USER REFERENCE MANUAL FOR TASK LEVEL DATA FLOW LANGUAGE: VERSION 1

Nicolas Graner & Jit Biswas

Department of Computer Sciences
University of Texas at Austin
Austin, Texas 78712

TR-86-05 January 1986

Table of Contents

List of Figures

1 Abstract

The language TDFL (task level data flow language) is in its formation stages. We have implemented a core of the language which we are certain about. We also have the tools with which to implement further primitive elements of the language as and when they are deemed necessary. This document defines the core of TDFL, its implementation and examples of its use.

2 Introduction

Given a computation graph, it is possible to traverse it in a manner such that each task executes asynchronously and the only constraints on scheduling are due to the dependencies between tasks as revealed in the graph. In this report we outline a simple graphical programming language called TDFL (Task Level Data Flow Language) that allows the expression of computations as task level data flow graphs. We also outline an implementation of TDFL on a dual processor Cyber 70 mainframe.

3 The Core of TDFL

Core here refers to a minimal set of operations that may be used to represent a significant class of computations.

The core of TDFL consists of **nodes** and **arcs**. **Nodes** are active elements of a computation. They can be of the following types.

- **General**
- **Initiator**
- **Terminator**
- **Loop**
- **Merge**

Arcs are directional FIFO channels that transport tokens, or data values between nodes. By including self loops (an arc leaving and entering the same node), we have an easy way of representing state retention. Associated with each arc is a positive number that denotes the integral number of words constituting a token for that arc. This number is fixed for the duration of the computation.

We briefly describe the actions of each of the above node types.

General nodes. These have any number of input and output arcs. The body is

executed if and only if a token is available on each incoming arc. After execution a token is placed along each output arc.

Initiator nodes. These have no input arcs and any number of output arcs. The execution of these nodes is spontaneous. With each execution a token is placed on each output arc. Typically a program contains only one initiator, although we do not make this a restriction.

Terminator nodes. These are orthogonal to Initiator nodes in that they have any number of input arcs but no output arcs. They execute only when they have a token on each input arc. Typically a data flow program contains only one terminator that accumulates and displays the results.

Loop nodes. A loop node has two input arcs, called the main and feedback input arcs, and two output arcs called the main and feedback output arcs (Fig 1). It also has a boolean predicate function that can be evaluated taking as argument a token that arrives on an input arc. Operationally a loop node behaves as follows. There are two distinct states that the loop node can be in - "expecting main input" and "expecting feedback input". Initially a loop node is in the state "expecting main input". It is activated by the presence of a token at its main input arc; the feedback input arc being ignored at this point. If the result of the predicate function applied upon the token is **true**, it places the token upon the main output arc and goes back to the "expecting main input" state. If the predicate evaluates to **false** it puts the token on the feedback output arc, ignoring the main output arc, and enters the state "expecting feedback input". Until the predicate becomes true, for each subsequent firing the loop node removes a token from its feedback input arc. As before once the boolean predicate is true, the loop node places the token upon the main output arc and goes back to the "expecting main input" state.

Merge nodes. A merge node has any number of input arcs and only one output arc. The node executes whenever a token is available on at least one input arc. The act of execution amounts to removal of a token from one of the input arcs (nondeterministically selected), and putting that token on the unique output arc.

4 Examples

We do not yet have a data flow language from which we can compile into graph representations. Thus the form of the graph is entirely in the hands of the programmer. It is hoped that the programmer will build "reasonable" graphs from these primitives. We now present some examples of TDFL programs.

Example 1. A triangular solver.

