two ways, namely (i) by the availability of data required in the object's creation, which is referred to as "activation by availability", as in data flow, and (ii) by a process wishing to access the object, called "activation by
need" for data, as in lazy evaluation [17]. In either case activation is performed by control signals (i.e. control tokens).

In the use of activation by availability, a control signal is sent to a subgraph each time one of its data inputs (i.e. data tokens) becomes available for use. When the set of inputs is "complete" the subgraph would be initiated. This method of activation seems most appropriate for passing partial results between arithmetic and conditional statements, etc. In Figure 14, for example, when the values i1, i2, and i3 are generated, the quadratic function will be activated to produce the results r1 and r2.

In activation by need a control signal would arrive at the subgraph, requesting it to generate (or make available) a particular object. If the subgraph had not already generated the required object, and the inputs to the subgraph were not all available, further activation by need control signals would be propagated back through the directed graph. This method of activation seems particularly useful for accessing "retained" (i.e. stored) data, such as files, but can also be used for simple expressions, as in Figure 14, for requesting the objects r1 or r2.

These two distinct forms of evaluation, besides solving what we view to be problems of data flow [6], such as accessing files, can also be used, we believe, at run time by a computer architecture to optimise dynamically the computations being executed. If, for example, performance was the key requirement of a computer then processes would be executed largely through activation by availability utilising maximum concurrency, whereas if efficient utilisation of resources was at a premium in the SDG computer, the computer could selectively execute subgraphs using activation by need.

At certain points during the execution of a computation it will be possible to delete particular objects when they are of no further use. These points are related to the concept of temporary and retained objects, as well as input/output objects. In the SDG representation deletion has been divorced both from creation, to which it is tied in the conventional control flow representation (i.e. assignment to a variable destroys the previous data value contained in the variable), and from usage, to which it is tied in the data flow representation (i.e. after a single use a data token disappears). Data created locally, within the boundary of a subgraph, would only be accessible inside that subgraph and would be retained as long as the process representing the subgraph is active. This data retention constraint is defined by the boundaries of subgraphs, but, owing to the hierarchical nature of subgraphs, what appears as temporary data at one level may appear as retained data at some lower level. The lifetimes of retained data such as files must also be specified for the concurrent system by the programming language and by the SDG representation. Using Figure 14 as an illustration, suppose that the object "t" represents stored data, then "t" would only be accessible within the function "quadratic" and would be deleted when the process terminated (i.e. when x1 and x2 were released).

4.3 Atomic Operations

An SDG program representation can be visualised as giving rise to
system activity which has a "dynamic" structure corresponding to the 
"static" structure in the program. Each subgraph in the SDG 
representation will correspond to a language feature such as a func-
tion. The activity that each function defines will be atomic, in 
the sense described in [18]. Thus from the outside, so to speak, it 
will be possible to view each function activation as an operation 
which occurs instantaneously, and from the inside, no changes will 
be visible in the enclosing (i.e. calling) environment. These are 
the structuring notions which are discussed in [15], where the main 
emphasis is on their relevance to fault tolerance. However from our 
present point of view, the ability of a subgraph to hide details, 
restrict interactions with its environment and operate as an indi-
visible, non-interfering process is the main facility in the SDG 
representation for controlling the concurrency and complexity of 
both software and hardware sub-components of a system. 

Each atomic function (e.g. the quadratic function in Figure 14) per-
forms a single task for some "user" subgraph. A function activation 
performing a task can itself become a user requesting service from 
functions at a lower level; these resulting function activations 
also appear to operate atomically. Thus the user of a subgraph may 
be another subgraph or, say, a person sitting at a terminal, which 
requests service from the subgraph either by (i) presenting the sub-
graph with a complete set of inputs (i.e. activation by availabi-
ity) or by (ii) asking the process to produce some subset of its out-
puts (i.e. activation by need).

