HyperDEBU: a Multiwindow Debugger for Parallel Logic Programs

Hidehiko TANAKA, Jun-ichi TATEMURA
{*tanaka*, *tatemura*}@mtl.t.u-tokyo.ac.jp

Department of Electrical Engineering,
Faculty of Engineering, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 JAPAN

Abstract. In this paper, a multiwindow debugger HyperDEBU for fine-grained highly parallel programs is presented. The target language of HyperDEBU is Fleng which is one of the committed choice languages. This debugger supports many kinds and levels of views of programs, and helps user to locate bugs efficiently.

Keywords: Programming Environments; Testing and Debugging; Concurrent Programming

1 Introduction

Debugging parallel programs is much more difficult than debugging sequential programs because many processes run simultaneously and interact with each other. While most traditional debuggers only trace each process, they can hardly handle inter-process communication. To comprehend inter-process communication, some prototype systems have been developed : a time-process diagram and an animation of program execution [1]. The time-process diagram represents execution of a program in a two-dimensional display with time on one axis and individual process on the other axis. The animation displays snapshots of processes one after another. However, these are not enough for the debugging of parallel programs.

Though a Committed-Choice Language (CCL), which is a kind of parallel logic programming languages, enables the description of fine-grained highly parallel programs, it is very hard to trace many small processes in a CCL program. The time-process diagram cannot easily display numerous flows of data and control. It is hard to understand an animation which displays many processes created dynamically. To understand execution of a fine-grained highly parallel program such as a CCL program, the relation among control/data-flows is more important than time-sequences of events.

The role of debugger is to present a user the program execution status in the form of model. The user observes and controls the program execution through the model, and finds out bugs through comparing it with his intension.

In this paper, we introduce a communicating process model to represent the execution of a CCL program, and describe a multiwindow debugger HyperDEBU which implements this model on multiwindow interface. This debugger provides flexible views which enable a user to examine and manipulate complicated structures composed of multiple control/data flows of a fine-grained highly parallel program.

2 Committed-Choice Language Fleng

Committed-Choice Languages (CCLs) such as Guarded Horn Clauses (GHC) and Concurrent Prolog are parallel logic programming languages which have a control primitive "guard" for synchronization [4]. Fleng [2] is a CCL designed in our laboratory. We are developing the Parallel Inference Engine PIE64 [3] which executes Fleng programs. Fleng is simpler than other CCLs in the sense that Fleng has no guard goal, though guard mechanism is incorporated in the the head part.

A Fleng program is a set of horn clauses like:

$$H{:-}B_1, \cdots, B_n. \quad n \geq 0$$

,where H and B_i are predicates called goals.

The side to the left of :- is called the*head part*, and the right side is called the *body part* whose item B_i is called a *body goal.*

Execution of Fleng program is repetition of rewriting a set of goals in parallel. For each goal, one of the clauses whose head can match with the goal, is selected, and then the goal is rewritten by the body goals of the clause. The execution begins when initial goal is given, and is completed when no goal remains. The rewriting operation is called *reduction*, and the matching operation is called *unification.*

Unification is an algorithm which attempts to substitute values for variables such that two logical terms are made equal. To realize communication and synchronization in concurrent logic programs, unification in CCL is divided into two classes : *guarded unification* and *active unification.* Guarded unification is applied in the head part of a clause and variables in the goal are prevented from being substituted. Such unification is *suspended* until these variables have values. Active unification is applied in the body part of a clause and is able to substitute values for variables of goals.

The synchronization mechanism of guarded unification prevents reading a variable before it is bound to a value and eliminates many nondeterministic bugs due to synchronization.

For example, assume that the following program is given :

```
init :- send(X),receive(X).                    (1)
receive([D|S]) :- do(D), receive(S).           (2)
send(S) :- S = [0|S1], send(S1).               (3)
```

If an initial goal `init` is given, this is unified with the head of the clause (1) and two goals `send(X)` and `receive(X)` , which have a shared variable `X` , are generated by reduction. When the goal `receive` is tried to unify with the head of clause (2), it is found that the argument of `receive` is required to be a structured data `[D|S]` (it is called *list cell*) in order to complete the guarded unification. Then this reduction of `receive` is suspended if its argument is a variable at this time. On the other hand, the goal `send` is unified with the head of the clause (3), and reduced into two goals `S = [0|S1]` and `send(S1)`. The goal `S = [0|S1]` is a system predicate for active unification which unifies two terms `S` and `[0|S1]`. After this unification, the guarded unification of the goal `receive` and the head of (1) can be completed, and then the reduction of `receive` is resumed and two goal `do` and `receive` are generated.

When a Fleng program is executed, many goals are reduced in parallel, synchronizing each other in the way described above.

Fleng is a fine-grained highly parallel programming language in which goals are executed in parallel. Since many goals are created and terminated dynamically, it is very difficult to examine and manipulate execution of a goal.

3 Modeling Execution of Fleng Program

3.1 Requirements of the Model of Program Execution

The role of a debugger is to show users a model abstracted from the execution of a program. A programmer examines and manipulates the execution of the program through this model, compares the model with an intended model, and finds bugs from the difference between them. In this section, some requirements of a model which represents execution of a CCL program are described from the following two aspects.

Parallel Logic Program Since a synchronization primitive "guard" makes a difference of semantics between CCL and a pure logic program language, some operational meaning must be added to the declarative semantics for CCL. The problem with the semantics has been discussed in several works [5] [6]. We must consider the causality relation between input and output of a program. Accordingly, a model which represents execution of CCL program must take account of the input-output causality.

Fine-Grained Highly Parallel Program A model which represents execution of a fine-grained highly parallel program is required to abstract essential data from numerous information on execution of a highly parallel program, and to assist a user, who can deal with only a limited amount of information, to find a bug from it. Such a model is expected to have the following features :

- flexible levels of abstraction
- flexible aspects of abstraction

3.2 Process Model

We model the execution of a Fleng program using processes which communicate each other. In this section we describe the process model for execution of a Fleng program.

Notion of Process for CCL The conventional notion of process for CCL is associated with one goal or one sequence of goals. The process model proposed in this paper is equivalent not to one goal but to all of its subgoals generated by reduction. From outside, the execution of the process is looked upon as the input-output behavior of the process. The substance of the process is a set of goals derived from a goal. Since a goal is reduced to sub-goals, a process can be divided into sub-processes. It corresponds to execution tree which represents a computation of a logic program. The tree has subtrees whose roots are the nodes of the tree. The tree corresponding to the process can be divided into subtrees which corresponds to the sub-processes of the process.

In the process model described above, the definition clause is considered to define a relation between a process and its sub-processes.

Definition of Process The process model described above is formalized as follows. Our debugger provides this model to programmers.

Let G be a goal which is computed in a Fleng program and Q be a set of goals which are derived from G. We call the process P whose substance is Q "*the process with respect to* G", and call G "*the topgoal of* P".

The process P with respect to G is represented as follows from outside.

$$\langle G_{skel},\ I/O,\ S,\ G_{ins} \rangle$$

G_{skel} is the skeletal predicate of G whose arguments are replaced by distinct variables like this :

$$p(v_1, \ldots, v_i)$$

, where $v_1, \ldots, v_i$ are the distinct variables which can be regarded as the ports for communication with outer processes.

I/O is I/O data flow which represents the communication of the process P. They show how the variables of G_{skel} are substituted by unifications outside and inside of the process. Details of I/O data flow will be described later.

S is the state of P, which is one of the following states:

- *terminated*
  indicates that all of the goals in P have terminated.
- *suspended*
  indicates that there are no active goals in P and that there are some suspended goals. When one of the suspended goals is activated by input data which is substituted by unifications outside of P, the state turns into *active*.
- *active*
  indicates that some active goals exist in P and that P can be executed.

