Control and Data Flow Visualization for Parallel Logic Programs on a Multi-window Debugger HyperDEBU

Junichi TATEMURA, Hanpei KOIKE, Hidehiko TANAKA
{tatemura,koike,tanaka}@mtl.t.u-tokyo.ac.jp
Department of Electrical Engineering, The University of Tokyo
7-3-1 Hongo, Bunkyo-ku, Tokyo, 113 JAPAN

Abstract. A fine-grained highly parallel program has many threads of execution. The first task to debug it is comprehending the situation of the execution. For this task, it is important to visualize the execution. Our debugger HyperDEBU for a parallel logic programming language Fleng visualizes control / data flows of execution of a Fleng program according to a user's intention. Breakpoints are introduced as information which represents a user's intention or points of view. HyperDEBU uses this information to visualize execution of a program. HyperDEBU enables efficient debugging by its visual examining / manipulating facilities.

1 Introduction

When a program shows unexpected behavior, the first task for a 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. However, 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.

There are two types of program visualization researched; a visual debugger for sequential programs, and an algorithm animation to enable a user to understand an algorithm. They have different levels of abstraction. A visual debugger displays an execution at the abstract level with primitives of programming languages, and enables users to examine the precise behavior of the program. An algorithm animation system [1] displays an execution at the abstract level with an individual algorithm. Some codes are inserted as "probes" into each part of a program to call animation program dedicated to the program (or an algorithm).

However, for debugging highly parallel programs, they can not apply directly. If a debugger uses only the low level abstraction of the visual debugger, it is hard to comprehend the global situation of the program. On the other hand, the high level abstraction of the algorithm animation is not applicable directly because it requires complete specification for visualization, which may be a heavy load on a programmer and a cause of additional mistakes. Moreover, to find a bug, it must visualize unexpected behavior and provide the lower view for bug locating.

To realize a debugger for highly parallel programs, we need a visualizing technique using a proper abstraction for what we expect at that time. The debugger needs to visualize execution of a program according to a user's intention which varies during the debugging.

In this paper, we propose a method of visual debugging in which a debugger makes efficient use of a user's knowledge given in addition to source codes. This knowledge represents the user's intention or points of view. The debugger uses this information to visualize execution of a program. It does not require complete information about the program; it provides low abstraction level debugging when it has no information from a user, and, as it is given information, enables the user to debug the program at higher abstraction level.

We developed a multiwindow debugger HyperDEBU for a Committed-Choice Language Fleng which is one of the parallel logic programming languages. HyperDEBU visualizes execution of a Fleng program

and enables users to comprehend the global situation of the execution. An execution history of a Fleng program comprises of a large amount of control flow and data flow information. This debugger visualizes these flows according to a user's intention on global and local views. In this paper, we describe the multiwindow debugger HyperDEBU and the methods and facilities of its visualization of control / data flows.

2 Committed-Choice Language Fleng

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

A Fleng program is a set of horn clauses like:

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

The side to the left of :- is called the *head*, and the right side is called the *body* 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 into the body goals. The new goals are added to the set of goals. The execution begins when an 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 a head part of a clause and variables in a goal are prevented from being substituted. Such unification is *suspended* until these variables have values. Active unification is applied in a 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 value, and eliminates many nondeterministic bugs due to synchronization.

Therefore, a clause in a Fleng program defines control flows with *goal reduction* and data flows with *active / guarded unification*.

3 HyperDEBU : a Multi-window Debugger for Fleng Programs

We developed a multiwindow debugger HyperDEBU which provides a multi-dimensional interface.

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 multi-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 multi-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 conventional multiwindow debuggers use a window as a sequential debugger assigned to one of the processes. However, multi-dimensional information cannot be handled well in this way. HyperDEBU provides windows flexible enough for programmers to examine and manipulate complicated structures composed of multiple control/data flows. Tracing links on a window which displays information of a program execution, a user can get an expected window.