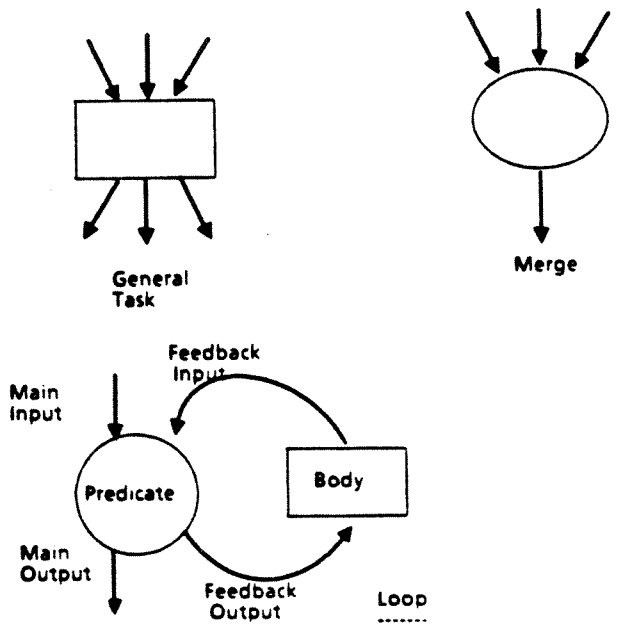

Figure 1: Primitive nodes of TDFL

Let Ax = b be a system of linear equations in n unknowns. A is an n by n coefficient matrix, x is an n by 1 vector of unknowns. b is an n by 1 vector of constants representing the right hand side (RHS) values. The algorithm for solving these equations, called Gaussian Elimination proceeds in two steps. The first step puts the system into upper triangular form by "zeroing" out or pivoting the subdiagonal elements of A, making appropriate changes to b. The second solves the revised system of equations by back substitution. Here we are interested in only the second step. We assume that we have a system of equations TX = b where T is an n by n lower triangular matrix, X is an n by 1 vector of unknowns and b is an n by 1 rhs vector. For example, say the first step puts a given 3 by 3 matrix into triangular form as shown below:

$$\text{Solve} \quad \begin{bmatrix} -1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 3 \end{bmatrix} \begin{bmatrix} X_1 \\ X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} -1 \\ 2 \\ 6 \end{bmatrix}$$

First we solve $-X_1 = -1$ from row 3. Substituting $X_1 = 1$ in row 2 gives us $X_2 = 1$. Substituting the values of X_1 and X_2 in row 1 gives $X_3 = 1$.

Parallel formulation. The parallel formulation we use is shown in Fig 2. Let k be a factor of n. Let $s = n/k$. A k way partition of the system corresponds to $k(k+1)/2$ subblocks of T and k subvectors of X and b. Each diagonal subblock of T is a lower triangular matrix of side s. Each non diagonal subblock is a square matrix of side s.

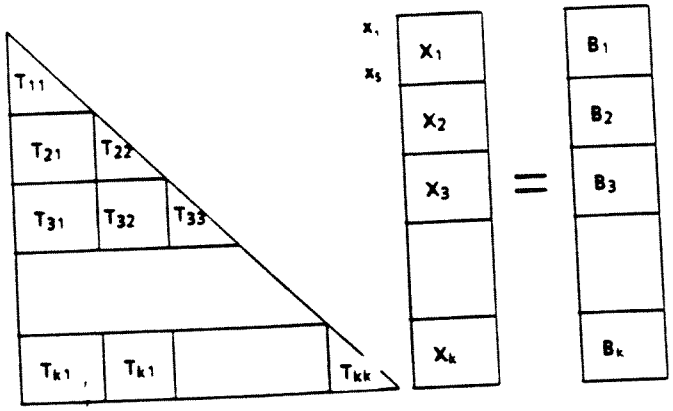

Figure 2: Partitioning of back substitution.

Notations. Let T_{ij} denote the subblock on the ith row and jth column of the partition. Let X_i denote the ith subvector of unknowns $[x_{(i-1)*s}+1, \ldots x_{(i*s)}]$. Similarly B_i denotes the ith subvector of rhs values.