The above definition makes the model suitable for running programs or a program which runs perpetually, as this model is procedural rather than declarative.

G_{ins} is the *instance* of G. It is regarded as the result of the substitution for G_{skel} by I/O.

Input/Output of a Process I/O data flow of a process is represented as a tree structure which we call an *I/O tree*. This is a tree of causality among active unifications and guarded unifications.

Consider the I/O tree of a process with respect to a goal G. Since the arguments of G_{skel} can be regarded as ports for communication, there are I/O trees corresponding to each port.

The construction of I/O tree O can be represented as follows :

$$
\begin{aligned}
O ::= & \; C \text{ '|' } O \\
| & \; U \text{ '}\times\text{' } O \\
| & \; O \text{ '+' } O \\
| & \; \textit{'Nil'}
\end{aligned}
$$

Each factor has the following meaning respectively :

1. $C|O$
   This indicates input-output causality; O exists if input satisfying the condition C exists. C consists of guarded unifications. It is also graphically represented as:
   ```
   cond(Condition)
    |- O
   ```
2. $U \times O$
   This indicates that the active unification U exists and is followed by O. It is also graphically represented as:
   ```
   T1 = T2
     |- O
   ```
   where `T1 = T2` is an active unification and `O` is the I/O tree which substitutes variables in the term `T2`.
3. $O + O$
   This indicates that the path of I/O is divided. It is also graphically represented as:
   ```
   |- O1
   |- O2
   ```

An Example of I/O tree A horn clause describes not only the relation between a process and its sub-processes, but also describes the relation between I/O tree of the process and I/O tree of the sub-processes. I/O tree is defined recursively by the clauses committed (selected) in the execution of the program. Detail of the definition is described in [7]. In this paper, we will not explain the details but show a simple example of I/O tree.

Assume that the following program and an initial goal `append([1,2],[3],R)` are given.

```
append([H|X],Y,Z) :- Z = [H|Z1], append(X,Y,Z1).
append([],Y,Z) :- Z = Y.
```

This program connects a list given in the first argument and a list given in the second argument, and returns it through the third argument. The result will be `R = [1,2,3]`.

Then we will show I/O tree of the process with respect to the initial goal. G_{skel} of the process is represented as $append(A, B, C)$. The output of C is represented as the following I/O tree :

```
cond(A = [D|E]) <= (A = [1,2])
  |- C = [D|F]
      |- cond(E = [G|H]) <= (E = [2])
          |- F = [G|I]
              |-cond(H = []) <= (H = [])
                  |- I = B
```

This meaning is as follows : The active unification `C = [D|F]` exists on condition that the guarded unification `A = [D|E]` succeeds. There is the subsequent I/O tree to substitute variable `F` in the term `[D|F]`. `F` is unified with `[G|I]` on condition that the guarded unification `E = [G|H]` succeeds.

We can also abstract the I/O tree by replacing its subtree with its subprocesses as follows :

```
cond(A = [D|E]) <= (A = [1,2])
  |- C = [D|F]
      |- F / append(E,B,F)
```

`append(E,B,F)` is the subprocess of `append(A,B,C)` and unifies `F` by the inner unification of it.

3.3 Features of Process Model

The process model proposed in the previous section has the following features :

- hierarchy of processes
  A process consists of some subprocesses.
- multiple views of a process
  A process has three views; a set of goals, an execution tree, and I/O causality.

Figure 1 shows the process model. This model satisfies the requirements for a model of a fine-grained highly parallel program. The other definitions of process such as one goal and one sequence of goals do not seem applicable to debugging a highly parallel program.

Representing input-output causality, the process model also satisfies the requirements for the semantics of CCL. We have applied this model to *algorithmic debugging* of Fleng programs [7].

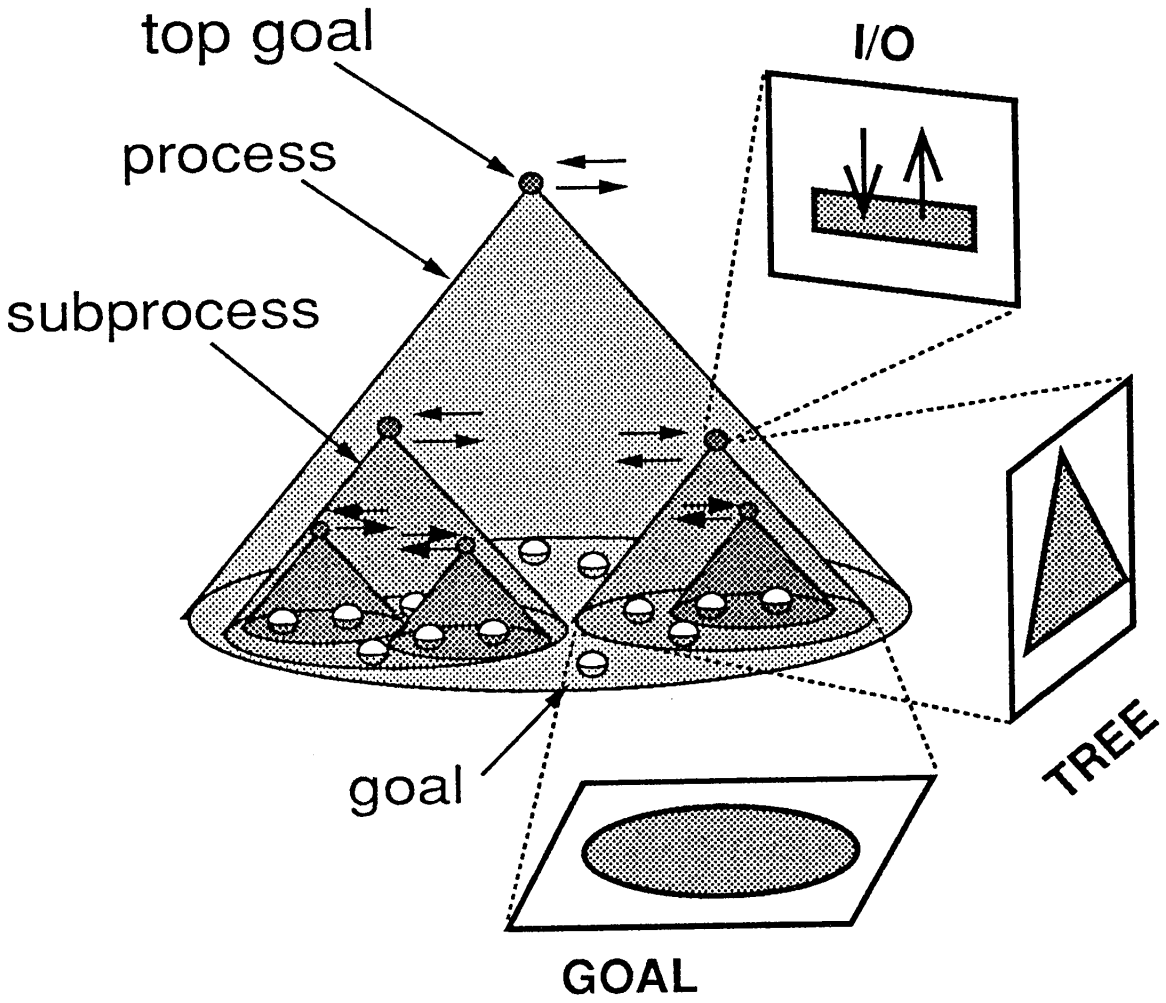

Fig. 1. The Process Model

4 Multiwindow Debugger HyperDEBU

4.1 Design of Debugger for Fine-grained Highly Parallel Programs

A sequential program has only one thread of execution, which can be debugged with a sequential interface. On the other hand, a parallel program has multiple complicated control/data flows which are considered to be high-dimensional information. If a sequential interface is used to debug a parallel program, the bottleneck between a programmer and the program makes it difficult to examine and to manipulate the execution of the program. Therefore, a high-dimensional interface is necessary to debug a parallel program.