The conventional notion of process for CCL is associated with one goal or one sequence of goals. The process model for our debugger is equivalent not to one goal but to all of its subgoals generated by reduction. Let G be a goal and P be a set of goals which are derived from G. We call P "*the process with respect to G*", and call G "*the topgoal of P*". Since a subgoal can be a topgoal, a process consists of some subprocesses. This hierarchy of processes makes the debugger applicable to highly parallel programs.

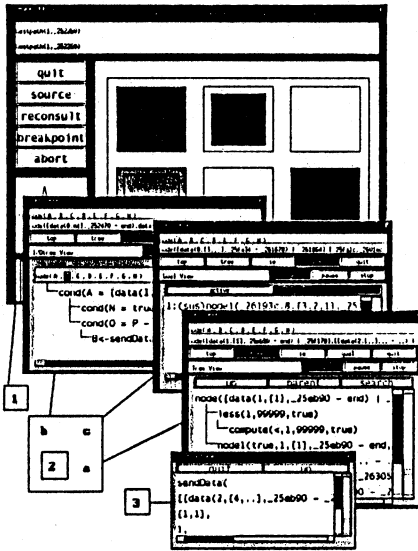

Fig. 1. Overview of HyperDEBU

HyperDEBU consists of the following windows : (1) toplevel-window, (2) process-windows ((a) TREE view (b) I/O tree view (c) GOAL view), and (3) structure-windows. Figure 1 shows an overview of HyperDEBU.

HyperDEBU has these features which cooperate with each other to aid a user to find a bug.

1. **Various views for bug locating.** Since it is necessary to zoom gradually in on the location of the bug as the breadth of a view is kept properly, a debugger's view is required to have the flexibility of views from global to local. A toplevel-window provides a global view. A user can get a process-window to examine detail of any process displayed on the toplevel-window. A process-window enables examination and manipulation of the process. To locate bugs, a user can get a subprocess as another window from this process. The process-window has three views of the process. Moreover, HyperDEBU has a structure-window which provides a data-level view.
2. **Visualization.** The global view of the toplevel-window visualizes execution of a Fleng program. This function helps a user to comprehend the global situation of the execution, and makes bug locating more efficient.
3. **Breakpoints for parallel execution.** Since a parallel program has multiple threads of control flow, a new mechanism to control the execution is needed in order to debug a highly parallel program. 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.
4. **Browsing program code.** To comprehend static information of Fleng programs, HyperDEBU has a prototype of the program code browser. This function helps a user to set breakpoints, make static debugging, and correct a source code.

4 Program Visualization on HyperDEBU

4.1 Basic Approach

To visualize execution of a highly parallel program, a debugger must deal with a large amount of control flow and data flow information. A Fleng program can be represented by visualizing *goal reduction* as control flow and *guarded / active unification* as data flow. However, visualizing all goals and data is hard to comprehend.

HyperDEBU visualizes these flows using proper abstraction according to a user's intention which is given as additional information. The debugger provides low level abstraction for debugging when it has

no information from a user, and, as it is given information, enables the user to debug the program at higher level abstraction.

We introduce "breakpoints" as information which represents a user's intention or points of view.

4.2 Breakpoints for Parallel Execution

Breakpoints are specified as pairs of "point" and "direction". They are regarded as additional knowledge about a program to be debugged. The following places in a program can be specified as a point : (1) predicate (a set of clauses which have the same name), (2) clause, (3) body goal, and (4) argument of goal.

There are directions as follows :

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

The "process" breakpoint and the "stream" breakpoint are regarded as a user's information to visualize control flows and data flows, respectively. These details are described later.

4.3 Local and Global Views

HyperDEBU provides *local* views with low level abstraction to locate bugs, and *global* views with high level abstraction to comprehend the global situation. Each type of views represents control and data flows of execution of a program.

Local Views with Low Level Abstraction : A user can examine and manipulate execution of a program through process windows of HyperDEBU. The process window has multiple views to show control and data flows; TREE view and I/O tree view. These views use the abstraction at the level with goals and data within a process. The TREE view visualizes a tree of goal reduction to represent control flows. The I/O tree view visualizes a tree of guarded /active unifications to represent data flows. These *local* views are helpful to zoom in on the location of the bug, and to examine detailed behavior of the erroneous part of the program.