Strategy. The basic idea is to reduce each row of blocks into a set of s equations in s unknowns. The first row $T_{11} X_1 = B_1$ is already in this form and the solution of this can be assigned to a sequential task P_{11}. The equations represented by the ith row of blocks can be written as:

$$\Sigma^{i}_{j=1} \; T_{ij} X_j = B_i \ldots \text{(i)}$$

Clearly, as soon as X_1 has been evaluated by P_{11} we can assign to P_{i1} the task of removing these unknowns from equations (i) by carrying out the transformations:

$$\Sigma^{i}_{j=2} \; T_{ij} X_j = B_i - T_{i1} X_1$$

The new set of equations are:

$$\Sigma^{i}_{j=2} \quad T_{ij} X_j = B_i'$$

where B_i' is the new sub vector of rhs values. After this transformation the 2nd row of blocks now represents a set of s equations in s unknowns $T_{22} X_2 = B_2'$ and can therefore be solved by task P_{22}. Values of X_2 thus evaluated can be substituted in parallel by tasks $\{P_{i2} : 3<=i<=k\}$ to transform the remaining equations by removing the unknowns X_2.

In short all we have done is to extend the most obvious algorithm of backwards substitution to blocks instead of unit variables. If k=1, the algorithms are identical. The computation graph resulting from the above mentioned task assignments is shown in Fig 3.

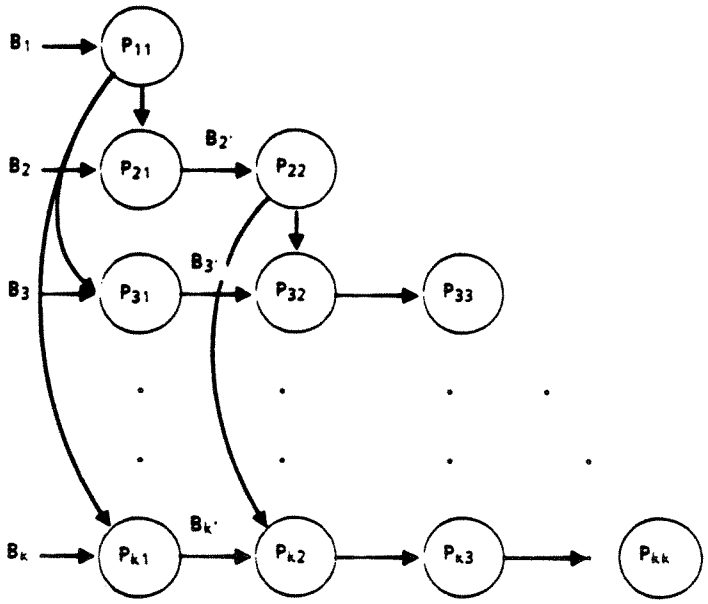

Figure 3: Computation Graph for back substitution.

Note: P_{ij} has subblock T_{ij} of coefficients. Task P_{i1} takes as inputs the original vector B_i from the environment. P_{ii} determines the values of X_i and returns it to the environment.

Example 2. A partitioned iterative problem.

Say we have a problem in which a certain computation is performed upon equal partitions of an array (Fig 4). At the end of the compute phase neighbouring partitions exchange their results. This scheme is repeated a number of times after which the computation terminates. The problem is formulated for two partitions in TDFL as follows (Fig 5).

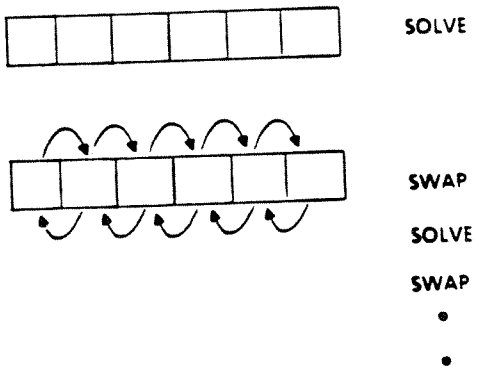

Figure 4: Partitioning for example 2.

