# Resource Management in Dataflow

A.J.Catto

Departamento de Computacao e Estatistica
Universidade Federal de Sao Carlos
Caixa Postal 384
13560 Sao Carlos SP
BRASIL

J.R.Gurd

Department of Computer Science
University of Manchester
Oxford Road
Manchester M13 9PL
ENGLAND

## Abstract

Recent proposals for nondeterministic facilities in high-level dataflow programming systems have stopped short of giving details of low-level implementation. The underlying machine is assumed to provide basic nondeterministic operations which lead to the required high-level effects. This paper gives details of a practical implementation of one such high-level language, Id {3}, for a specific dataflow computer, the Manchester prototype {11}. It adds to previous work by the authors {7, 8} in which implementations of Communicating Processes {12} and Distributed Processes {5} were proposed.

Id is based on an unravelling dataflow interpreter which closely resembles the labelled token scheme used in the Manchester prototype. Thus translation of Id programs into suitable machine code is relatively straightforward. However, instead of requiring complex nondeterministic operators to support resource managers as in {1}, the existing simple matching functions of the Manchester system {8} prove to be adequate.

For the non-specialist reader, the Manchester labelled dataflow schema and the resource management constructs of Id are outlined before details of implementation are given.



## 1 Introduction

This paper addresses the development of nondeterministic software for dataflow computers, using the prototype system being built at Manchester University {11} as a target for code generation. To date, the few attempts to exploit the potential of dataflow machines for nondeterministic programs have stopped short of giving implementation details {2, 3, 9, 13}. In previous papers {7, 8}, the authors have introduced practical low-level nondeterministic dataflow primitives and shown them to be applicable to the implementation of Communicating Processes {12} and Distributed Processes {5}. However, the efficiency of such implementations is questionable, mainly because the languages retain features of the sequential multiprocessors they were designed for.

In this paper, we report on an implementation of resource managers in the high-level notation of the language Id {1, 3}. This language has been developed from consideration of coloured (unravelled) dataflow graphs, and is eminently suitable for the Manchester machine. The implementation model confirms the natural mapping that was expected (compared with Distributed Processes and Communicating Processes), and shows that the implementation can be achieved without the specialized nondeterministic machine-level operators that were originally proposed {3}.

The two following sections outline the Manchester machine-level notation and resource managers in Id, respectively. The major section describes the implementation model and identifies critical areas for performance.

## 2 Machine Level Dataflow Graphs

A machine-level dataflow algorithm is expressed as a flowgraph. In flowgraphs supported by the Manchester system, nodes implement functions of one or two arguments, and arcs convey data tokens between nodes. In addition to the static arcs determined by a flowgraph, dynamic arcs may be created by the run-time action of some nodes (see section 2.4).

In reentrant flowgraphs, such as loops and user-defined functions, potential clashes between unrelated token sets must be prevented. Tokens could be physically separated, either by preventing a node from firing while any of its output arcs holds a token, or by replicating the reentered subgraph each time it is used. However, the Manchester dataflow system uses logical separation implemented by labelling tokens (see section 2.1). The labelling scheme allows many tokens to wait for their partners at the same two-input node without being confused. Alternatively, it can be viewed as creating several logical instances of the node, each with a unique label, which effectively unfolds the flowgraph {1, 3}. This guarantees maximum asynchronism in execution {1}, and dispenses with actual copying of the flowgraph.

### 2.1 Token Labelling

Labelling of tokens thus allows coexistence of independent instances of a flowgraph in the sense of the Id unravelling interpreter. A label is attached to each token, and only tokens with identical labels can be used as arguments for each firing of a node. As currently implemented, the label comprises three independent fields whose names reflect their most common usage. An activation name separates tokens which belong to distinct activations of a given flowgraph. An iteration level separates tokens which belong to distinct cycles of a loop in a flowgraph. It has the same value for all tokens generated in a particular cycle. An index separates the scalar components of a linear structure. The activation name and the iteration level together determine the colour of a token.

### 2.2 Token Matching

A token that reaches a two-input node requests a match, specifying an **{s-function, f-function}** pair known as the matching function. If an identically labelled partner is waiting at the node, the match succeeds and the **s-function** specified by the incoming token is executed. If no partner is available, the match fails, and the **f-function** specified by the incoming token is executed. The full set of matching functions supported by the Manchester dataflow system is described elsewhere {8}. The following are all that are used in this paper:

(a) {extract, wait} (EW) is the default two-input matching function. If the match fails, the incoming token waits at the node until a partner arrives. If the match succeeds, the node fires and both input tokens are consumed.