When "users" access the subgraphs of a concurrent SDG computation 
they frequently do not act in co-ordination, hence the SDG computing 
system must cope with multiple, and non-determinate, requests for 
the service of a subgraph and also the updating (e.g. generating new 
versions of a function or file) of the subgraph. It is also impor-
tant that the SDG representation allows the programmer to take a 
"simple" view of the way multiple requests are serviced by a sub-
graph. One solution is for the SDG system to queue the requests for 
service automatically, as illustrated by Figure 15. The actual 
method of servicing these multiple, and possibly non-determinate, 
requests, however, depends upon the nature of the individual sub-
graphs and the way they can give rise to processes which will per-
form the actual task. For example, a single user subgraph control-
ling a physical resource such as a line-printer would queue all re-
quests greater than one, while a re-entrant software subgraph could 
conceptually give rise to an unlimited number of processes which 
could service requests concurrently and a file manager might service 
reads concurrently but process writes serially.

Subgraphs can therefore be classified into two groups, namely (i) 
those without memory, which are unaffected by servicing a request, 
and (ii) those which contain memory, whose state is altered by exe-
cution. Basically, memory-less subgraphs operate by what we term 
[4] "dynamic parallelism", in that each request for service is per-
formed by a new invocation of a process, while subgraphs containing 
memory operate by "static parallelism" in which a single process 
sequentially services the queue of requests. (These two forms of 
concurrency are illustrated by Figure 16.) Therefore a computing 
system supporting the SDG representation would need to provide both 
types of concurrency.
Figure 16: Two ways of representing and exploiting parallelism
It remains to be seen whether some of the more sophisticated forms of concurrent activation of atomic operations discussed in [15] could also be incorporated into a GCF architecture which was being used to implement SDG programs.
5. THE SDG Programming Language

The programming language presented in this section is intended to act as a vehicle for investigating languages based on the structured directed graph concepts and also to help identify a new style of programming suitable for expressing the "natural structure", such as concurrency, of an algorithm. Thus the aim is to have concurrency implicit, and sequentiality explicit, rather than the more conventional opposite approach. Programs are made up of functions whose individual bodies consist of groups of statements (though they are preferably termed "declarations"), binding objects (values) to names, which must obey a single assignment rule, and are regarded as being activated concurrently. A programmer can introduce sequencing constraints by having a statement cause further statements to be activated through data dependencies - such statements can be part of a subsidiary function, or of a conditional or iterative construct. The objects manipulated by a program have an existence which is tied to that of the activation of the function in which they are declared. Thus the relationship of the language to the structured directed graph scheme of program representation, though close, is implicit - there is no reason for the user programmer to think in terms of the graph.

The programming language is based on the SDG "name and object" model and on the basic form of program statement:

\[
\text{name generator} = \text{object generator}
\]

which a programmer would use to specify sets of name and object dependencies (i.e. how new names are created and bound to objects).

\[
\text{name} ::= \text{simple name} | \text{structure element name}
\]

\[
\text{simple name} ::= \text{identifier}
\]

\[
\text{structure element name} ::= \text{simple name}[\text{component selector}]
\]

Each name is defined (i.e. declared) at one point in a sub-program or function and can be bound only once to an object at runtime (i.e. the unique assignment form of the single assignment rule [6]). Activation of such a binding may be either by availability or need, depending on the type designation of the name or the way the computer chooses to execute the statement, but this should not concern the programmer. Once the object has been created and named it may be read any number of times during the lifetime of the program component in which it was created. Thus deletion of objects should no longer be of concern to the programmer. The primary aim of the name/object binding mechanism is to provide the programmer with a means for specifying the necessary synchronisation constraints for his computation without resorting to notions such as a "sequential flow of control" implicit in all conventional systems.

The style used in the following brief description of the programming language is borrowed from Algol 60. Curly braces {} have been introduced into the BNF to denote zero or more repetitions of the enclosed material. The programming language, set at a PASCAL level,
as opposed to the more sophisticated Alphard [19] or CLU [20], basically consists of three classes of statement, namely (i) type declarations, (ii) function declarations and (iii) simple declarations (i.e. the conceptual commands of the language).

\[
\text{<program>} ::= \text{<declaration>} ; \{ \text{<declaration>} ; \}
\]

\[
\text{<declaration>} ::= \text{<type declaration>} | \text{<function declaration>} | \text{<simple declaration>}
\]

5.1 **Type declarations**

\[
\text{<type declaration>} ::= \\
\text{TYPE} \text{<type name>} = \text{<data type>}
\]

\[
\text{<data type>} ::= \text{<unstructured data type>} | \text{<structured data type>}
\]

Type declarations in a language are used to declare a name which can be subsequently used within the program to define the set of values which some other name may assume. Only one type may be associated with a name although the type can be quite complex, as for example when defining a record. For simplicity the data type mechanism has been modelled on a combination of that used in PASCAL and that discussed by Hoare [21]. Two classes of data types may be defined, (i) unstructured data types consisting of enumerations and subranges, and (ii) structured types which include cartesian products, discriminated unions and power sets.