Global Views with High Level Abstraction : The toplevel window of HyperDEBU provides a global view with high level abstraction. After a user comprehends situation of execution using this global view, he can get a process window from the toplevel window as a local view to locate bugs. To realize this visualization facility, we introduced (1) a set of visualized objects to represent processes and streams, (2) a method to decide visualized objects from program code and user's intention, and (3) a method to arrange visualized objects on a display. Note that one of problems on the visualization of Fleng program execution is the need for dynamic location of display objects since goals and data themselves are created dynamically.

In the following two sections, we describe details of our visualization technique for control and data flows.

5 Visualizing Control Flows

5.1 Visualized Objects

HyperDEBU visualizes creations and state transition of processes. Since a process is defined as a set of goals derived from a goal, each goal in the execution history has corresponding process. HyperDEBU visualizes only processes with respect to some particular goals.

Figure 2 shows an overview of the toplevel window. 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.

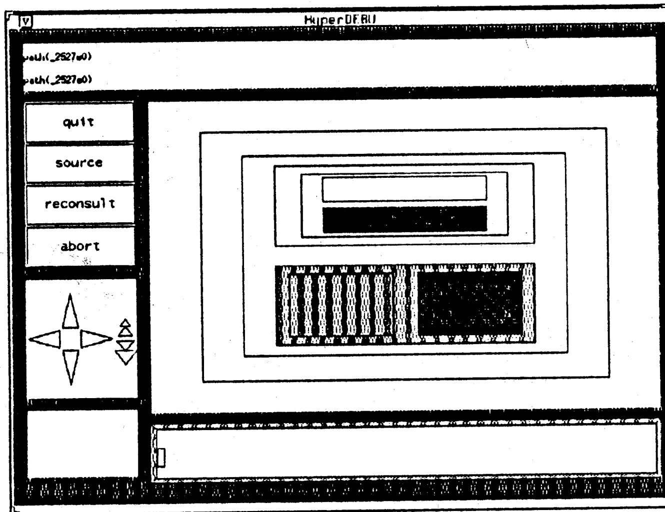

Fig. 2. Visualizing Control Flows

5.2 Breakpoints

HyperDEBU visualizes a process as a rectangle only if "process" breakpoint is specified at its top goal. Other processes are abstracted as internal goals in their parent processes, and not visualized as rectangles.

Since a rectangle may have a large number of goals as its elements, this visualization facility is applicable to a realistic highly parallel application composed of modules in which many goals are executed in parallel.

Even when the same execution of a program is visualized, a user can get various display from different view points according to he selects where breakpoints are set.

5.3 How to Display

As processes are created, HyperDEBU arranges rectangles dynamically using the following policy.

Let P be a process to be visualized. Then we introduce the following terms:

- $Sub(P,n)$: n-th sub-process of a process P.
- $D(P)$: depth in the hierarchy of rectangles ($D(Sub(P,n)) = D(P)+1$, $D(P) = 0,1,2,\cdots$).

The rectangle of P is divided into $X \times Y$ areas (X lines and Y columns) and each area is called $a(i,j)$ ($i = 1,\cdots,X$, $j = 1,\cdots,Y$). $Sub(P,n)$ is placed at $a(i,j)$. (i,j,X,Y) is decided as follows : if $D(Sub(P,n)) = 2k$ ($k = 0,1,2,\cdots$),

$$(i,j,X,Y) = \begin{cases} (m, n-m(m-1), m, m) & \text{if } m^2 \le n < m(m+1) \\ (n-m^2, m+1, m, m+1) & \text{if } m(m+1) \le n < (m+1)^2 \end{cases}$$

if $D(Sub(P,n)) = 2k+1$ ($k = 0,1,2,\cdots$),

$$(i,j,X,Y) = \begin{cases} (n-m(m-1), m, m, m) & \text{if } m^2 \le n < m(m+1) \\ (m+1, n-m^2, m+1, m) & \text{if } m(m+1) \le n < (m+1)^2 \end{cases}$$

where $m = 1,2,3,\cdots$.