(b) {increment, defer} (ID) is used to count passing tokens. If the match fails, the counter has not yet been preset, and so the incoming token is temporarily withdrawn, to try again later. If the match succeeds, the node fires and the incoming token is consumed, while the partner has its value incremented and then remains waiting at the node.

(c) {preserve, generate} (PG) is used in conjunction with a **branch on empty** node (see section 2.4), to isolate the first token to traverse an arc. For the first token the match fails, a value of type **empty** is substituted for the missing partner, the node fires and the incoming token is consumed, meanwhile a copy of it is generated at the partner input. For each subsequent token the match succeeds, the node fires and the incoming token is consumed, while the generated partner remains waiting at the node.

### 2.3 Representation of Data

A token carries a typed value in a data field, and has control fields which define its index, colour, destination address and matching function. A token represented as:

<T:V> ix:I col:C addr:N.P mf:F

carries a value V of type T, has index I and colour C, is destined to go to input port P at node N, and specifies matching function F. The colour of a token may be written as an.il, to make explicit its activation name an and iteration level il.

A linear structure is similar to an Id stream {3}, and has its component tokens represented within curly brackets, e.g.

$\{\langle T_k:S_k\rangle$ ix:k col:C addr:N.P mf:F | k:1...}.

In this paper, valid data types T are the control types introduced above (**col**, **addr**, **mf**), the combinations [**addr**, **mf**] and [**col**, **addr**, **mf**], the nonnegative integers (**ord**) and **empty**. The type of user-defined data is not specified, but may be thought of as (e.g.) integer. We will omit default field values wherever this simplifies the text and diagrams.

### 2.4 Representation of Code

Code is represented as flowgraphs in which nodes are specified by the form of their inputs and outputs. There are two types of node; machine-level nodes and macro nodes. The machine-level nodes are specified below. Subscripts on inputs and outputs stand for **l**efthand and **r**ighthand arcs, as drawn in the later flowgraph figures.

combine col, addr, mf - specifies a dynamic arc for exit from a shared node (a dynamic arc is set up at run-time, rather than compile-time):

```
input  = <addr:A mf:F> col:C
output = <col:C addr:A mf:F> col:C
```

separate structure - isolates the first component of a structure:

```
input    = {<S_i> ix:i | i:1...}
output_l = <S_1> ix:0
output_r = {<S_i+1> ix:i | i:1...}
```

set col - sets the colour of its data input to a given control value:

```
data input    = <val> col:C
control input = <col:C'> col:C
output        = <val> col:C'
```

set ix, set il - similar to set col

set col, addr, mf - sends its data input along a given coloured dynamic arc:

```
data input    = <val> col:C
control input = <col:C' addr:A' mf:F'> col:C
output        = <val> col:C' addr:A' mf:F'
```

proliferate - creates a bounded linear structure:

```
data input    = <val> ix:0
control input = <ord:N> ix:0
output        = {<val> ix:i | i:1..N}
```

synchronize - forces synchronization of tokens on a pair of arcs:

```
input_l  = <val_1>
input_r  = <val_2>
output_l = <val_1>
output_r = <val_2>
```

swap ix, il - swaps the index and iteration level of its input token:

```
input  = <val> ix:i col:a.j
output = <val> ix:j col:a.i
```

branch on empty - routes its data input according to the type of its control input:

```
data input    = <val>
control input = <x>
output_l      = if (x =  empty) then <val>
output_r      = if (x <> empty) then <val>
```

In the flowgraph figures, data input arcs are directed to the top of the nodes, control input arcs to one side. Output arcs always emerge from the bottom. Machine-level nodes are drawn as solid boxes, macro nodes as double-walled boxes. Dotted macro node boxes represent expansions which are not defined in the text, either because they are trivial, or because they are beyond the scope of this paper. Dashed macro node boxes represent shared nodes which are expanded once only. Where arcs are referred to by name in the text, the name is written in a simple dotted box in the figure. Global inputs to each flowgraph are drawn **□**, and literal-valued inputs to all nodes are drawn **T**.