Since a user compares a model represented by a debugger with the expected behavior of the program when he/she debugs a program, the debugger must provide a view of the kind he/she wants. Accordingly, the debugger must provide views which have flexible levels and aspects of abstraction.

Most of conventional multiwindow debuggers use many windows, each of which is assigned to one of the processes as a sequential debugger. However, high-dimensional information cannot be handled well in this way. On the other hand, HyperDEBU provides a user with a few kinds of windows he/she wants, through tracing links on a window which displays some information of a program execution.

HyperDEBU consists of the following windows.

1. toplevel-window
2. process-windows
   (a) TREE view (b) I/O tree view (c) GOAL view
3. structure-windows

Figure 2 shows an overview of HyperDEBU.

4.2 Requirements of The Debugger

In this section, we will discuss some requirements of the debugger design in accordance with the concept described in the above section.

Comprehending the Global Situation of the Execution When a program shows unexpected behavior, the first task for the programmer to debug it is comprehending the situation of the execution of this erroneous program. A programmer can debug a sequential program by tracing one thread of execution through some event filters. On the other hand, it is very difficult to trace a complicated structure composed of a large number of control and data flows of a highly parallel program. Even if he tries to trace some threads of these flows through an event filter, it is a hard problem to determine what should be extracted from such an enormous amount of data, and, furthermore, he can not comprehend their relationship with each other. That is to say, it is very much more difficult to comprehend a global situation of the execution of a highly parallel program than of a sequential program.

To solve this problem, a debugger is required to provide a macroscopic view abstracted from the information of the execution in order to help a programmer to comprehend the situation of the erroneous program.

Zooming on the Location of the Bug After the user identifies the area where bugs seem to exist by comprehending the global situation, he must examine this area more minutely and hopes to reach the erroneous points. Since the information of the execution is enormous and complicated, it is necessary to zoom gradually in on the location of the bug with the breadth of a view (that is, a level of abstraction) kept properly.

To zoom in on the location of the bug efficiently, it is important to extract as much useful information from a user as possible, with the amount of data to treat limited to as little as possible. For this reason, a debugger is required to display many kinds of information which the user needs, in abstracted form(with small

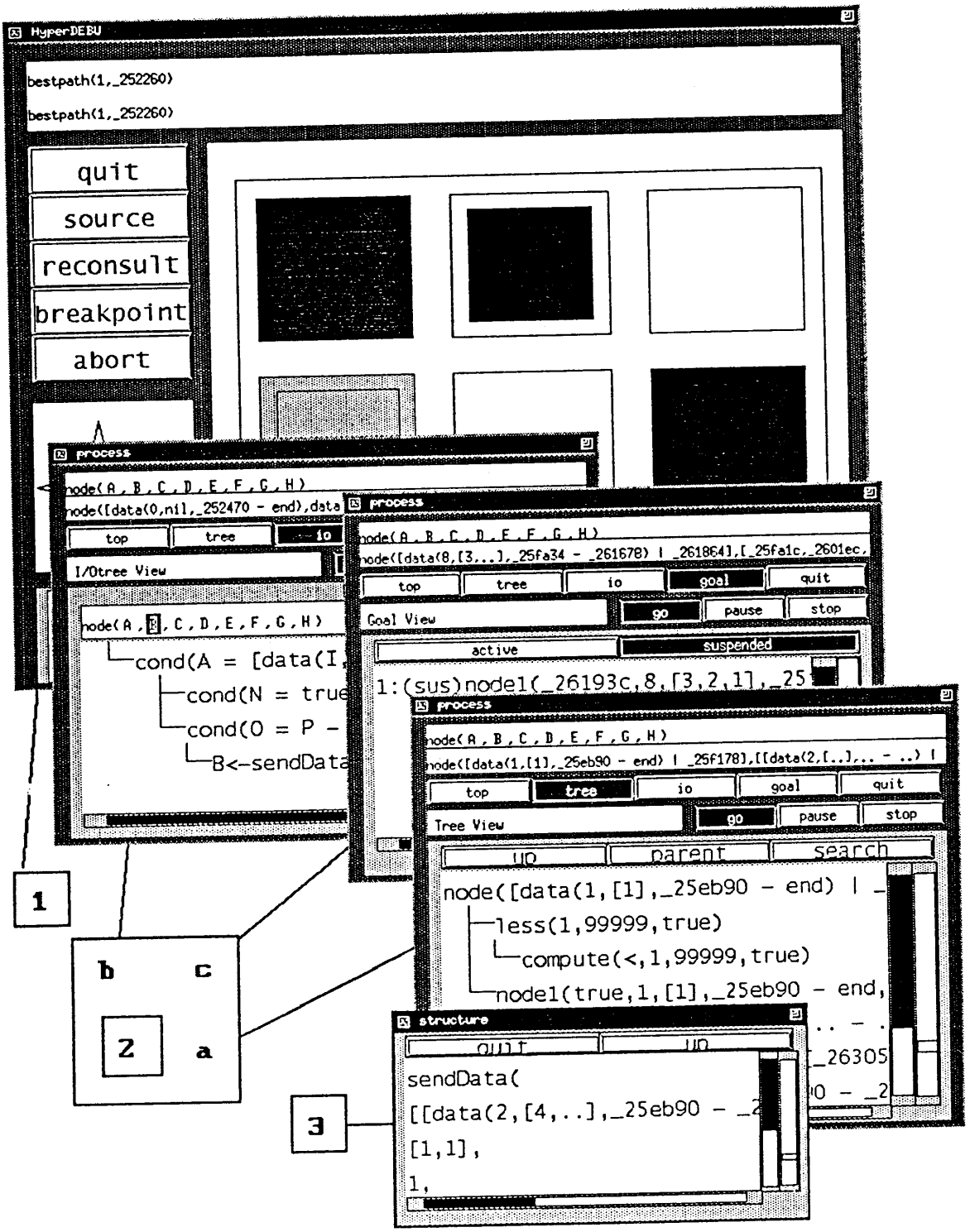

Fig. 2. Overview of HyperDEBU

amount of characters). Moreover, it is essential that the abstracted information on the screen is easy to understand and that the directions the user can give to the debugger are elastic. A high-dimensional interface plays an important role in solving this problem.

Grasping the Static Information of the Program Since a programmer has to grasp the static information of the program to debug it, he refers to the source code if necessary throughout debugging. To debug a concurrent program, he must trace multiple flows in the source code simultaneously. Although most of conventional multiwindow debuggers provide a window as a source level debugger for each process, this method is not applicable to a highly parallel program.

Control of the Execution To debug a program, a programmer examines and manipulates its execution. Therefore, a mechanism to control the execution is needed. For a sequential program, the programmer sets up breakpoints in desirable places to make the execution stop there and to examine the state of the program. However, a new control method is required to debug a highly parallel program which has many threads of execution.

The nondeterminism of a parallel program is also important. To debug such a program, a debugger has to control this nondeterministic behavior. Note that the synchronization mechanism of guarded unification of CCL is helpful in eliminating many nondeterministic bugs.

4.3 Features of HyperDEBU

In this section, we will describe the functions of our debugger HyperDEBU, which is designed to satisfy the requirements described in the above section.

Various Views for Bug Locating

Toplevel-window and Process-windows For the flexibility of levels of views, that is, in order to support global view as well as local view, HyperDEBU provides a toplevel-window and process-windows.