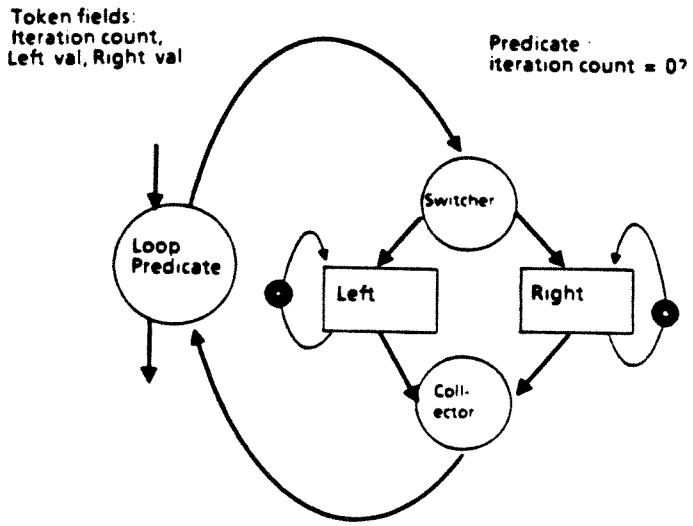

Figure 5: Formulation for example 2 with two partitions.

The dots on the self loops for LEFT and RIGHT indicate initial state (i.e. the partition of the array) and the self loops themselves indicate state retention. A token has three fields, a count field a left and a right value. The Switcher node swaps the left and right values of a token. The Collector node builds a new token from the left value field of its left input, the right value field of its right input and the count value field (of either input) decremented by 1.

Example 3. Airline reservation

The third example illustrates the use of a merge node. S_1 and S_2 are booking agents that receive replies to requests for seats in an airplane. A request is of the form:

```
Source id
Number of seats
```

Replies may be one of the following two forms:

```
Destination id          |   Destination id
Done                    |   Failed
```

The booking agents scan the incoming stream of replies, throwing away tokens that do not apply to them (Fig 6).

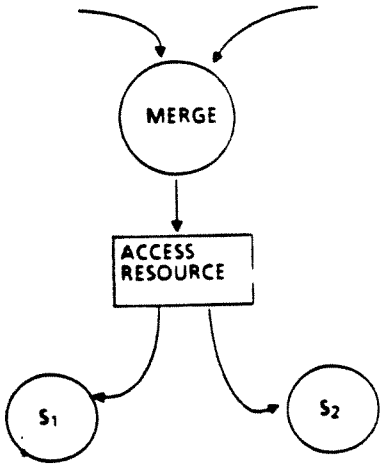

Figure 6: Airline reservation.

5 Implementation

Overview. The system consists of an "initializer", which takes a description of the program graph and starts the asynchronous execution of the nodes of the graph. Once each task (node) has been started, the initializer is no longer needed. The initializer uses a set of predefined files, which contain prototype encapsulations written in Pascal. There is one such encapsulation for each of the node types in Section 3. The initializer "plugs in" the user written code into these encapsulations compiles the resulting files and forks processes under the UT-2D operating system.

1. Task table:

The task table is stored in a user named file. It includes one line for each task; each line has three or four fields separated by one or several spaces:

- **task index:** an integer that uniquely identifies the task. Indices must run sequentially from 1 to N, otherwise an error is detected by the initializer.

- **task category:**

```
1 = general
2 = initiator
```

```
3 = terminator
4 = merge
5 = loop
```

- **body name:** name of the file containing the source code of the body (1 to 7 letters and/or digits). This file should conform with the format described below, and be in the pfset when the program is started. Several tasks may use the same body (they will each get a different copy of it). The system may be extended in the future to accept task bodies either in source or object form.

- **input file name: (optional)** This file will be assigned to the standard identifier INPUT when the task is executed, so it can be read without specifying a file name. Several tasks may use the same input file. The option is selected by preceding the filename with the character 'I'.

- **FORTRAN file name: (optional)** This feature allows some procedures of the task to be written separately in FORTRAN and separately compiled. At runtime the binary file named is linked to the rest of the program. Several tasks may use the same input file. The option is selected by preceding the filename with the character 'F'.

2. Channel table:

The channel table must be in a separate user file. It includes one line for each channel, in any order (since channels are not globally indexed); each line has seven fields separated by one or more spaces:

1. **channel name:** a string of 1 to 10 characters that uniquely identifies the channel. This name is not used by the data flow system, but is stored in ESM with the channel and can be used for debugging.