As n increases in the execution, rectangles are arranged as in Figure 3 ($D(Sub(P,n) = 2k$). This method to arrange displayed objects has the following features:

- efficient use of area in a rectangle
- applicable to animation since it is easy to understand the relationship between the adjacent states of placements
- simple incremental arrangement algorithm (easy to implement for dynamic visualization)

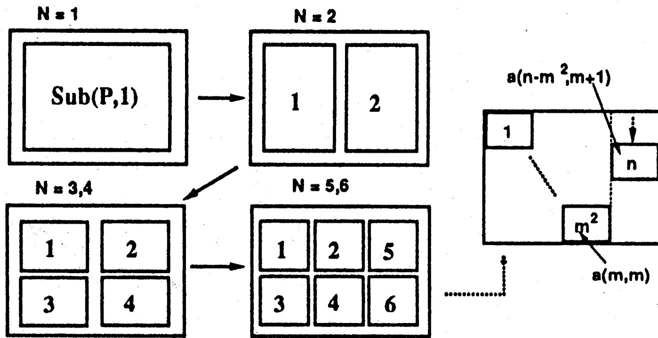

Fig. 3. Arranging Visualized Objects of Control Flow

6 Visualizing Data Flows

6.1 Visualized Objects

HyperDEBU visualizes particular *streams* which make main global data flows of the computation. The stream-based communication is a essential programming technique in CCL which enables a continual communication between processes. It is realized using shared variables as follows.

1. One of goals which have a shared variable writes some data structure into this variable (stream output).
2. When the value of the variable is bound, the other goals read it (stream input).
3. The data structure includes new variables which are shared by goals and used to continue the communication.

To represent the stream-based communication, a debugger must visualize its "*stream variables*" and its "*stream actions*". A "*stream variable*" is a variable which is shared by some goals and used for communication as described above. "*Stream actions*" are operations of these goals to the stream variable, and include creation of the stream, distribution of the stream, and input/output of the stream. These actions are visualized as Figure 4.

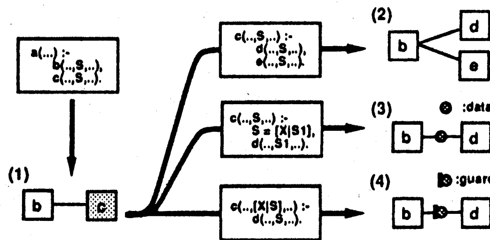

Fig. 4. Creation, Distribution, Input and Output of Stream

Figure 4-(1) is the object visualized when the body goals b and c, which share a new stream variable S, are created. This figure represents the creation of a new stream. Figure 4-(2) is the object visualized when the goal c is reduced into d and e and the number of goals which have a reference to the stream variable S is increased. This figure represents the distribution of the stream. Figure 4-(3) is the objects visualized when a data structure [X|S1] is substituted for the variable S. S1 is a new stream variable for the next communication. This figure represents the stream output. Figure 4-(4) represents the objects visualized when c is reduced into d after waiting for a data structure [X|S] which is displayed as a "guard". This figure represents the stream input.

To visualize a stream, the places corresponding to the stream actions as represented in Figure 4 must be specified in the program codes. Since many clauses have the stream actions to the same stream, these places are too many for a user to specify. A debugger is required to show essential information using as less specification from a user as possible. To realize this requirement, we should make clear (1) what a user wants to see, (2) how the user tells it to a debugger, and (3) what the debugger selects to visualize.

User's Point of View : A stream can be examined from various points of view. As a point of view, a user selects one of processes, and tries to examine: (1) its companions for the stream-based communication, and (2) its stream actions.

User's Specification : A user's intention is represented by selecting the stream to be visualized and the process as a point of view, that is, by specifying the process and its argument which has a stream variable. The user can specify the argument of the predicate as a "stream" breakpoint.