A toplevel-window, which provides a global view, is opened initially, and a user provides initial goals into it. The user can examine and manipulate execution of a program in the global scope with this window. Moreover, the user can get process-windows to examine the details of any processes displayed on the toplevel-window. The toplevel-window manages all of the process-windows.

A process-window, which can be opened for any process, enables examination and manipulation of the process. To locate bugs, a user can get subprocesses as other windows from this process.

Moreover, HyperDEBU has a structure-window which provides a data-level view. A user can get it from any data displayed on windows of HyperDEBU. Since all of the data on HyperDEBU are updated as the program runs, a user can examine state of the program dynamically.

Three Views of Process-window A process-window enables bugs to be located efficiently using the flexible levels and aspects of the process model. The process-window has three kinds of window called views as follows.

- TREE view : This shows an execution tree (control flow relationship of the computation).
- I/O tree view : This shows a tree of input-output causality (data flow relationship of the computation).
- GOAL view : This shows a set of goals (snapshot of the computation).

They enable flexible examination from multiple aspects. A user can get a subprocess as a window from their views.

Program Visualization The global view of the toplevel-window visualizes execution of a Fleng program. Execution of a CCL program is represented by visualizing the following two flows :

- control flow: execution status of goal reduction
- data flow: execution status of guard and unification

Their histories correspond to an execution tree on TREE view, and a tree of input-output causality, respectively. However, visualizing all goals and data is hard to comprehend. The toplevel-window visualizes only processes with respect to some particular goals and only some particular data-flows. A user needs to specify what is "*particular*" , depending on the user's intention. HyperDEBU provides "*breakpoints*" to specify it.

Note that one of the problems of visualization of execution status is the need for locating the display objects dynamically on the screen since the goals and data themselves are created dynamically.

Control-flow The toplevel-window visualizes processes with respect to some particular goals to provide a global view. Figure 3 shows an overview of the toplevel-window. It is regarded as the view from the top of Figure 1. Each process is displayed as a rectangle.

- A color of the rectangle indicates the state of the process (white, light gray and dark gray indicate active, suspend,and terminated respectively)
- A nest of rectangles indicates the relation between a process and its subprocess.
- A topgoal of a process is displayed when the mouse cursor enters the corresponding rectangle.
- Clicking a rectangle generates a new process-window for this process.

Contents of all the windows are updated reflecting the state of the execution dynamically. By observing creations and state transition of processes, and by observing modification of data in the arguments of topgoals, a user can comprehend the execution of Fleng program correctly and easily.

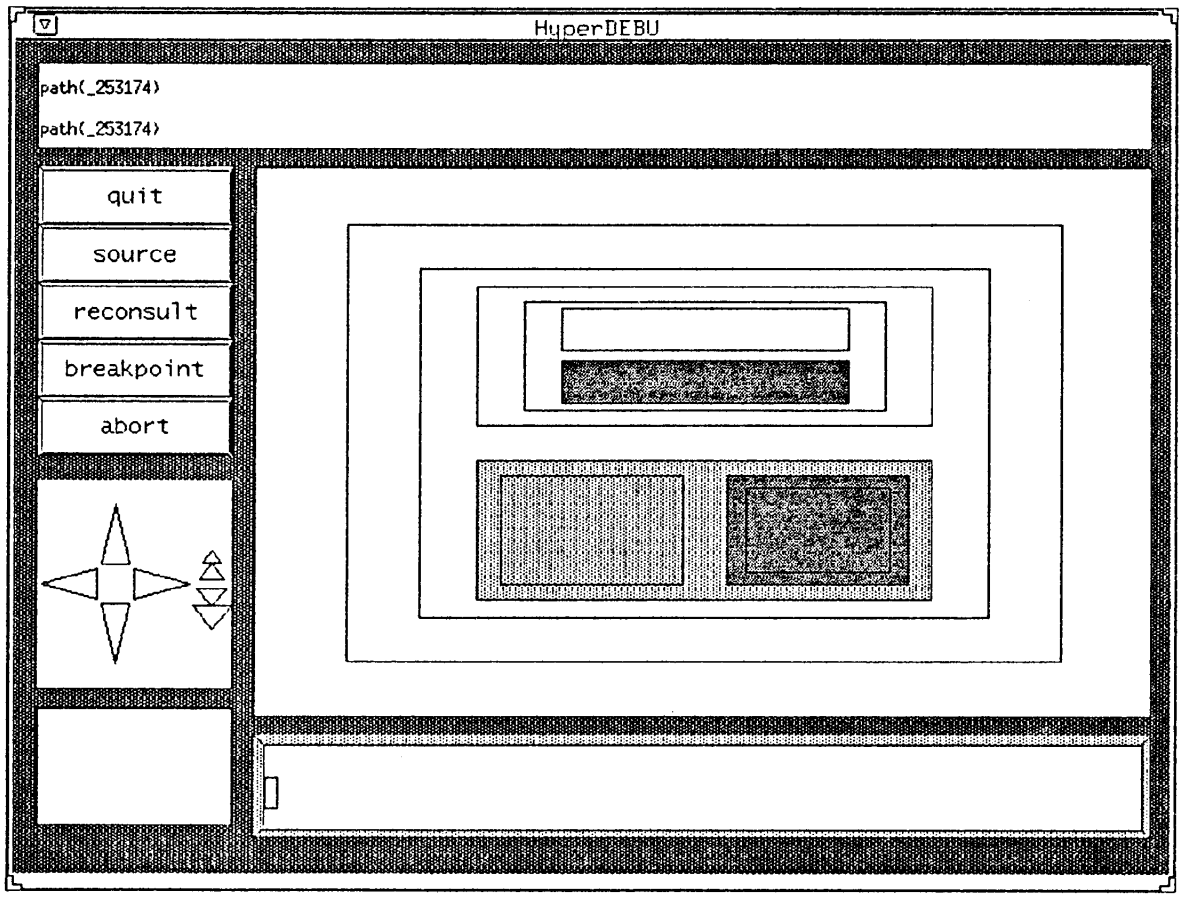

Fig. 3. A Toplevel-window

Data-flow Though data flow is represented as a tree of input-output causality in the I/O tree view, the display of the I/O tree view is too complicated to understand the global data flow of a large scaled program. To solve this problem, the toplevel-window should visualize some particular data flows. We provided a feature of visualizing some particular stream between processes.

Figure 4 shows the 4 forms of stream for visualizing the data flow. They are (1) the generation of stream, (2) the distribution of stream, (3) the data output to the stream, and (4) the data input from the stream(guard).

Breakpoints for Parallel Execution The conventional approach to debugging a sequential program is to stop the program at a breakpoint and to examine the state of execution. However, this approach is not applicable to a parallel program which has multiple control flow.

We extend "breakpoints" as a debugger's knowledge given by a user before the execution of the program. The debugger uses this information to control the execution, visualization and static debugging.