2. **index of source task:** the index (see "Task table") of the only task that will be able to send to this channel.

3. **index of channel in source task:** all the output channels of one task are numbered sequentially from 1 to nbout. This index uniquely identifies this channel among the output channels of its source task.

4. **index of destination task:** the index (see "Task table") of the only task that will be able to receive from this channel.

5. **index of channel in destination task:** all the input channels of one task are numbered sequentially from 1 to nbin. This index uniquely identifies this channel among the input channels of its destination task.

6. **size of tokens:** tokens transmitted along this channel will all be of type **array[0..size] of integer** , i.e. they will in fact contain one more word than requested by this parameter (word 0 is used as a tag by the system, but is also accessible to the user).

7. **number of buffers:** this is the maximum number of tokens that can be sent to the channel before any is received (actually, the system will reserve one more buffer in order to implement a FIFO). Usually, the value of this parameter won't affect significantly the performance of the system.

3. The user written task bodies:

The function actually performed by a task is determined by its "body". This is a procedure that is called as soon as a token has been received on every input channel (on one input channel for a merge). It receives these tokens as input parameters and returns new tokens to be sent along output channels.

Each body is contained in a separate file (see "Task table" above) with the following format:

- a line consisting of the following word starting in column 1.

  BODY

- a procedure declaration, which is different for each category of tasks:

  - general -- procedure body(var x : inset; var y : outset);

  - initiator -- procedure body(var y : outset; var b : boolean);

  - terminator -- procedure body(var x : inset);

  - loop -- procedure body(var x1, x2 : intoken; var y1, y2 : outset);

  - merge -- procedure body(var x : intoken; var y : outset);

- [1] **The procedure proper:** It may use any constant, type or global variable and call any function defined in the encapsulation (but not modify a variable!). Of special interest are the following:

---

[1] formal parameter names may be different, only the types are required; these types need not be defined in the body, they are globally defined as:

type intoken = array[0..maxinsize] of integer; outtoken = array[0..maxoutsize] of integer; inset = array[1..nbin] of intoken; outset = array[1..nbout] of outtoken;

```
const null : a token t is a null token if and only if
             t[0] = null
      default : value given to the tag of all output
                tokens before body is called. Therefore
                the tag need not be assigned a value in
                the body..
      nbin  : number of input channels for this task
      nbout : number of output channels for this task
      maxinsize  : maximum size of input tokens for
                   this task
      maxoutsize : maximum size of output tokens for
                   this task

        (note that every token is passed as an array of
        size maxinsize or maxoutsize, but only the first
        k elements are meaningful, where k varies with
        each channel and is supposedly known to the body
        beforehand)

var   nulltoken, killtoken : outtoken
          predefined output tokens

procedure intorel( inp : integer; var out : real    );
      integer to real conversion
procedure reltoin( inp : real   ; var out : integer );
      real to integer conversion
procedure intoalf( inp : integer; var out : alfa    );
      integer to alfa conversion
procedure alftoin( inp : alfa   ; var out : integer );
      alfa to integer conversion
```

4. Initiator.

The initiator must fire a number of times, then send killers to all its output channels. In order to do this, a extra boolean variable is provided to the body. If after executing the body this variable has the value 'true', output tokens are sent normally. If it has the value 'false', the value of the output tokens is ignored, killers are sent and the initiator terminates.

This implies that the body has some way to keep a global state (so that each execution does not return the same value). In a batch environment, this will normally be achieved by reading data from an input file. Thus a typical body for an initiator will have the following structure:

```
BODY
procedure body(var y : outset; var b : boolean);
```

```
begin
   if eof then
      b := false
   else begin
      b := true;
      ...
      read(data);
      ...
      y[i][j] := ...
      ...
   end;
end;
```

5. Generated files:

The following files are generated and saved in the pfset during execution (some of them may be missing if the program terminates abnormally); other temporary files are created but not saved.