The language provides the standard predeclared types: INTEGER, REAL, BOOL, CHAR.

**Examples:**

1) An enumeration :
   \[
   \text{TYPE suit} = \text{(club, diamond, heart, spade)}
   \]

2) A subrange :
   \[
   \text{TYPE year} = 1900..1980
   \]

3) A cartesian product :
   \[
   \text{TYPE complex} = \text{(realpart, imagpart:REAL)}
   \]

5.2 **Function Declarations**

\[
\text{<function declaration>} ::= \\
\text{FN} \text{<function name>} = \text{INPUT <parameters>} \\
\text{<program>} \\
\text{OUTPUT <parameters>}
\]

\[
\text{<parameters>} ::= \text{NULL} | \\
\text{<typed name>} | \\
(\text{<typed group>}{;\text{<typed group>}})
\]

\[
\text{<typed group>} ::= \text{<list of names>:<type name>}
\]
A function declaration is equivalent to a procedure or subroutine declaration of a conventional language except that all communication with a process representing an invocation of a function is via its input and output parameters. Recursive functions are allowed. Once a "function" has been defined it may be used in a similar way to functions in conventional systems. One detailed point — we allow monadic and dyadic functions to be written in the familiar infix format of arithmetic, which increases the readability of programs.

Examples:

```
FN add = INPUT (a,b:INTEGER)
  c:INTEGER = a+b;
OUTPUT c:INTEGER;
```

```
FN quadratic = INPUT (a,b,c:REAL)
  t:REAL = sqrt(b**2-4*a*c);
  x1:REAL = (-b+t)/(2*a);
  x2:REAL = (-b-t)/(2*a);
OUTPUT (x1,x2):REAL;
```

5.3 Simple Declarations

```
<simple declaration> ::= <name expression> = <object expression>;
<repetitive statement>
```

```
<name expression> ::= <typed name>
| (empty)
| <object expression> [ ; ]
```

```
<object expression> ::= <object> | : <object> A (S
  (empty) (object) [ , (object) ] )
```

```
<object> ::= <constant> | : <function call> |<
| <arithmetic-logical expression> |<
| <repetitive expression> |<
```

To specify an algorithm for computation a programmer supplies one or more simple declarations, which may be thought of as statements or commands, which specify how unique names in the <name expression> are to be bound to data values created by the <object expression>. The <name expression> may therefore be viewed as a name generator. Where a name expression consists of a sequence of identifiers then invocation of the statement may be thought of as being equivalent to concurrent assignment in conventional systems.
An *object expression* is any expression on the right hand side of an equals "=" operator that yields objects for binding to the names generated on the left hand side of the equals. The number of objects generated by the *object expression* must equal the number of names generated by the *name expression*. The simple objects: constants, simple names, and arithmetic and logical expressions, are the same as in conventional languages, such as PASCAL. Arithmetic expressions may also contain function calls.

Examples:

1) \( a:t = b; \)
2) \( (a_1, a_2, \ldots, a_n: t) = f_1(b_1, b_2, \ldots, b_m); \)
3) \( (a_1: t_1; a_2: t_2; \ldots, a_n: t_n) = f_2(b_1, b_2, \ldots, b_m); \)
4) \( a:\text{INTEGER} = 10; \)
5) \( b:\text{CHAR} = "c"; \)
6) \( (d:\text{REAL}; e:\text{REAL}) = ((x+y)*z, \sqrt{x}); \)

5.4 **Conditional expressions**

\[
\text{<conditional expression> ::= CASE <selector expression> OF}
\quad \{<selector><object expression>\} FO
\quad \text{<selector expression> ::= <object>}
\quad \text{<selector> ::= <object>:{<object>:}}
\]

The *conditional expression*, providing the language with a conditional capability, operates as follows: firstly the *selector expression* and the *selector*s are evaluated to select the target *object expression*, and secondly the selected *object expression* is evaluated to yield the value of the *conditional expression*. The *selector expression* must match exactly one *selector*.