Breakpoints are specified as pairs of "point" and "direction". The following place in a program can be specified as a point :

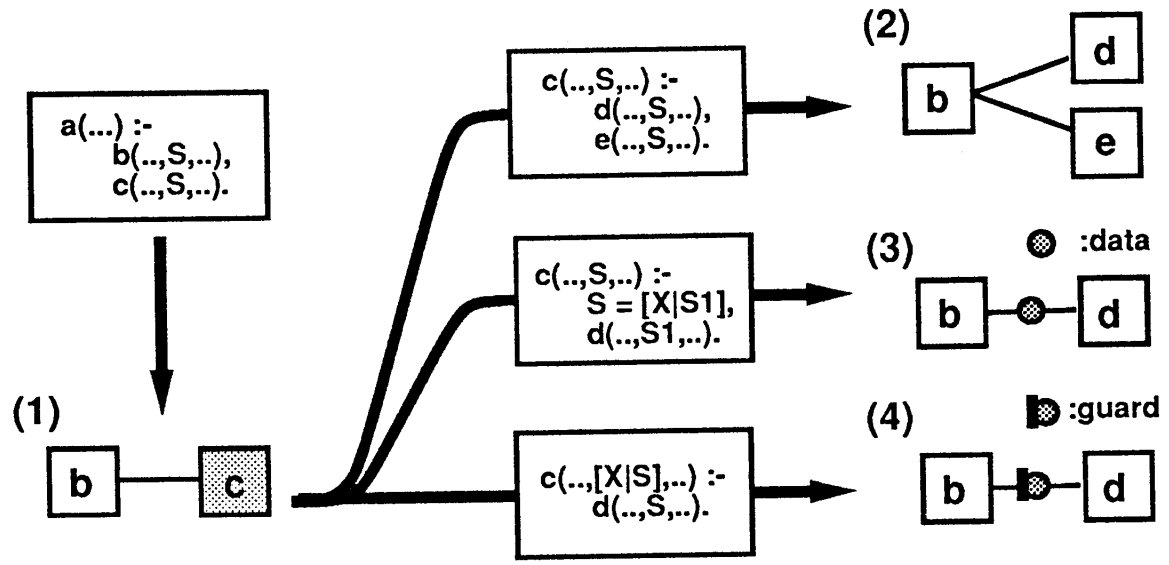

Fig. 4. The Generation, Distribution, IO of Stream

- predicate (a set of clauses which have the same name)
- clause
- body goal
- argument of goal (for dataflow)

There are directions as follows :

- pause : This directs to stop the goal there.
- process : This directs to visualize the process with respect to a goal.
- notree : This directs not to keep execution history.
- stream : This directs to visualize the data as a stream.

Browsing Program Code Since the need to set breakpoints before execution can be a burden to a user, a debugger is expected to aid the user to comprehend static information with which the user decides breakpoints. The program code browser of HyperDEBU is provided for this purpose.

This browser supports the following functions :

- tracing a graph of predicates (procedures) created from cross references caller/callee relations among predicates
- navigating from some execution point of a program to the corresponding point of source program
- searching the history and program code etc. from the name of predicate

These functions are applicable to the following aids to a user :

- setting breakpoints
- static debugging
- correcting a source code

5 Examples

In this section, we will demonstrate the effectiveness of HyperDEBU by showing two examples of debugging using HyperDEBU.

Example 1 The following example is a program to solve "good-path problem".

```
path(A) :- token(start,[],A,[]).

token(Node,History,H,T) :-
        (Node == goal ->
           H = [[goal|History]|T]
        ;
           next(Node,Next),
           checknext(Next,[Node|History],H,T)).

checknext([],History,H,T) :- H = T.
checknext([N|Ns],History,H,T):-
        member(N,History,Result),
        gonext(Result,N,History,H,T1),
        checknext(Ns,History,T1,T).

gonext(true,_,_,H,T) :- H = T.
gonext(false,Node,History,H,T):-
        token(Node,History,H,T).

next(start,Next) :- Next = [a,d].
next(a,Next):- Next = [start,b].
next(b,Next):- Next = [a,c,goal].
next(c,[b,d,goal]).                    %erroneous
%next(c,Next):- Next = [b,d,goal]. %correct
next(d,Next):- Next = [start,c,e].
next(e,Next):- Next = [d,goal].

member(_,[],R) :- R = false.
member(X,[Y|Z],R) :-
        (X == Y -> R = true ; member(X,Z,R)).
```

This program searches the paths on a directed graph and finds all paths from `start` to `goal`. The `next` clauses specify the directed graph; for example, the first clause tells the node `start` has two arrows directed to `a` and `d` respectively. To get the solution, an initial goal `path(X)` is given at first. Then it matches with the first clause of this program and a new goal `token` is generated and spawned. The `token` goals spawn themselves and search the paths from `start` for `goal`. A `token`, which has a node as the first argument `Node`, spawns a goal `checknext` if `Node` is not `goal`. The `checknext` spawns goals `gonext` for the nodes next to the node. Each `gonext` generates a `token` goal if the path from `start` to the node has no loop. If a `token` reaches the node `goal`, it links the solution with the variables `H` and `T` to make the list of the solutions. However, the erroneous

definition of **next** makes this program suspend illegally without returning the solutions.

At first, a breakpoint for visualization is placed in order to observe the creation of **token** goals which make the main control flows of this program. Figure 5 shows the visualization of control flow on the toplevel-window. The light-gray rectangles indicate the processes concerned with them are suspended.