Objects to be Visualized: The data structure substituted for a stream variable includes new stream variables for the next communication. While a user sets a single breakpoint on a stream variable for a stream, a debugger must visualize a series of stream actions to the variables concerned with the stream. Accordingly, the debugger must analyze static data flow of a program to select objects to be visualized.

6.2 Breakpoints

Specifying "stream" Breakpoint : When a user sets a breakpoint on a stream variable, a debugger must find new stream variables for the next communication within the data structure substituted for the variable. Since the place of a stream variable within a data structure depends on a user's intention (i.e. what is a stream to be visualized), the user must specify not only a place of a stream variable in an argument of a process, but also where are next stream variables in the data of the stream variable. For this reason, we introduce "***types***" of streams which are sets of data substituted for a stream variable :

$$S = s_1(S) + s_2(S) + \cdots + s_m(S) + t_1 + \cdots + t_n$$

where $s_i(S)$ is a term (i.e. pattern of data) including stream variable S and any other variables; the term represents a set of data which are unifiable to it by substituting any data for their variables. Only the same type of data can be substituted for S, which is a stream variable for the next communication. t_i is a term which has no stream variables. This is used for termination of a stream. The operator "+" represents the union of sets.

In most cases, a sequence of list is used for a stream in a Fleng program. This type of stream is represented as:

$$S \;=\; \texttt{[X|}S\texttt{]} + \texttt{[]}$$

This is the default stream type in the current implementation of HyperDEBU. Although a visualization mechanism supports general types of the stream, a user can specify only the list as a type through the current user interface. We need a new interface which enables a user to specify general types easily.

Selecting Visualized Objects : The debugger analyzes static data flows of the program with the information from the user, and selects the objects to be visualized. The debugger takes the following steps :

1. find all the stream variables concerned with the specified stream :
   (a) If a stream breakpoint is specified to a stream variable s, s is *concerned with* the stream.
   (b) When s_1 is a stream variable *concerned with* the stream, and T is a data structure unified with s_1, if T includes a next stream variable s_2 specified with the stream type of s_1, s_2 is *concerned with* the stream.
   (c) When s_1 is a stream variable *concerned with* the stream, and T is data structure unified with a variable s_2, if T includes s_1 as a next stream variable, s_2 is *concerned with* the stream.
2. judge stream actions to be (or not to be) visualized :
   (a) If a stream breakpoint is specified to a stream variable s, a stream action to s is visualized.
   (b) If a body goal is a predicate which has a stream action to be visualized, a stream action of its head is to be visualized.

The step 2 provides proper information for the user's point of view. The debugger visualizes a series of actions *from* the creation of the stream variable shared with companions of the specified process *to* the action of the process, and abstracts the other stream actions. For example, stream actions of a goal as a subroutine for sending messages, and ones of companions for communication with the specified process are not visualized if these goals have no stream breakpoint. These goals are still visualized as goals even if they are reduced into subgoals.

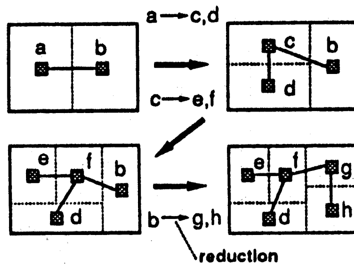

Fig. 5. Arranging Visualized Objects of Data Flow

6.3 How to Display

The debugger must visualize graphs composed of links of stream and nodes such as goals, data, and guards. To visualize such graphs for debugging, a method to arrange nodes on a display needs to satisfy the following requirements :

- To avoid additional tasks, a user does not have to specify the arrangement.
- Since graphs grow dynamically, the method should be incremental and suitable for animation.