## 3 High Level Resource Managers in Id {3, 5}

A manager is a means of enforcing an access policy, to ensure that the state of a resource shared by independent users follows a valid history. Such an access policy is expressed in Id by a definition, from which managers can be created and associated with particular resources. For instance, the definition

```
FileMan := manager (infile)
           (...)                                  (3.1)
```

might specify a general file managing policy, from which the statements

```
aMan := create (FileMan, a);
bMan := create (FileMan, b)                       (3.2)
```

create two uniquely named file managers, aMan and bMan, where the formal parameter infile of the definition is initialized with file values a and b, respectively.

A definition specifies named ports, via which managers and users interact, e.g.

```
FileMan := manager (infile)
           (entry read: RREQUEST;
                 write: WREQUEST
            do ...
            exit read: RRESULT;
                 write: WRESULT)                  (3.3)
```

A user accesses a port in a manager via a use construct, in a statement such as

```
wresult := use (aMan.write, wrequest)             (3.4)
```

The parameter aMan.write identifies the manager and port being accessed, wrequest brings the required input data, and wresult is the destination for the output. The use construct merely transfers the value wrequest and a reference to wresult to the entry of port aMan.write.

Each entry in a manager has an associated stream, where requests are buffered as they are received. In a manager created from definition (3.3), for example, stream RREQUEST will contain all requests received at entry read, while WREQUEST will contain those received at entry write. The manager produces streams RRESULT and WRESULT, whose components are in strict one-to-one correspondence with those of RREQUEST and WREQUEST. An implicit communication between the entry and exit of each port provides for $RRESULT_i$ and $WRESULT_j$ to be directed to the use constructs which originated $WREQUEST_i$ and $WREQUEST_j$, respectively.

Parameters like infile can be eliminated from a manager definition by a compile-time transformation documented in the Id report {3}. Such a transformation is not specifically dealt with here, but the following manager definition is considered to be the general case:

```
mandef := manager
          (entry name^1: X^1; ... name^n: X^n;
           do ...
           exit name^1: R^1; ... name^n: R^n)
                                                  (3.5)
```

## 4 Implementation of Id Resource Managers

To summarise the previous section, managers may be created and used (and also deleted, although we shall not consider deletion in this paper) using high-level Id constructs. It is intended that a manager definition be expanded just once, and that all flowgraphs should share this definition without their tokens being confused. This section shows how token labels can be used to achieve this.

Implementations of resource managers in the Id base language and the Manchester dataflow machine differ in two major ways. Firstly, Id postulates complex machine-level nodes which implement the base language operators, whereas the same effect is achieved by macro nodes in the less specialized Manchester system. Secondly, the techniques for generating unique labels vary. Id forms a new colour by concatenating the unique colour and address of the calling construct and using this as an activation name. This recursive technique is appealing, but it is infeasible on a machine with fixed label length. In the model presented below, consecutive activation names are generated by a "counter", local to the manager definition, which increments each time a create instruction is executed for this manager. The colour assigned to tokens referring to a particular instance of the manager is derived from the value of the counter at the time that instance was created. In both implementations, the actions involved in executing the base operators are the same.

create involves: (i) sending a request from the calling context to the manager definition, asking for an instance of the manager to be created; (ii) creation by the definition of an instance of the manager, by reserving a colour for that instance; and (iii) informing all users of the instance what the colour is, and the address of each manager entry to which the users can send data when they use the instance.

use involves: (i) sending data from the calling context to the appropriate manager entry address with the colour of the required instance; (ii) eventual servicing of this request by the manager, in turn with all other requests to this instance; and (iii) returning any output from the resource to the correct calling context.

In the Manchester implementation, these actions refer to the flowgraph in Fig.1, which corresponds to the manager definition (3.5). This definition is expanded just once, and is drawn as a dashed box in all flowgraphs which share it. With the exception of **Make**, the internal macro nodes are directly associated with the high-level constructs they implement. **Make** is the special macro node which is used to produce the unique coloured reference to the new manager. Its expansion is shown in Fig.2.