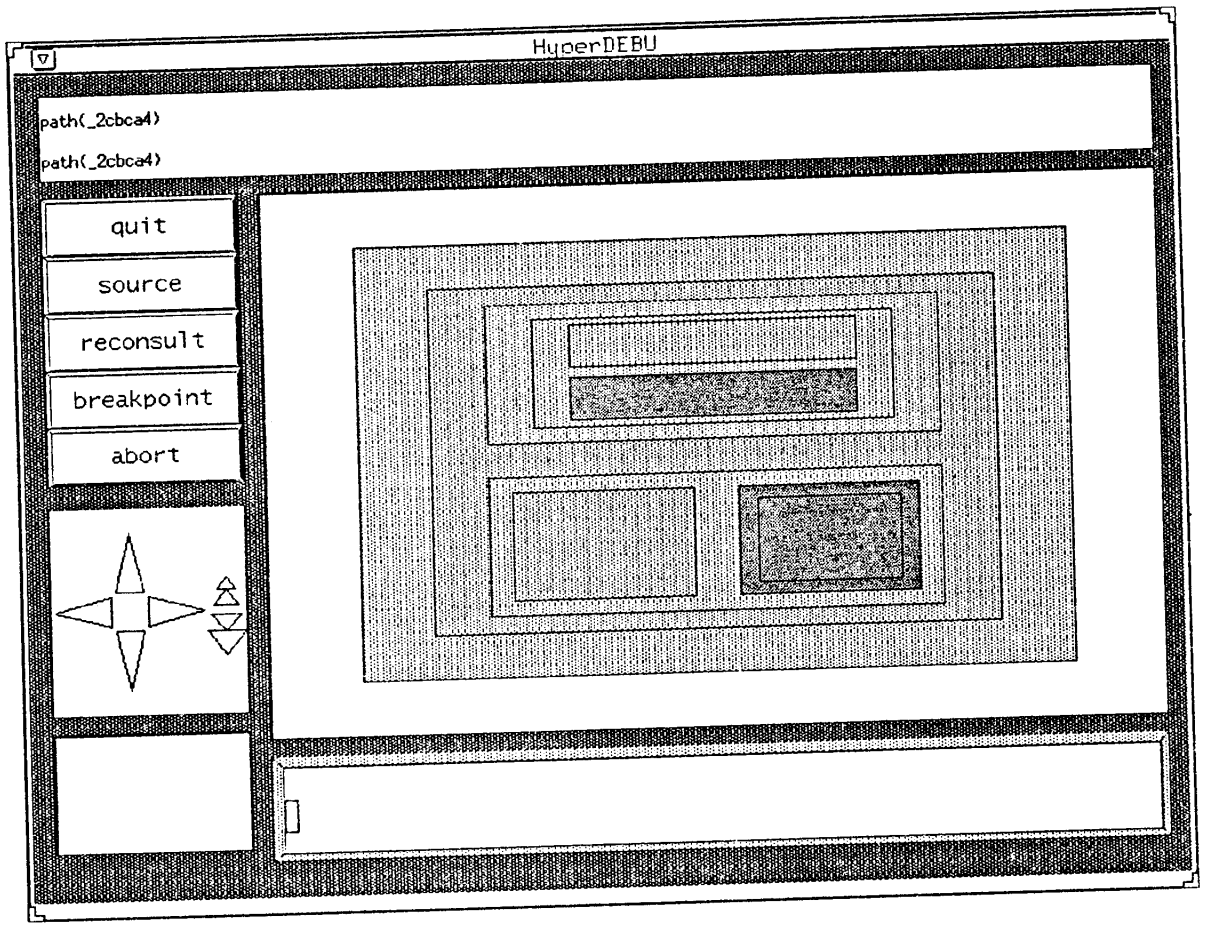

Fig. 5. Example 1-1: Visualization of Toplevel Window

Then, one of these processes, which fails to output results, is opened as a process-window (Figure 6). The third argument of its topgoal is still an undefined variable which must have been fixed. Figure 6 shows the display of an I/O tree view to examine the output of the third argument. It shows that **checknext** is suspended without outputting intended result. The structure-window which displays the instance of **checknext** shows the first argument of **checknext** is an undefined variable. The I/O tree view tells that the input of the first argument E must have come from **next(A,E)** which is suspended illegally. From this part on I/O tree view, we can open a window to display the clauses which defines **next** and get the definition clause which includes a bug as shown at lower side of figure 6.

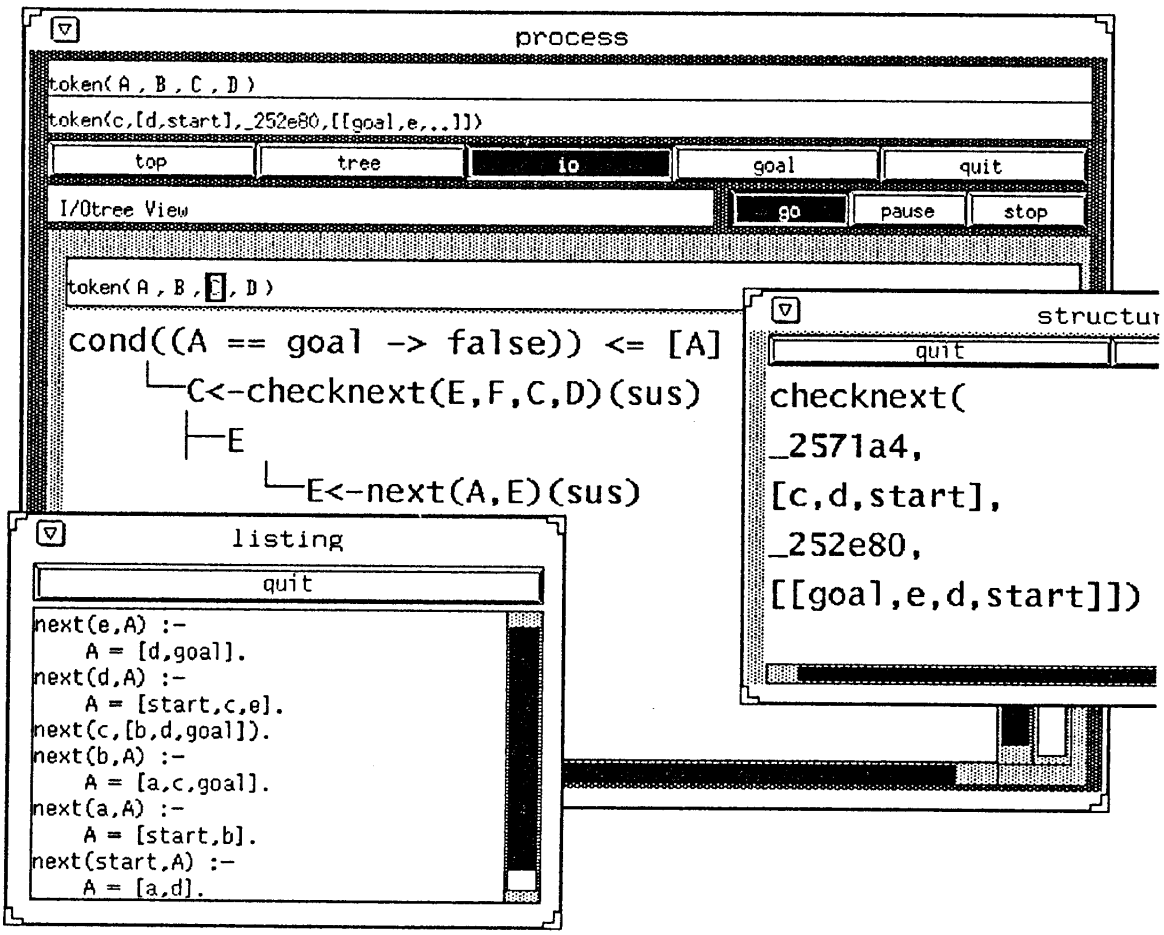

Fig. 6. Example 1-2: Process Window and Its I/O Tree View

Example 2 The next example program has a bug involving input-output causality. Although two programs ex1 and ex2 have the same input and output, their results are different.

```
ex1(A,Z) :-
        multi1([A|X],Y,Z),
        filter(Y,Yout),filter(Z,Zout),
        merge(Yout,Zout,X).
ex2(A,Z) :-
        multi2([A|X],Y,Z),
        filter(Y,Yout),filter(Z,Zout),
        merge(Yout,Zout,X).

multi1([A,B|X],Y,Z) :-
        Y = [A|Y1],Z = [B|Z1],multi1(X,Y1,Z1).
```

```
multi1([],Y,Z) :-
        Y = [], Z = [].

multi2([A|X],Y,Z) :-
        Y = [A|Y1],multi21(X,Y1,Z).
multi2([],Y,Z) :-
        Y = [], Z = [].
multi21([A|X],Y,Z) :-
        Z = [A|Z1],multi2(X,Y,Z1).
multi21([],Y,Z) :-
        Y = [], Z = [].

filter([A|In],Out) :-
        (A > 0 ->
            sub1(A,A1),
            Out = [A1|Out1],
            filter(In,Out1)
        ;
            Out = [A]).
filter([],Out) :- Out = [].

merge([A|X],Y,Z) :- Z = [A|Z1],merge(X,Y,Z1).
merge(X,[A|Y],Z) :- Z = [A|Z1],merge(X,Y,Z1).
merge([],Y,Z) :- Z = Y.
merge(X,[],Z) :- Z = X.
```

Although both of `multi1` and `multi2` split the input of the first argument into the second argument and the third argument, they are different with respect to input-output causality. While `ex2(5,X)` returns a result `X = [4,2,0,0]`, `ex1(5,X)` falls into deadlock and outputs nothing. Figure 7 and Figure 8 show the I/O tree views displaying the results of `ex1(5,X)` and `ex2(5,X)` respectively.

Regarding the output of `ex1`, `multi1` is suspended without output to `B`. It has got input `C = [A|F]` and waiting for next input from `merge` which is waiting for inputs of `G` and `E`. However, two `filter` which must output to `G` and `E` are both waiting for output of `multi1`. This loop of input-output causality brings this program into deadlock.

On the other hand, regarding the output of `ex2`, the output of `B` exists on condition that `C = [D|E]` and `E = [F|G]`. This condition is satisfied by the existence of the output of `merge` to `O` which requires the input `P`. A goal `filter` can output to `P` because the input `Q = [D|I]` exists on condition that `C eauals [D|E]`.

Example 3

Figure 9 shows an example of datafllow visualization. Large rectangles indicate processes, small squares the data generated to the streams, black circle with vertical line the gurds(data is received from the side with the vertical line), and a rectangle with characters a goal. Lines connecting these elements are streams.