Our method is as follows:

- **grouping nodes**: The nodes linked with streams each other are gathered as a group, and placed on the rectangle area in the same way to arrange processes.
- **arrangement in a group**: A rectangle area is divided into rectangles as many as nodes in the group. Each node is placed at the center of a rectangle. When a node is divided into new nodes, its rectangle is also divided and supplied for these nodes. The dividing line is vertical to the line which has created the rectangle for the node. Adjusting borders between rectangles, all the rectangle are kept the same size with each other. Figure 5 shows how to display a graph growing as nodes are generated.**a**,$\cdots$,**h** are nodes to be visualized such as goals, data, and guards. For example, the upper two graph show that **a** is divided into **c** and **d** by a stream action.

A user can specify both "process" and "stream" breakpoints to visualize control and data flows. A rectangle of process is also regarded as a node of the data flow graph. Data flow within a process is displayed in its rectangle.

7 Implementation and User Interface

HyperDEBU is written in Fleng itself, and runs on a Fleng system in parallel. It provides dynamic and interactive visualization and manipulation; since a user program and HyperDEBU run in parallel, a user can examine the state of the program and manipulate the execution at a run time.

In this section, we describe the visualization facility installed on HyperDEBU, by showing examples of displays.

7.1 Example of Data Flow Visualization

For an example of the data flow visualization on HyperDEBU, a program "primes" in Figure 6 is visualized.

This program generates a list of prime numbers. `gen/3` generates a list of integer from 2 to `Max`. `sift/2` gets this data and generates a filter which removes multiples of the integer `P`. We take up a stream which passes through these filters and generates prime numbers as the global data flow, and visualize it from two points of view.

According as points of view, we specify two different combination of breakpoints:

(1) Stream breakpoints are specified at the second and third arguments of `filter/3`.
(2) A stream breakpoint is specified at the first argument of `sift/2`.

The stream type of the breakpoint is a default type "*list*". The results are displayed in Figure 7-(1) and (2), respectively.

```
primes(Max, Ps) :- gen(2,Max,Ns), sift(Ns,Ps).
gen(N,Max,Ns) :- less_eq(N,Max,LE), gen1(LE,N,Max,Ns).
gen1(true,N,Max,Ns) :- Ns = [N|Ns1], add1(N,N1),
                       gen(N1,Max,Ns1).
gen1(false,N,Max,Ns) :- Ns = [].
sift([P|Xs],Zs) :- Zs = [P|Zs1], filter(P,Xs,Ys),
                   sift(Ys, Zs1).
sift([], Zs) :- Zs = [].
filter(P,[X|Xs],Ys) :- mod(X,P,Mod), eq(Mod,0,Eq),
                       filter1(Eq,P,X,Xs,Ys).
filter(P, [], Ys) :- Ys =[].
filter1(true,P,X,Xs,Ys) :- filter(P,Xs,Ys).
filter1(false,P,X,Xs,Ys) :- Ys = [X|Ys1], filter(P,Xs,Ys1).
```

Fig. 6. Example Program to be Visualized

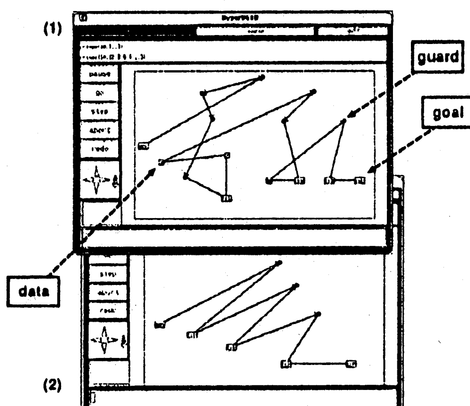

Fig. 7. Example of Data Flow Visualization

User Interfaces : The execution of the program "primes" is visualized as a graph on the toplevel window of each HyperDEBU in Figure 7. The elements of the data flow graph are as follows:

- data : a small square is output data of the stream.
- guard : a node composed of a circle and a line segment is a "guard", which gets input data from the side with the line segment.
- goal : a rectangle named by characters is a goal.
- stream : a line connecting nodes is correspond with a stream variable and shows the flow of data.

This data flow graph grows and changes its shape as the program is executed, and shows the situation of the execution dynamically by animation.