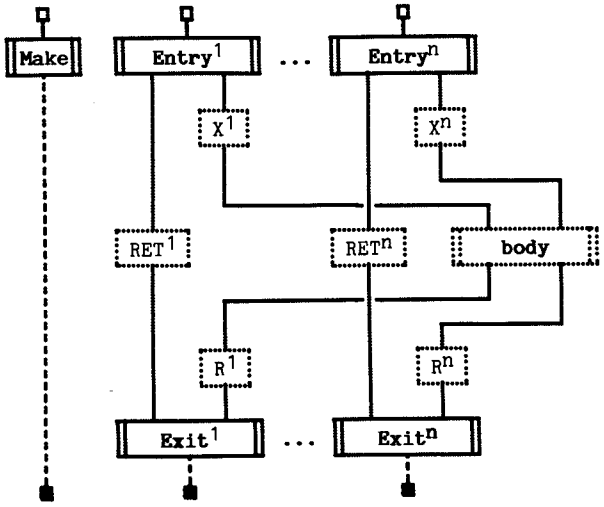


Fig.1 Manager definition flowgraph

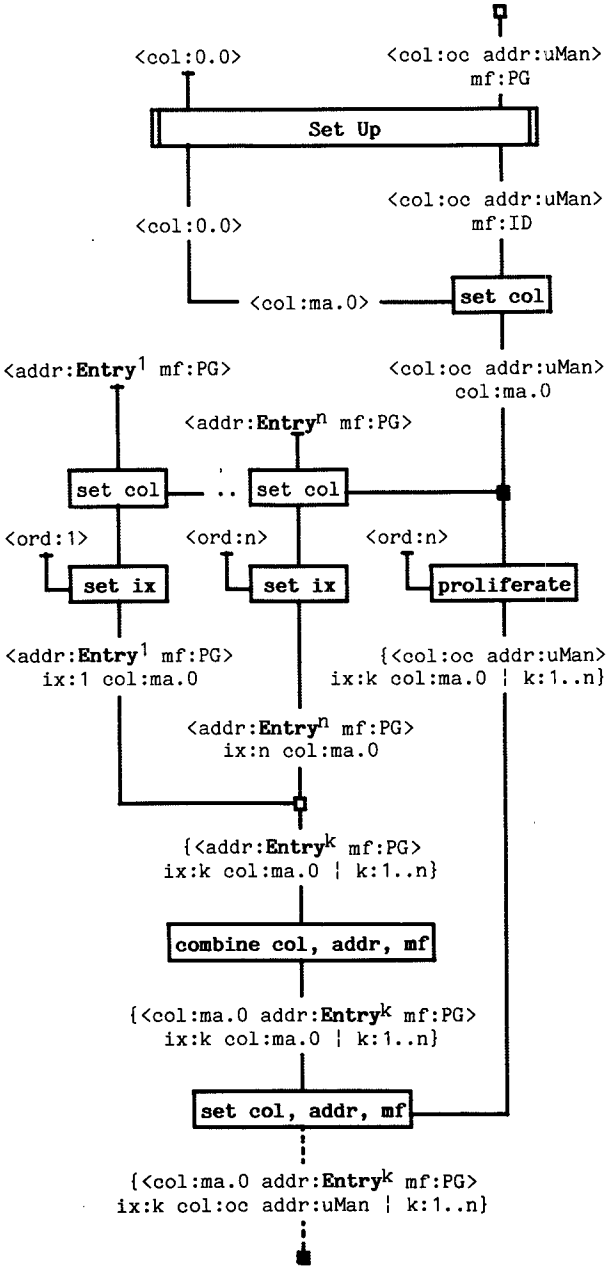


Fig.2 The **Make** macro node

The description of input and output to and from **Make** is as follows

```
input  = <col:oc addr:man> col:0 mf:PG
output = {<col:ma.0 addr:Entry^k mf:PG>
          ix:k col:oc addr:man | k:1..n}
```

(4.1)

The input is the colour and address of the node in the user flowgraph which is to receive the result of the create operation. The output is a structure which gives the colour and address of each entry in the newly created manager. These values are formed from the unique colour generated when the construct is executed, together with the static addresses of the entries in the definition.

The input passes through a **Set Up** macro node which merely initializes the colour generator the first time it is reached (see Appendix). The colour generator is a **set col** node accessed with **{increment, defer}** matching function. This guarantees that no token clashes will occur there, regardless of the rate of access to the macro node, and that no two such accesses will produce the same result. After a unique colour **ma.0** has been produced for the new manager, the output is straightforwardly produced.