This style of conditional expression has been chosen to exclude non-determinate operation. Non-determinate calculations are felt to be so important to concurrency that we wish to study them in isolation, as opposed to having them form a subset of the operation of condition statements, such as would be the case with Guarded Commands [22].

Examples:

1) \( \ldots = \text{CASE } c=0 \text{ OF true: } a; \)
   \quad \text{false: } b \text{ FO;}
2) \( \ldots = \text{CASE } \text{charac OF } 'a'::'b'::'c':: \text{charac; '1'::'9':: conv(charac) FO;}
3) \( \ldots = \text{CASE } i \text{ OF } 1\ldots 9: f(i) ; \)
5.5 Repetitive Statements and Expressions

<repetitive statement> ::= FOR <simple name> = <for expression> DO <simple declaration>{;<simple declaration>} OD

<repetitive expression> ::= FOR <simple name> = <for expression> DO <repeated term> OD

<for expression> ::= <initial value> TO <final value> |
<initial value> DOWN TO <final value> |
<initial value> BY <increment> WHILE <condition> |
<initial value> BY <increment> UNTIL <condition>

<repeated term> ::= <binary operator>(<object>)

Repetitive statements:

Repetitive statements in the SDG language perform a very special role by providing the programmer with a "shorthand" notation for defining groups of almost identical statements, e.g.

FOR i = 1 TO n DO
   a[i]:INTEGER = b[i] + c[i] OD;

In conventional control flow systems this statement would be treated as a loop that was sequentially executed, whereas in an SDG system this repetitive statement would be textually expanded

a[1]:INTEGER = b[1] + c[1];
   :;
   a[n]:INTEGER = b[n] + c[n];

either by the language translator at compile-time or by the SDG computer at run time. For this reason, all statements generated must obey the unique assignment rule. It is important to consider the simple name defined by a repetitive statement or expression as specifying a set of subcomponent values, to be applied during the textual expansion, rather than, in the traditional sense, as a store location that will be assigned a sequence of control values. In an example below, we indicate this set of subcomponent values by enclosing them in brackets e.g. (1,2,3,...).

There are basically two types of repetitive statements, namely (i) those such as "FOR i=1 TO n DO" which have a predefined range and can be expanded completely as soon as the value of n is known (similar to conventional FOR-loops), and (ii) those such as "FOR i=1 BY 1 WHILE a[i]>0 DO" which are conditionally expanded at run time (similar to conventional WHILE-DO loops) e.g.

i:=(1);
IF a[1]>0 THEN i:=(1,2) ;
IF a[2]>0 THEN i:=(1,2,3); 

We believe that the use of repetitive statements as a "shorthand" notation, similar to a parallel FOR-ALL, will be a key feature of future concurrent languages, and when combined with the SDG concept of names and objects provides a programmer with almost as powerful a tool to synchronise his computations, as the traditional programmer specified flow of control.

Examples:

1) FOR i=1 TO n DO
   FOR j=1 TO m DO
     a[i,j]:=INTEGER = b[i,j]+(c[j]*10) OD OD;

   is equivalent to

     a[1,1]:=INTEGER = b[1,1]+c[1]*10;
     a[1,2]:=INTEGER = b[1,2]+c[2]*10;
     ... 
     a[n,m]:=INTEGER = b[n,m]+c[m]*10;

2) "Laplace Mesh example"

   FOR i = 1 TO m DO
     FOR j = 1 TO m DO
       u[i,j,1]:=REAL = "initial values" OD OD;

   FOR n=1 BY 1 UNTIL stop[n] DO
     FOR i=1 TO m DO
       FOR j=1 TO m DO
         u[i,j,n+1]:=REAL = (u[i-1,j,n] + u[i,j-1,n] + u[i+1,j,n] + u[i,j+1,n])/4
       OD OD;

     stop[n]:=BOOL=
       FOR i=1 TO m DO
         FOR j=1 TO m DO
           AND( n[1,u[i,j,n+1]-u[i,j,n]] < error )
       OD OD;