Files associated with the initializer:

- **NICLINI:** Listing of the initializer generated by the Pascal compiler
- **NICOUT:** Output from the initializer
- **NICDAYF:** Dayfile for the job that ran the initializer

Files associated with each task ('i' represents a task index:)

- **NICOi:** Output file from task i. Contains output from the body, the encapsulation (to help debugging) and possibly the operating system (in case of error).
- **NICDi:** Dayfile for the job that ran task i. If the job terminates abnormally (inexistent file, compile time error, run time error...) an explanation will usually be found in this file.
- **NICSi:** Source code for task i. Identical to NICGEN, NIC1ST, NICTER or NICMER but with the appropriate values 'plugged in'. The body is not included, only a directive to instruct the compiler to include it is.
- **NICLi:** Listing generated by the Pascal compiler. Includes the source of the body.
- **NICCi:** Command file for the job that reads, compiles and executes task i.

Further details of the implementation can be found in the appendix.

6 Running the system

Let us consider the triangular solver problem on the following data:

$$A = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 2 & 7 & 0 & 0 & 0 & 0 \\ 3 & 8 & 12 & 0 & 0 & 0 \\ 4 & 9 & 13 & 16 & 0 & 0 \\ 5 & 10 & 14 & 17 & 19 & 0 \\ 6 & 11 & 15 & 18 & 20 & 21 \end{pmatrix} \qquad b = \begin{pmatrix} 1 \\ 9 \\ 23 \\ 42 \\ 65 \\ 90 \end{pmatrix}$$

We shall partition three ways. This gives rise to six subproblems, three of which are diagonal SOLVE tasks and three are non-diagonal MATVECT tasks.

The problem consists of a static graph (Fig 7). We convert this graph into a channel table and a task table in separate files. Once we have decided the file names we incorporate them in a file called NICSUB that is the batch submit file for the system. Nothing else needs to be changed.

The task table. (file named JITTASK)

```
1    1 JITSOLV I FIL1
2    1 JITMATV I FIL2
3    1 JITMATV I FIL3
4    1 JITSOLV I FIL4
5    1 JITMATV I FIL5
6    1 JITSOLV I FIL6
7    2 JITINIT I FIL7
8    3 JITTERM
```

The third column consists of task bodies. These are Pascal procedures that we would have written for the corresponding partitions of the triangular solver. Since there are no FORTRAN procedures we do not use the F option.

The channel table. (file named JITCHAN)

```
chan1   1  1   2 1     2    2
chan2   1  2   3 1     2    2
chan3   2 1    4 1     2    2
chan4   3 1    5 2     2    2
```

```
chan5   4   1   5 1     2    2
chan6   5 1     6 1     2    2
chan7   7     1   1 1     2    2
chan8   7     2   2 2     2    2
chan9   7     3   3 2     2    2
chan10   1     3   8   1     2    2
chan11   4     2   8   2     2    2
chan12   6     1   8   3     2    2
```

The next step is to edit a file called NICSUB and incorporate the names of the above files in place of JITTASK and JITCHAN.

The submit command is:

SUBMIT CC=NICSUB TM=100

This command starts a task that looks into the task table and forks all the tasks in serial order starting at the top of the table. For each task we maintain (in other words save on disk) the following files:

1. The source pascal program created (eg NICS1). This includes some additional routines of ours.

2. The submit module that was utilized by NICSUB to get the task started (eg NICC1).

3. The dayfile for the task (eg NICD1).

4. The listing file for the task (eg NICL1). (It is useful only in helping us catch compilation bugs. In content it is really no different from 1).

5. The output file of the task (NICO1).

7 Future work

Critical examination.

Our major design goal was to make the system modular and self-contained. Application dependent elements have been constrained to a few files clearly separated from the system proper, making it easy to write/modify an application with little or no knowledge of the environment (this is consistent with the concept of task in CSL).