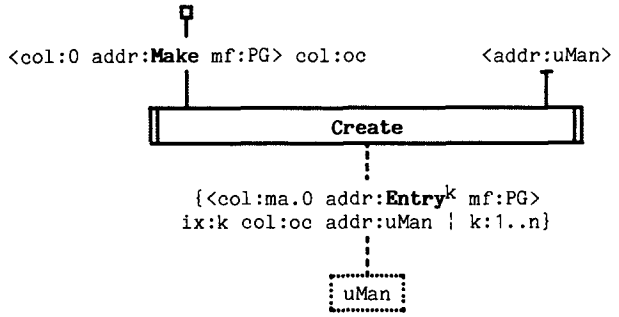


Fig.3 The manager owner flowgraph

## 4.1 Creation of a Manager

Recall that a manager is created from definition (3.5) by a statement of the form

```
uMan := create (mandef)
```

(4.2)

The create construct is implemented by a **Create** macro in the user flowgraph, as shown in Fig.3. It is supplied with the access address and colour (zero) of the **Make** macro node in the appropriate **mandef**, and returns the output of **Make** as the value of the "variable" uMan.

The **Create** macro node, expanded in Fig.4, is specified as

```
input_l = <col:0 addr:Make mf:PG> col:oc
input_r = <addr:uMan> col:oc
output  = {<col:ma.0 addr:Entry^k mf:PG>
           ix:k col:oc addr:uMan | k:1..n}
```

(4.3)

The values of both inputs to this macro node are known at compile time, and its output is the same as that of **Make**.

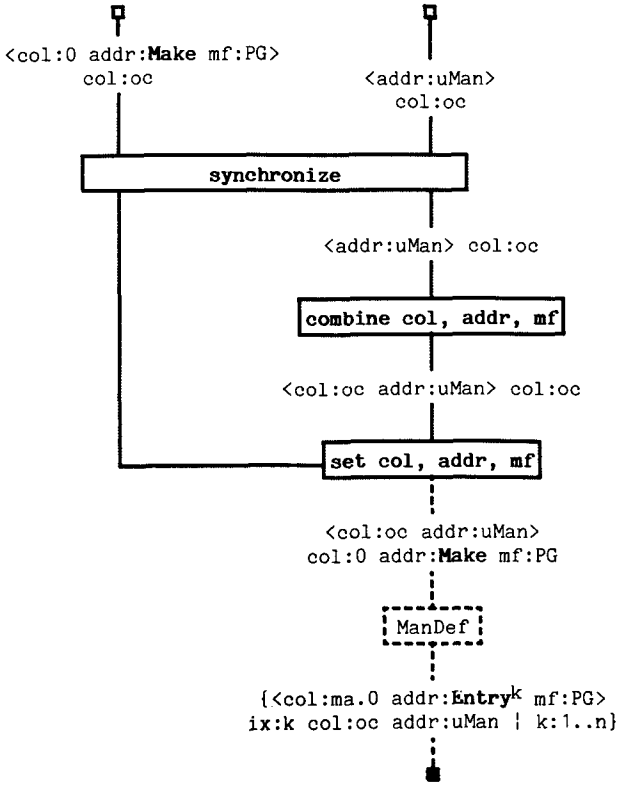


Fig.4 The **Create** macro node

## 4.2 Use of a Manager Instance

A user accesses a port in a manager by means of a statement such as

```
outdata := use (uMan.name^j, indata)        (4.4)
```

Expression **uMan.name**$^j$ is equivalent to **uMan[j]**; i.e. the component in structure **uMan** with index **j**. The argument **indata** is the input required from the user by the manager.

The **use** construct is implemented in the user flowgraph by a **Use** macro node, which replaces the **U** and **U**$^{-1}$ base language operators of Id {3}, as shown in Fig.5. It accepts the required input data, together with the colour **uc** and address **arg** of the manager entry being accessed, and eventually yields the result of the activation.

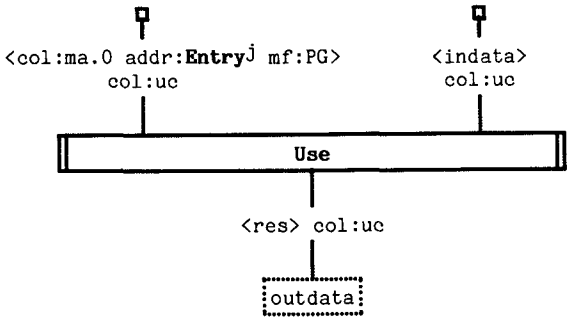


Fig.5 The manager user graph

**Use** is specified as

```
input_l = <col:ma.0 addr:Entry^j mf:PG> col:uc
input_r = <indata> col:uc
output  = <res> col:uc addr:outdata
                                            (4.5)
```

Fig.6 shows the expansion of this macro node. The **Store** macro node {4} saves a structure containing the input data and the address **r** of the **set ix** node. The colour and address of the node where the structure is saved are passed to the appropriate **entry** in the manager. The result returned by the manager to the **set ix** node has an index which is spurious to the user, and this is cleared before exiting.