A user can manipulate objects interactively. When the mouse cursor points each node, the contents of the node is displayed at a sub window of the toplevel window. Clicking a node generates a structure window for this node. A user can examine the data more in detail with this window.

User's Point of View : The two toplevel windows (1) and (2) in Figure 7 provide different views. The former shows stream actions and companions of **filter**, and the latter shows ones of **sift**. In Figure 7-(1), a stream starts with the goal **gen** at the left side, goes through four guard objects, and gets at the goal **filter**. This indicates **filter** gets data through these guards from **gen**. Then the stream continues going to the next **filter** through two data objects and two guard objects. This indicates the **filter** outputs these data and the next **filter** gets them. In Figure 7-(2), stream actions of **filters** are not visualized. HyperDEBU visualizes only the creation of **filter** to represent stream actions of the goal **sift**. Although the stream actions of **gen** are not visualized in both views, selecting a breakpoint at **gen** makes them visualized.

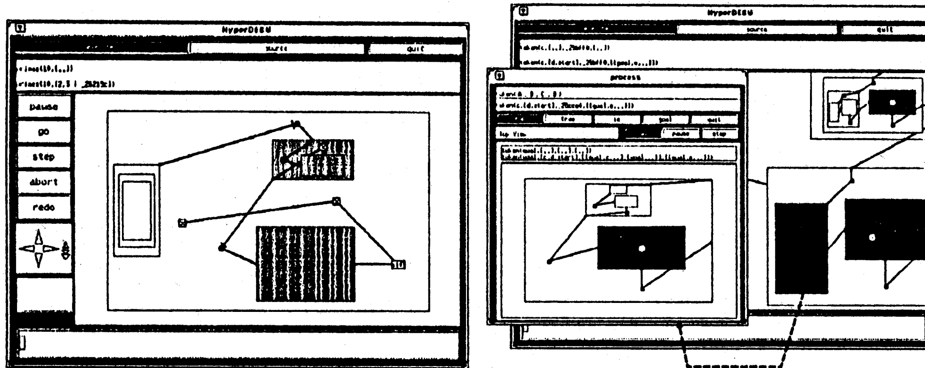

Fig. 8. Example of Control and Data Flow Visualization

7.2 Example of Control and Data Flow Visualization

The left side of two displays in Figure 8 shows an example of control and data flow visualization with "process" and "stream" breakpoints. The process breakpoints are specified at gen/3 and filter/3. The gray rectangles are the "filter" processes, one of which has its stream actions in its own rectangle. Since data flow graphs within a process are visualized in the rectangle of the process, the visualized data flows are made layered and easy to understand.

HyperDEBU supports the magnification of the display and zooming by process windows in order to help a usr's comprehension of complicated control and data flow graphs. The right side of Figure 8 shows an example of complicated graphs. The view of toplevel window is magnified and one of visualized processes is opened as a process window, in which the control and data flows of the process is visualized. The process window can also display the local view to show detailed information.