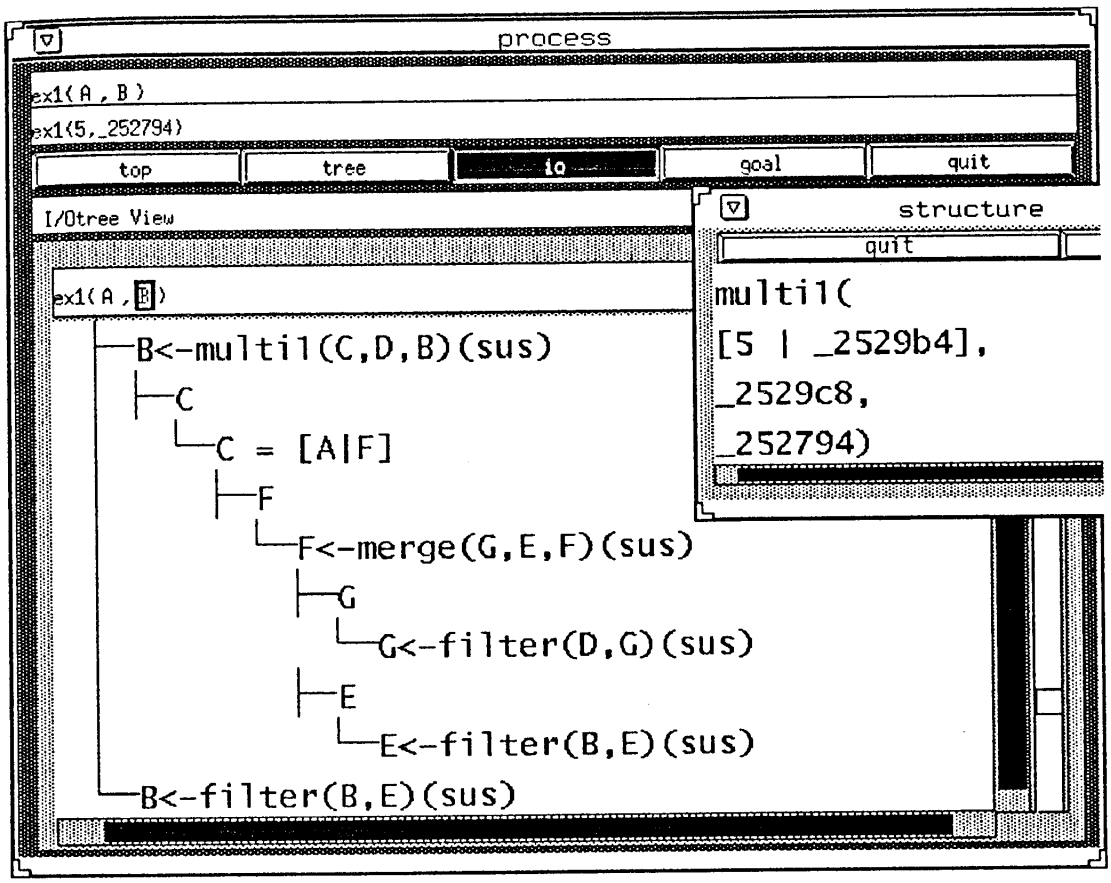

Fig. 7. Example 2-1: I/O Tree View for **ex1**

6 Discussions

6.1 Related Works

PIMOS[8], the operating system on the parallel inference machine PIM, has a tracer for language KL1 which is a practical version of GHC on PIM. It traces a particular thread of execution. However, when we have to deal with many threads of a highly parallel program, it is very difficult to understand them. Moreover, some facilities to trace data flows are required. Although PIMOS can also detect such goals that fall into deadlock, the relation and histories of these goals are not presented to users. On the other hand, HyperDEBU enables to solve this problem since it provides flexible views ranging from global to local view and enables users to examine expected control/data flows and to locate bugs efficiently.

There are some debuggers which visualize the execution of a CCL program as communicating processes. For example, [9] is one of such debuggers. However, they regard one sequence of goals as a process. For this reason, they can hardly visualize execution of a highly parallel program concisely. The flexibility of abstraction, which HyperDEBU has, is required to debug such a highly parallel program.

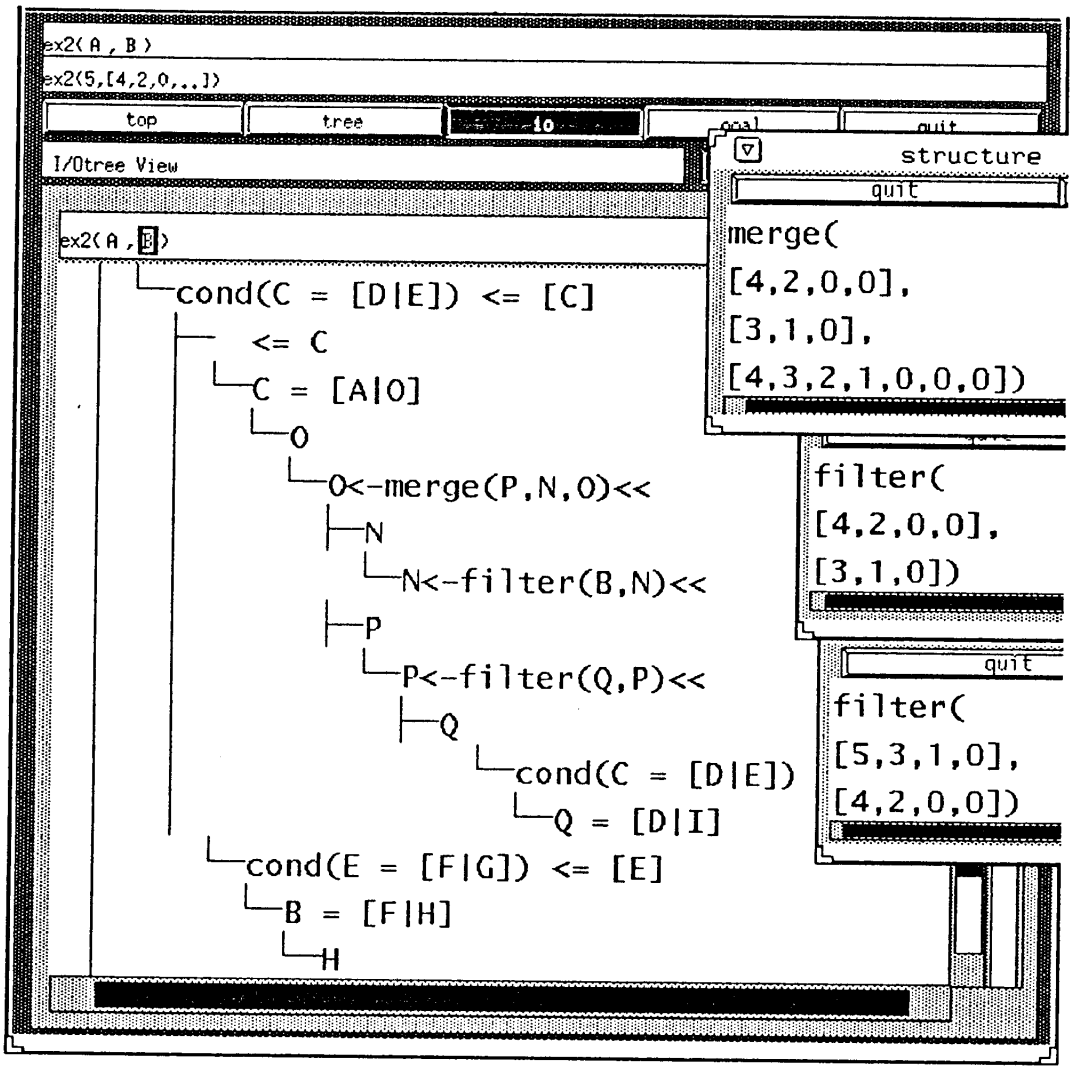

Fig. 8. example 2-2: the I/O tree view for ex2

6.2 Nondeterminism

Nondeterminism of a CCL program is caused by the following factors.