## 4.3 Manager Entry and Exit

The interaction between a **Use** macro node and a port **name**$^j$ in a manager occurs via the pair of corresponding **Entry**$^j$ and **Exit**$^j$ macro nodes shown earlier in Fig.1. Each input to **Entry**$^j$ is used to retrieve the corresponding argument structure which is broken down into its two components; i.e. return information (colours and addresses) and input data. These are placed in corresponding positions in two separate structures, **RET**$^j$ and **X**$^j$, respectively, which are built during the lifetime of the manager. Each related use contributes an additional component to each structure. The body of the manager processes the data input structure, **X**$^j$, and eventually produces a result structure **R**$^j$. **Exit**$^j$ directs each component of **R**$^j$ to the appropriate node in the user flowgraph, using the information supplied by the corresponding component of **RET**$^j$.

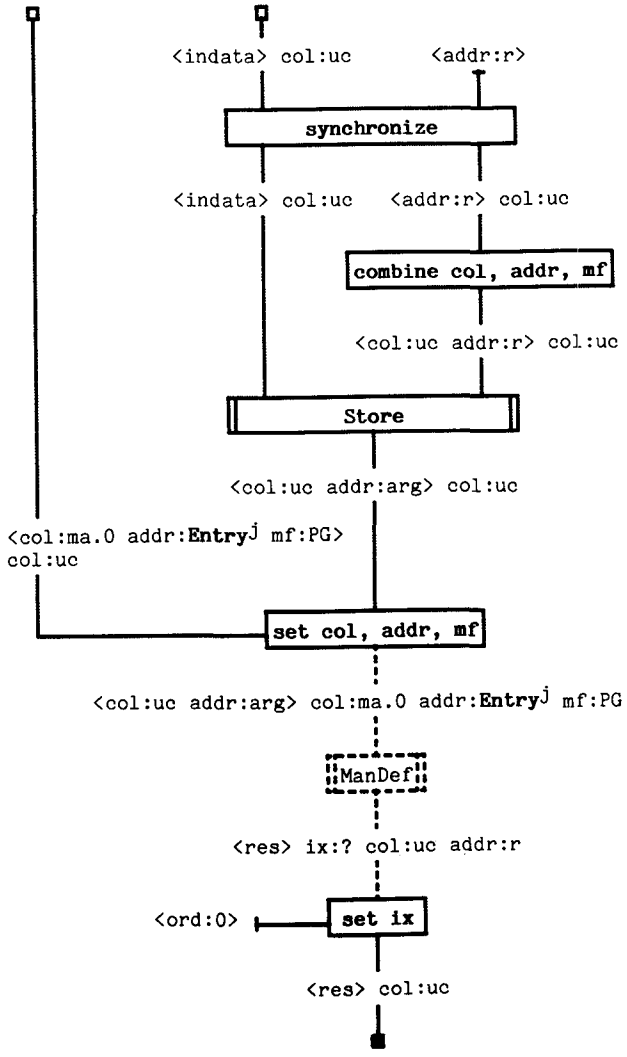


Fig.6 The **Use** macro node

The nondeterministic **Entry** macro (Fig.7) is specified as

$$\begin{aligned} \text{input} &= \{\langle \text{arg}_k\rangle \ \text{col:ma.0 mf:PG} \mid k{:}1\ldots\} \\ \text{output}_l &= \{\langle \text{col:uc}_k \ \text{addr:r}_k\rangle \\ &\qquad \text{ix:k col:ma.0} \mid k{:}1\ldots\} \\ \text{output}_r &= \{\langle y_k\rangle \ \text{ix:k col:ma.0} \mid k{:}1\ldots\} \end{aligned} \tag{4.6}$$

All inputs to an **Entry** have the same colour and must be separated before two-input nodes can be used without risk of a token clash. **Entry** creates a sequence from the requests it receives by setting their iteration levels to unique values. This is done at the **set il** node, which is initialized by **Set Up** (see Appendix) and safely accessed with {**increment, defer**} matching function.