8 Example of Debugging

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

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

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

token(Node,History,H,T) :- eq(Node,goal,F),
                           token1(F,Node,History,H,T).
token1(true,Node,History,H,T) :- H = [[goal|History]|T].
token1(false,Node,History,H,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].
```

This program searches the paths on the 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.

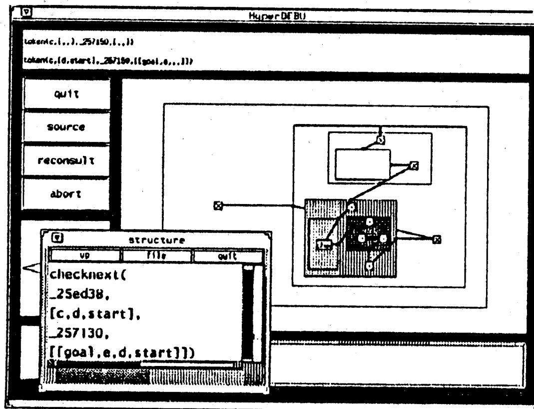

Fig. 9. Visualizing the Erroneous Program : Scene 1

At first, breakpoints for visualization are placed in order to understand the global situation of the execution. The main control flows of this program are composed of creations of `token` goals, and the main data flows are considered as the stream which links all the solutions with a list. So a "process" breakpoint is specified at `token`, and "stream" breakpoints are specified at the third and fourth argument of `token`. Figure 9 shows the visualization of control and data flows on the toplevel window. The nested rectangles represent the processes with respect to `token` goals. These process are linked with a stream in sequence. They searches for solutions in parallel, and when a process finds a solution, it is linked with the stream. However, a process represented as a gray rectangle is suspended and a goal is still linked with the stream in the rectangle. A structure window is opened by clicking this goal, and shows the goal `checknext` is suspended waiting for the data of its argument. This goal stops data flow of the list of solutions.

Then a process window is opened from the gray rectangle in which `checknext` is suspended. Figure 10 shows the opened process window. The rectangle of the process turns to black in the toplevel window. The process window displays I/O tree View to visualize detailed data flows.

This view tells that `checknext` is waiting for the data from `next` which is suspended illegally. Finally, we can detect a bug in the definition of `next`.

In this example, we visualize the global data flow which is used to collect solutions, and find a bug in the other local data flow. Visualizing main global control and data flows helps a user to comprehend the situation of the execution. Then, after zooming in on the location of the bug with a process window, a user can find bugs in its local views of control and data flow.

9 Related Works

As several recent works for the visualization of CCL programs, VISTA [5], Pictorial Janus [6], etc. are researched. They have different policies for visualization from our debugger.

VISA is a performance visualization tool which displays control flows of a CCL program as a colored tree. This tool does not deal with data flow visualization.

Pictorial Janus unifies the program visualization and the visual programming. The execution of a program is fully visualized. Oppositely, when the visualized program is given, the program runs visually. However, to use this system for debugging, a proper abstraction method according to a user's intention and a interactive manipulation facility are required.

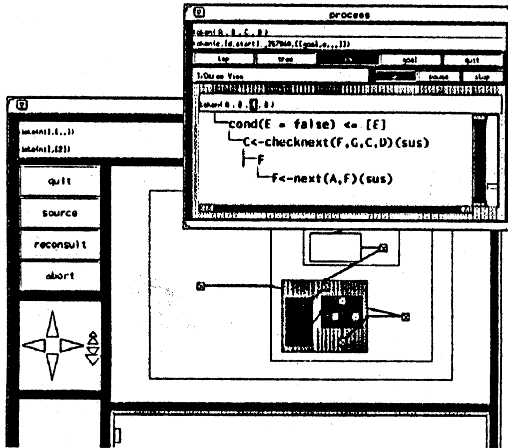

Fig. 10. Visualizing the Erroneous Program : Scene 2

10 Future Work

For more improvement, we have some subjects as follows:

- more sophisticated algorithm to arrange visualized objects.
- grouping visualized data flow objects to reduce complexity of data flow graphs.
- applying the information from a user to static debugging.
- enhancement of functions to display objects, for example, using colors.

11 Conclusion

In this paper, we described the control and data flow visualization facility of a multiwindow debugger HyperDEBU for parallel logic programs. It provides proper information according to a user's intention, and helps to comprehend the situation of the execution. Applying the debugger to examples, we demonstrated that this function is useful to find a bug efficiently.

HyperDEBU is runnable on any Fleng systems since it is written in Fleng itself. Currently, it is running on the Fleng interpreter on UNIX sequential workstations and Mach parallel workstations. Being used in development of application programs, it is being evaluated and improved .

References

[1] Brown,M.H.: Exploring Algorithms Using Balsa-2, IEEE Computer, Vol.21 No.5,pp.14-36 (May 1988).

[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] Tick,E.: Visualizing Parallel Logic Programs with VISTA, International Conference on Fifth Generation Computer Systems 1992, pp.934-942 (1992).

[6] Kahn, K.M. : Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs, International Conference on Fifth Generation Computer Systems 1992, pp.943-950 (1992).