Iteration in the traditional sense (i.e. WHILE-DO) is not supported by the language because it is felt to be an inherently control flow concept for handling recurrence relationships. For example, finding a square root by the Newton-Raphson method:

\[ \text{x[i] = first approximation} = \begin{cases} 0.5*(x[i-1] + n/x[i-1]) & \text{if } i>0 \\ 0 & \text{if } i=0 \end{cases} \]

sqrt(n) = x[i] for the smallest i such that
\[ |x[i] - x[i-1]| < \text{error} \]
In traditional programming languages the recurrence is usually optimised by discarding used iterate values, with a consequent slight loss of clarity:

\[
\begin{align*}
x &:= \text{first approximation;} \\
y &:= x - 1; \\
\text{while } |x-y| > \text{error do} \\
\text{begin} \\
y &= x; \\
x &= 0.5^{*}(x+n/x); \\
\text{end}; \\
sqrt &= x;
\end{align*}
\]

LUCID [23], on the other hand, is true to its name and produces a very clear representation:

\[
\begin{align*}
\text{FIRST } x &= \text{first approximation} \\
\text{NEXT } x &= 0.5^{*}(x + n/x) \\
\text{sqrt} &= x \text{ AS SOON AS } |\text{NEXT } x - x| < \text{error}
\end{align*}
\]

(We would argue that this is even clearer than the mathematical representation above!)

In the SDG language the recurrence remains explicit:

\[
\begin{align*}
x[0]:\text{REAL} &= \text{first approximation;} \\
\text{FOR } i=1 \text{ BY 1 UNTIL } |x[i]-x[i-1]| < \text{error DO} \\
x[i]:\text{REAL} &= 0.5^{*} (x[i-1] + n/x[i-1]) \text{ OD;}
\end{align*}
\]

This example exposes a significant hole in the SDG language which must be filled - how do we extract the "result" from the square root iteration? We know that the final iterate is somewhere in the structure called \( x \), but we do not know its exact name. Here is one rather inelegant method for obtaining the result:

\[
\text{sqrt:REAL} = \text{FOR } i=1 \text{ BY 1 UNTIL } |x[i]-x[i-1]| < \text{error DO} \\
+ (\text{CASE } |x[i]-x[i-1]| > \text{error OF} \\
\text{TRUE: } 0; \\
\text{FALSE: } x[i] \text{ FO} );
\]

A possible approach to a more elegant solution is to isolate this type of iterative construct as an "iterative expression", whose purpose would be to produce a single, final set of iterates, and to include it in the syntactic class \langle object \rangle. This approach is essentially the same as that of Ackerman's \textsc{for..iter} construction in MIT's Preliminary Data Flow Language [11], and that of "loop expressions" in Arvind and Gostelow's Id language [24].

The shortcoming of the SDG language mentioned above calls into question the usefulness of the \langle for expression \rangle s:

\langle initial value \rangle \text{ BY } \langle increment \rangle \text{ UNTIL } \langle condition \rangle

and \langle initial value \rangle \text{ BY } \langle increment \rangle \text{ WHILE } \langle condition \rangle

within \langle repetitive statement \rangle s, although they may still be useful
within <repetitive expression>s. Research is proceeding in this area.

Repetitive expressions:

Repetitive expressions consist of a class of APL-like operators which provide the programmer with a shorthand notation for building large expressions from similar, repeated subexpressions.

For example, the vector product

\[ x = \sum_{i=1}^{n} a[i] \times b[i] \]

is usually defined as a recurrence relation, on \( x \), in conventional programs, e.g.

\[
\begin{align*}
X & = 0 \\
& \text{DO } 1 \text{ I=1,N} \\
& \quad X=X+A(I) \times B(I) \\
& 1 \text{ CONTINUE}
\end{align*}
\]

This program is usually mimicked in data flow languages (similar to LUCID [23]) by a program fragment of the form:

```
FIRST i=0
NEXT i=i+1

FIRST x=0
NEXT x=x+a[i]*b[i]

answer = x AS SOON AS i>n
```

It seems more natural that, in a concurrent language, the vector product should be expressed in a form that merely defines the data dependencies, such as:

\[ x: \text{INTEGER} = \text{FOR } i=1 \text{ TO } n \text{ DO } (a[i] \times b[i]) \text{ OD}; \]

which is equivalent to

\[ x: \text{INTEGER} = a[1] \times b[1] + a[2] \times b[2] \ldots + a[n] \times b[n]; \]

or even as a recurrence relationship which obeys the unique assignment rule:

\[
x[0]: \text{INTEGER} = 0; \\
\text{FOR } i=1 \text{ TO } n \text{ DO} \\
x[i]: \text{INTEGER} = x[i-1] + a[i] \times b[i] \text{ OD};
\]

\[ \text{answer:INTEGER} = x[n]; \]

The form of repetitive statements and expressions for a concurrent
language is still far from clear, and this is our reason for keeping the syntax close to a conventional notation.