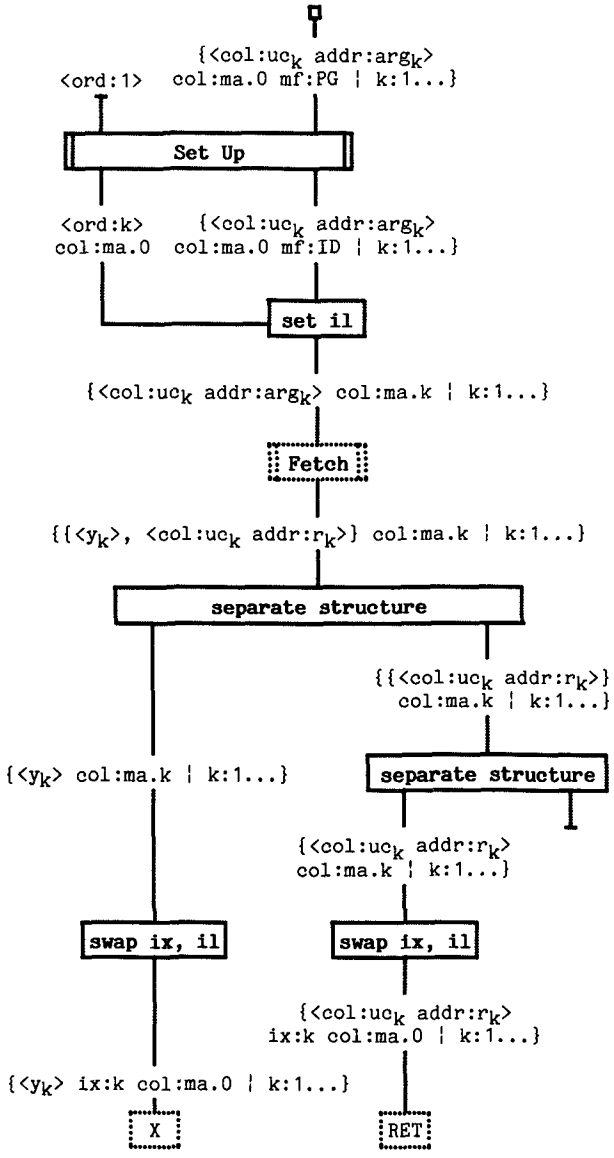


Fig.7 The **Entry** macro node

Once the requests are separated in this way, a **Fetch** macro {4} can be used to yield a sequence of argument structures. The first **separate structure** node yields a sequence of input tokens, which is converted into structure **X**, and a remainder structure. This is also separated, yielding a sequence of return colours and addresses, which is converted into structure **RET**.

The **Exit** macro simply returns the appropriate results. It is shown in Fig.8, and is specified as

$$\begin{aligned} \text{input}_l &= \{\langle \text{col:uc}_k \ \text{addr:r}_k\rangle \\ &\qquad \text{ix:k col:ma.0} \mid k{:}1\ldots\} \\ \text{input}_r &= \{\langle R_k\rangle \ \text{ix:k col:ma.0} \mid k{:}1\ldots\} \\ \text{output} &= \{\langle R_k\rangle \ \text{ix:k col:uc}_k \ \text{addr:r}_k \mid k{:}1\ldots\} \end{aligned} \tag{4.7}$$

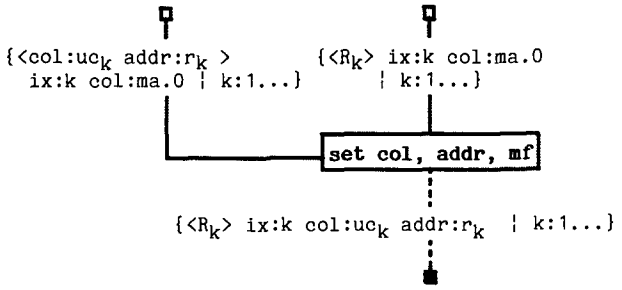


Fig.8 The **Exit** macro node

## 5 Conclusions

Id is a suitable high-level nondeterministic language for dataflow machines because it is based on parallel data driven semantics. Translation of programs is relatively straightforward, and the resultant object code makes good use of the parallel hardware. On the other hand, the Id notation is at a lower level than that of Communicating Processes {9}, and it involves the programmer in more tedious detail than may be necessary. This suggests that Id might be upgraded usefully, as long as this could be achieved without damaging the underlying data driven nature.