1. racing active unifications
2. nondeterministic commitment of clauses

In case 1, a programmer can find a failure of one of these unifications which does not happen in a correct program. Then, using an I/O tree view, the programmer can find the race of outputs to one variable in a tree of active unifications and guarded unifications.

In case 2, introducing some extension such as an OR-tree of input-output causality, an I/O tree view will be able to represent nondeterministic commitment. To analyze how this nondeterminism effects the results of the program execution, some tools need to be developed. However an OR-tree of input-output

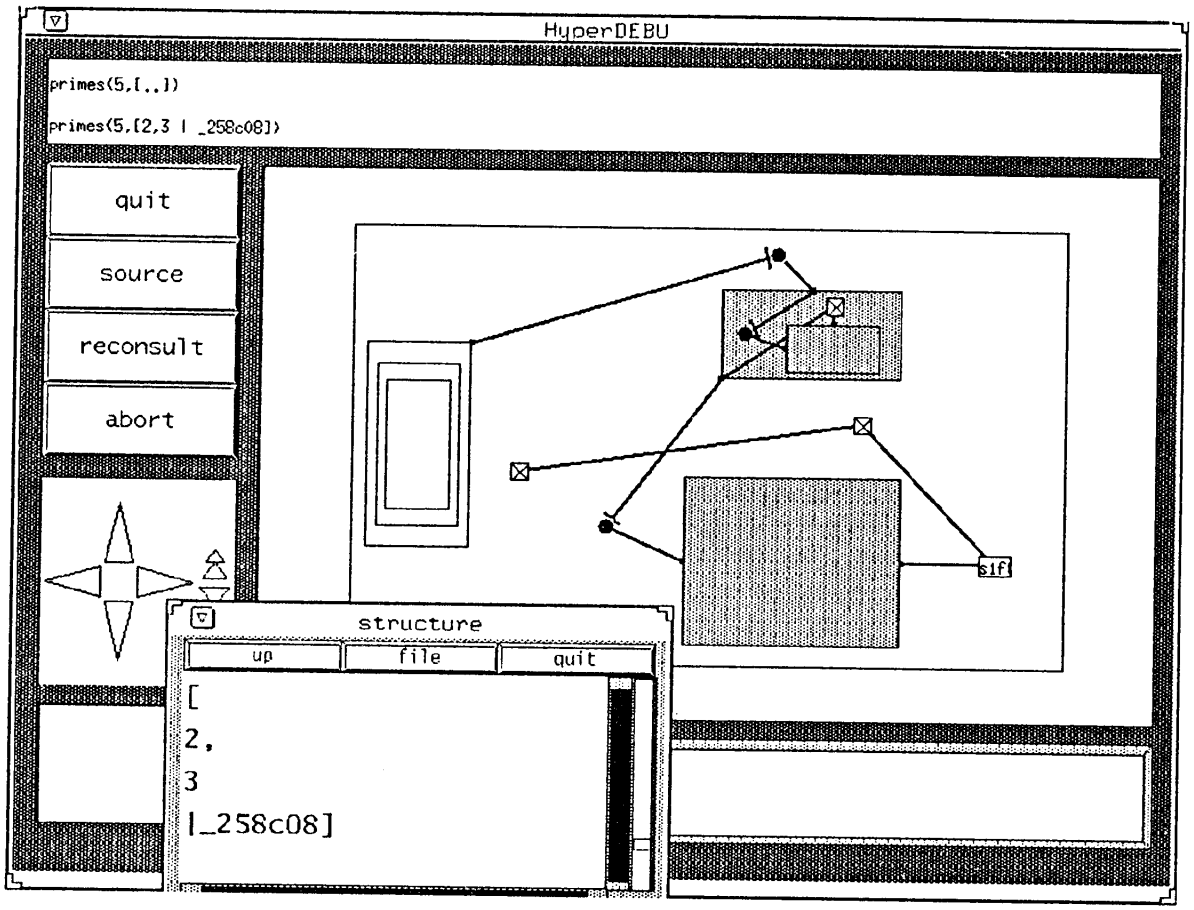

Fig. 9. Example of Visualization of Dataflow

causality is too complex to compute in practical time. Unlike verification, we think that a debugger needs only to report the existence of nondeterminism without computing all possible execution of the program. We are planning to introduce some breakpoints for nondeterministic commitment to enable users to control nondeterminism.

6.3 Costs of Time and Space

When a program is executed with a debugger, there is some overhead such as time to execute codes for debugging and memories for event histories. Although the performance of HyperDEBU is enough for practical use now, more improvement is desired.

HyperDEBU takes much time since it uses meta-interpreter to manage execution of goals. We will design an improved version of the goal management mechanism which is realized by adding some primitives to the interpreter or by modifying the compiled code.

Although event histories are useful to debug parallel programs, it is almost impossible to record whole events. Therefore, HyperDEBU provides a breakpoint to designate such goals (execution tree) that should be recorded. The

visualization of program execution through toplevel window helps to identify the erroneous process and results in decreasing the volume of history to store.

7 Future Work

We have realized a realistic debugger for parallel logic programs. For more improvement, we have some subjects as follows:

- support of compiler, operating system, and architecture to debugger
- control of nondeterministic behavior
- static analysis to help debugging
- performance debugging
- total environment for parallel programming

8 Conclusion

In this paper, we introduced a communicating process model to represent execution of a Fleng program, and presented a multiwindow debugger HyperDEBU which represents this model on a high-dimensional interface.

HyperDEBU is written in Fleng and runs on a Fleng interpreter on UNIX workstations and Mach parallel workstations. It is used in the development of application programs including HyperDEBU itself.

References

1. Mcdowell,C.E. and Helmbold,D.P.: *Debugging Concurrent Programs*, ACM Computing Surveys, Vol.21 No.4, pp.593-622 (1989).
2. Nilsson, M. and Tanaka, H.: *Massively Parallel Implementation of Flat GHC on the Connection Machine, Proc. of the Int. Conf. on Fifth Generation Computer Systems*, p1031-1040 (1988).
3. Koike, H. and Tanaka, H. : *Parallel Inference Engine PIE64*, in *Parallel Computer Architecture*, bit, Vol.21,No.4,1989, pp.488-497 (in Japanese).
4. (Ed.) Shapiro, E. : *Concurrent Prolog : Collected Papers* , (Vols. 1 and 2), The MIT Press (1987).
5. Murakami, M. : *An Axiomatic Verification Method for Synchronization of Guarded Horn Clauses Programs*, ICOT Technical Report TR-339, ICOT (1988).
6. Murakami, M. : *A Declarative Semantics of Parallel Logic Programs with Perpetual Processes*, ICOT Technical Report TR-406, ICOT (1988).
7. Tatemura, J. and Tanaka, H. : *Debugger for a Parallel Logic Programming Language Fleng*, Proc. of Logic Programming Conference '89 (1989).
8. Chikayama,T. and Suzaki, K. : *PIMOS 1.5 Introductory Manual*, ICOT Technical Memorandum TM-884, ICOT (1989).
9. Maeda, M., Uoi, H. and Tokura, N. : *Process and Stream Oriented Debugger for GHC Programs*, Proc. of Logic Programming Conference '90 (1990).