5.6 Files and Input/Output

The handling of files and input/output in the language is very much interrelated with the SDG concepts of hierarchy and level, and the deletion of objects by the system.

From within a SDG language function the input and output parameters appear as retained objects, that is the names associated with the input parameters at execution time have been bound to objects within the calling function, and the objects returned through the output parameters are to be bound to names within the calling function.

Following this policy, the files and devices used as data input and output channels by a SDG program have the appearance (to the SDG program) of being retained objects belonging to some outer function, i.e. "the operating system". Hence the "operating system"/SDG interface behaves similarly to an internal SDG function call. Data input to a program will be passed to the input parameters of the outermost function of the program, and objects output by this function will be transmitted to appropriate files or devices. The information necessary to set up the association between input/output parameters and files or devices will be supplied to the "operating system" with the external request for execution of the SDG program (e.g. by a user at a terminal).

As a simplifying extension to this mechanism the SDG language provides a standard input channel which will be passed implicitly through all function INPUT parameter lists as a structure name READ, and a standard output channel which will be passed implicitly through all function OUTPUT parameter lists as the structure name WRITE. Again the execution request must specify file or device linkage to these parameters at the level of the outermost function of the SDG program.

The handling of OUTPUT objects is closely tied to indicating to the SDG system when it can delete a given object. The objects within a given function are conceptually viewed as belonging to one of two object domains, namely (i) temporary, and (ii) input/output. When a function terminates all temporary objects can be deleted immediately, while output objects are retained and are bound to names in the calling function. Creating an output object corresponds to moving a temporary object to the input/output object domain. Thus, if the temporary object "t" is created and is to be output, a copy will be transferred to the input/output object domain, and hence will not be deleted automatically at function termination.

As with temporary objects, input/output objects may contain sub-components which can also be referenced either "by name" or "by position". However the notation for identifying the sub-components is still to be finalised.

The SDG language offers the following syntactic sugar as an aid to digestion of assignments to the standard output channel WRITE:

```plaintext
WRITE[i] = x;
```
may be written in the more familiar form:

```
WRITE(i,x);
```

Example of input/output:

```
FN quadratic = INPUT(fa,fb,fc:REAL)

... ...

OUTPUT(fx1,fx2:REAL);
```

i) input/output by name through standard channels

```
a:REAL = READ['a'];
b:REAL = READ['b'];
c:REAL = READ['c'];

(x1,x2:REAL) = quadratic(a,b,c);

WRITE('x1=',x1);
WRITE('x2=',x2);
```

ii) input/output by position through standard channels

```
(a,b,c:REAL) = READ[ ];

(x1,x2:REAL) = quadratic(a,b,c);

WRITE(x1,x2);
```

iii) input/output by name through an explicit channel

```
FN mainprog = INPUT datafile:REAL

a:REAL = datafile['a'];
b:REAL = datafile['b'];
c:REAL = datafile['c'];

(x1,x2:REAL) = quadratic(a,b,c);

resultfile['x1='] : REAL = x1;
resultfile['x2='] : REAL = x2;

OUTPUT resultfile:REAL;
```

5.7 Issues to be Resolved

The SDG programming language presented above is still very much in its infancy and a number of important concepts need to be investigated and finalised. For example, we have found a class of programs, typified by sort-merge [6], which seem to be difficult to specify in concurrent languages. This problem suggests that the present language requires enrichment with extra control statements based on both "activation by availability" and "activation by need". Another inadequacy of the present SDG language is the lack of (i) constructs to support non-determinate operation within a computa-
tion, and (ii) concepts to support the uncoordinated update of re-
tained objects.
6. DEVELOPING AN MIMD COMPUTING SYSTEM

6.1 Status

It has been our intention that the project initially concern itself with clarifying the two forms of program representation, namely the machine level GCF and the language level SDG, which has involved building simulators of GCF-type computer architecture, and implementing software such as programming language translators. A minimal interface is maintained between these software tools by pivoting their designs on a common "target machine language" for GCF-type computers. This target language also acts as an informal definition of the GCF class of computers.