Experiments to evaluate the performance of this and earlier implementation models are now being carried on the Manchester prototype dataflow system emulator. However, the major test of these ideas will come when the hardware for the system is completed, and a pilot compiler has been written.

For the present, this work demonstrates that a dataflow system can accommodate both deterministic and nondeterministic computation in much the same way as a conventional machine, without resorting to specialized hardware and with the attendant benefits of parallel data driven execution.

Acknowledgements

We wish to acknowledge the stimulus provided by our colleagues in the Dataflow Research Group at Manchester. We are also indebted to Arvind and Keshav Pingali of M.I.T. for their helpful comments on, and sugestions for revision of, the original manuscript. Arthur Catto was supported by CAPES of the Ministry of Education of Brazil. Construction of the prototype dataflow computer is being funded by the Science and Engineering Research Council of Great Britain, under its Distributed Computing Systems Programme.

## Appendix

### A.1 The Set Up macro

Subgraphs are normally initialized by means of a trigger, which is generated by the system at the outermost flowgraph level and then propagated to all inner subgraphs. The Set Up macro (Fig.9) is an alternative approach, which initializes subgraphs on demand. Arguments $arg_k$ are directed with {preserve, generate} matching function to a branch on empty node whose control input port is unconnected. The subscripts refer to the arbitrary time order in which the arguments reach the branch node. By the {preserve, generate} matching action described in section 2.2, the first token is directed to the left output of the branch node to generate the initial value val, whilst subsequent tokens are directed to the right.

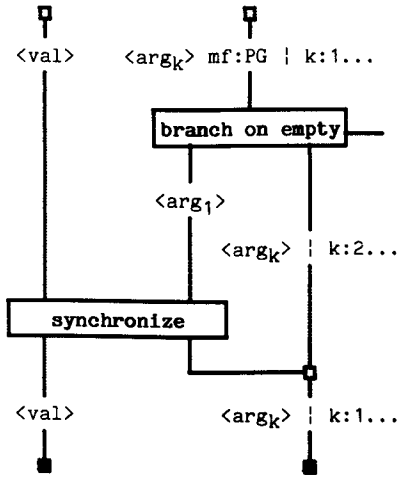


Fig.9 The Set Up macro node

References

{1} Arvind and K.P.Gostelow, A Computer Capable of Exchanging Processors for Time, Information Processing 77, North Holland, 1977, 849-853.

{2} Arvind, K.P.Gostelow and W.PLouffe, Indeterminacy, Monitors and Dataflow, Proceedings of the Sixth Symposium on Operating Systems Principles, November 1977, 159-169.

{3} Arvind, K.P.Gostelow and W.Plouffe, An Asynchronous Programming Language and Computing Machine, Department of Information and Computer Science, University of California, Irvine, December 1978.

{4} D.L.Bowen, Implementation of Data Structures on a Dataflow Computer, Ph.D. Thesis, University of Manchester, June 1981.

{5} P.Brinch Hansen, Distributed Processes: A Concurrent Programming Concept, Communications of the ACM, vol.21, no.11, November 1978, 934-941.

{6} A.J.Catto, Nondeterministic Programming in a Dataflow Environment, Ph.D. Thesis, University of Manchester, July 1981.

{7} A.J.Catto and J.R.Gurd, Nondeterministic Dataflow Graphs, Information Processing 80, North Holland, 1980, 251-256.

{8} A.J.Catto, J.R.Gurd and C.C.Kirkham, Nondeterministic Dataflow Programming, Proceedings of the Sixth ACM European Regional Conference, March 1981, 435-444.

{9} N.De Francesco et.al., On the Feasibility of Nondeterministic and Interprocess Constructs in Dataflow Computing Systems, Proceedings of the First European Conference on Parallel and Distributed Processing, February 1979, 93-100.

{10} J.R.Gurd, J.R.W.Glauert and C.C.Kirkham, Generation of Dataflow Graphical Object Code for the Lapse Programming Language, Lecture Notes in Computer Science, vol.111, June 1981, 155-168.

{11} J.R.Gurd and I.Watson, Data Driven System for High Speed Parallel Computing, Computer Design, vol.9, nos.6 and 7, June and July 1980, 91-100 and 97-106.

{12} C.A.R.Hoare, Communicating Sequential Processes, Communications of the ACM, vol.21, no.8, August 1978, 666-677.

{13} P.R.Kosinski, A Dataflow Language for Operating Systems Programming, ACM Sigplan Notices, vol.8, no.9, September 1973, 89-94.