The user has a lot of freedom in using the system. For instance, each token carries a "tag" that can take any integer value. A few predefined values are known to the system

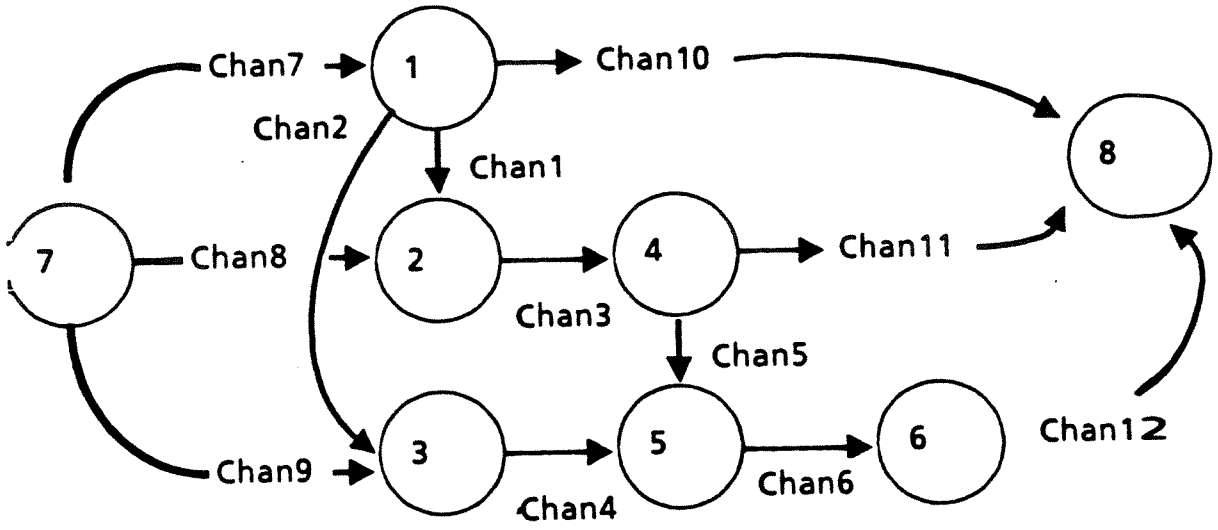

Figure 7: Node and edge labels for sample run.

(default, null, kill), but the user can define any value for his own usage, such as loop control (for which no standard mechanism is provided). Of course, freedom implies responsibility, and the system won't be able to catch most programming errors involving user-defined features.

We also avoided requesting redundant information from the user (each task or channel feature is described only once). This of course doesn't allow for error detection/recovery, which may be a problem as long as our applications are written manually, but should make things easier for a compiler.

What could be improved.

We tried to make the system as machine-independent as possible. However, some peculiarities of the UT-2D operating system required special treatment. In particular, the need to transfer files to and from the pfset and the necessity to write command files in order to make system calls are highly non-standard. We tried to constrain those problems to low-level functions, but porting the system to another machine would still involve some nontrivial recoding.

In order to keep the system separated from user-defined tasks, the interface has been simplified to a minimum. This means that each tasks receives as input a array of tokens, each element of which is an array of integers received from a given channel. Therefore, each element can be referenced only by a pair of indices (channel number and address in the token) and not by a symbolic name. It may be awkward for the user to write the task bodies with this constraint, but we don't see any other solution until we can use a data flow language and let the compiler do the mapping from names to indices.

For the same reason, tokens cannot have any data type other than array of integers. Conversion functions (reltoin, intorel, alftoin and intoalf) are available to the task body, but have to be called explicitly when needed.

Appendix

Standard files.

The following files are always present in the permanent file set and are independent of the particular application being executed:

- **NICINIT:** source code for the initializer.
- **NICGEN:** prototype encapsulation for a general node.
- **NIC1ST:** prototype encapsulation for a initiator node.
- **NICTER:** prototype encapsulation for a terminator node.
- **NICLOO:** prototype encapsulation for a loop node.
- **NICSUB:** command file to run the program in batch mode.

Application dependent files.

These files describe the application to be executed. They should be generated automatically from a high-level description in some appropriate language, but are currently hand-written.