In the architecture area the group is investigating the design of a GCF computer, which might be viewed as forming a single, one module computer (i.e. providing processing and storage) or forming part of a larger, tightly coupled, multi-computer system. The internal design of this module is based on one or more rings (i.e. circular hardware buffers) that provide communication between a number of asynchronous units, such as processing elements, memories and input-output channels. Investigation of various communication mechanisms for interconnecting these computer modules is also underway using simple simulation models.

Currently two GCF software simulators are in operation, one is being used to study the hardware design of the, above, ring-based computer architecture, while the other supports a programming environment by interpreting the GCF target machine language.

In the programming language area, three software tools have been implemented, namely (i) a GCF assembly language, (ii) a high level data flow programming language based in part on the syntax of the SDG language, and (iii) a preprocessor that translates programs written in FORTRAN into the GCF assembly language. Both translators, for the assembly language and the data flow language, generate the target machine language. Current language work centres on defining the semantics of the SDG language and its underlying supporting mechanisms.

6.2 Further objectives

The GCF architecture defines a very "flat", or unstructured, machine. Hence the implementation of an SDG language program would involve a compilation "de-structuring" the SDG program, in much the same way that an Algol 60 program has to be "de-structured" in order to generate a S/370 machine-code program.

Our longer term goal is to design an extended version of the GCF architecture which enables the structure inherent in an SDG program to be retained explicitly at run time. (This is analogous to the way in which, for example, Burroughs computers [14] retain the block structuring information from Algol programs, and the Newcastle Reliability Project's EML architecture [25] retains control flow structure at run time).

If this can be done, the resulting computer would be able to control both data retention and the pattern of concurrent execution of in-
structions directly. The control graph would be extended to include this extra information and the boundaries of subgraphs would be represented explicitly in the machine, so that the control graph would become much more like the conceptual SDG representation described in section 3. The control graph itself would be used in a much more sophisticated way, because, figuratively speaking, not just one but rather two forms of tokens would flow along its arcs. Control tokens would flow, as before, and trigger the execution of data manipulation instructions. However, there would also be "need tokens", which could be generated, and which would flow in the opposite direction through the arcs of the graph until they came to a subgraph boundary, where they would be transformed into control tokens, which would then start flowing normally. Thus the scheme of "activation by need" would be incorporated directly into the basic machine architecture, rather than simulated by a simple control graph (as was illustrated in Figure 6). The actual machine representation of the graph might for example be derived from that described in section 3, but with each control instruction having a level number field incorporated in it, or alternatively new forms of instructions, representing boundaries of subgraphs (the equivalent of "block begin" and "block end" instructions) could be introduced.
7. CONCLUSIONS

It will be clear from the above account that many issues remain to be resolved, and that further investigation may cause a number of the decisions that we have taken to be revised. However our work to date, and our interactions with other groups working on the design and programming of MIMD systems have encouraged us to continue our exploration of systems based on implicit concurrency and explicit sequentiality. In particular, they have encouraged us to continue our attempt to develop a synthesis of ideas from the control flow and data flow approaches to parallel computation. We see as the major issues to be investigated (i) the feasibility of cost effective hardware implementations which enable high degrees of concurrency to be maintained, and (ii) the useability of implicitly concurrent programming languages. On both these fronts we would at present claim merely that the approach has sufficient interest and uncertainty to merit continued research.
ACKNOWLEDGEMENTS

The ideas briefly presented in this document owe much to the support and encouragement of our colleagues. In particular, at the University of Newcastle upon Tyne, Richard Hopkins designed and implemented the data flow language and Heather Robinson produced the FORTRAN translator. The project has benefitted through close collaboration with Ronan Sleep, working on data driven computation at the University of East Anglia, and has also been greatly aided by discussions with Gill Ringland of C.A.P., and Bob Hopgood and Rob Witty of the Rutherford Laboratory of the UK Science Research Council, which partially funds the Project.

REFERENCES