1. **NICTASK:** The task table, includes one line for each task with the following information:
   - task index (an integer that uniquely identifies the task).
   - task category (an integer from 1 to 5, as defined earlier).
   - body name (the name of the file containing the source code of the body).
   - optionally, an input file name (this file will be assigned to the standard identifier INPUT when the task is executed).
   - optionally, a FORTRAN (binary) file name.
2. **NICCHAN:** The channel table consisting of one line for each channel which indicates:
   - channel name (1 to 10 characters)

- index of source task.
- index of channel in source task (an integer that uniquely identifies the channel among all the output channels of the source task).
- index of destination task.
- index of channel in destination task (an integer that uniquely identifies the channel among all the input channels of the destination task).
- size of tokens in the channel (number of words per buffer).
- number of buffers.

3. **Bodies:** One file for each task containing the source code of its body. Each of these files should have a first line containing only the word BODY, followed by the declaration and code of a procedure called body. The declaration varies with each category of task.

4. **Optional:** An input file for some or all of the tasks (an output file will be automatically created by the system as needed). Usually, only the initiator(s) will need an input file. A FORTRAN generated binary file may also optionally be specified.

Channel Implementation.

Channels are implemented using shared memory (ESM). A channel consists of a descriptor and a set of buffers organized as a FIFO. The descriptor contains the following items:

- head pointer (to first token in channel).
- tail pointer (to first free buffer).
- number of words per buffer.
- number of buffers.
- channel name (alfa stored as an integer).

A channel should only be accessed by two tasks, a reader and a writer, though nothing prevents other tasks from accessing it as well. There may be several channels from one task to another, or even from one task to itself (in the latter case the task should be a merge, otherwise it is automatically deadlocked).

All commands required to run a data flow program are in a file which is submitted in batch mode. (See example in section 4). The initializer is read, compiled and executed. It creates the source code for each task, saves them in the pfset, creates command files and forks the corresponding processes. Each command file, when executed, reads the source of a task, compiles and executes it, and saves its output (as well as the compiler-generated listing, dayfile and other files to allow debugging).

A note on null and kill tokens and the merge node.

Every token carries a 'tag' which can take any integer value. However, two particular values are recognized by the system: 'null' and 'kill'.

A third value, 'default', is defined but doesn't have any special meaning. It is the value given to an output token if the task body doesn't set it explicitly.

The meaning of 'null' is: if the tokens received on **all** incoming channels are null, the body is not executed and a null token is sent to every output channel. If at least one token is non null, the body is executed and gets all the tokens (null and non null) as arguments.

Any task may generate null tokens as output. This is particularly useful to implement conditionals. The meaning of 'kill' is: if **at least one** token received is a 'killer', the body is not executed, a 'killer' is sent to every output channel and the task terminates. In principle, when a killer is received, all other tokens received at the same time should also be killers, but this is not checked for by the system. Sending a token to a task that has terminated yields unpredictable results.

A killer is generated by the initiator after a number of executions (e.g. when there is no more input to be read) and propagates through all the tasks to ensure complete termination. Any task may generate killers but this will usually not be needed.

Specification of merge nodes.

A merge node behaves very much like a regular (general) node, except that it need not receive a token on every input channel to fire. It may appear anywhere in a data flow graph, and it is the user's responsibility to ensure that the program is sound, deadlock free and meaningful.

Specifically, a merge node executes the following algorithm:

1. select an input channel according to some scheduling policy (currently plain round-robin).

2. poll the channel to see if it contains a token. If it is empty, go to 1. (unlike a regular node, a merge executes a non-blocking receive).

3. get a token from the channel. If this token is:

   - **regular (neither null nor kill)** execute the body, which produces one output token for each output channel. Send these tokens and go to 1.

   - **null** don't execute the body. Send a null token to every output channel and go to 1.

   - **kill** don't execute the body, don't send any token. Mark the channel from which this was received as 'inactive', and don't try to receive from it subsequently. When all input channels are 'inactive', send a killer to every output channel and terminate. Otherwise go to 1.

A merge node following these specifications has been implemented.