Translation from Transactions of IPSJ

A Program Design Visualization System

ITARU ICHIKAWA*, ETSUO ONO* and TOMOHARU MOHRI*

This paper discusses a program design visualization system, describing the background and explaining program design visualization procedures and the basic system configuration. It focuses on how to correlate a program with graphics, on tracer embedding as the system's basic architecture, and on the structure of graphic objects structure generated internally in the system. It also provides an example of visualization done with an experimental system generated in PSI, with ESP used as the programming language.

1. Introduction

How software operates is vital to supporting software reuse and verifying design and program consistency. As workstation techniques advance, the use of graphic information is gradually replacing that of character information in effectively promoting the understanding of program structure and operation [1].

The visual programming environment is one approach to promoting such understanding; programming is done by supplying graphic information through graphic programming or specification description languages. PegaSys [2], for example, uses descriptions in graphics to generate formatted program design sheets. Other approaches include visual debugging utilities such as VIPS [3] and PROEDIT2 [4], which display program operation by using graphics, with the degree of abstraction at the programming language level.

Algorithm animation, a method close to our approach, makes processing algorithms understandable by using graphics to display program operation. Balsa [5], for example, displays the processing patterns of different sorting algorithms, enabling the principles of operation and efficiency to be compared and understood.

To support software reuse and verify design and program consistency, our program design visualization system visualizes program execution by using graphics that have the degree of abstraction used in design. The degree of abstraction of these graphics is equivalent to that of the graphics used by PegaSys, and higher than that of those used by VIPS and PROEDIT2, enabling users to understand program operation more easily. Our system supports simultaneous visualization from different views, further promoting understanding.

This is a translation of the paper that appeared originally in Japanese in Transactions of IPSJ, Vol. 31, No. 12 (1990), 1801-1811.
*Fujitsu Limited.

The use of graphics in the design stage allows, the system to confirm program and design consistency. It can also support design analysis. We eventually hope to support program reuse by using visualization techniques for understanding the operation of individual program parts.

2. Program Design Visualization Methodology

2.1 Environment

In our study, we assumed the environment shown in Fig. 1. Graphics are used to produce a chart during the specification description and design stages of a program's generation. Correspondence is provided between the program and the original chart. A program's operation is visualized by using design stage graphics and correspondence.

The following are input to the system:
1. Object program
2. Chart definition
3. Graphics and program correspondence

The output of the system is an animation display (Fig. 2).

If the program does not operate as expected, the cause is an important consideration—particularly if it is the program! As an experiment, we embedded a bug in a normally operating program. The result confirmed the display to be abnormal. We made the following assump-

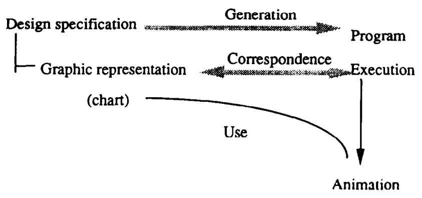


Fig. 1 Environment of program design visualization.

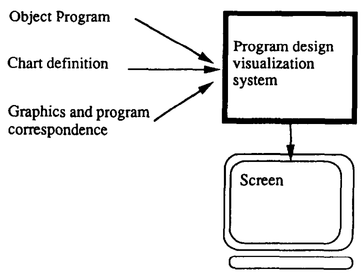

Object Program
Chart definition
Graphics and program correspondence
Program design visualization system
Screen

Fig. 2 Program design visualization input and output.

tion about the foregoing:
Assumption:
The program design visualization system operates normally if a correct input is given, but the validity of correspondence is not ensured.

We assume that the validity of the correspondence has been verified before an input is given to the system.

2.2 Correspondence

Correspondence associates a program with graphics. Here, correspondence is divided into static and dynamic. The correspondence between the program and graphics is defined by visualization instructions, which fall into two types: visualization commands, which establish correspondence with operations, and mapping definitions, which establish correspondence with elements.

A mapping definition defines the following two expressions (p and g) in the format $p \rightarrow g$:
- p: Expression representing an element in the program
- g: Expression representing a graphic element

where p is represented as an expression in the program, and g is indicated by a label in the chart. If graphics with the same label appear in more than one chart, the label of the relevant chart is added to indicate the chart to which the graphic belongs.

The dynamic correspondence defines the correspondence between the operations of the graphics on the chart and execution in the program. For the execution string in the program and the operation string on the chart, we defined a mapping relationship for each substring.

A visualization command defines the correspondence between execution string S in the program and visualization operation A in the graphic structure, by Hoar's axiom $P\{S\}Q, A$, where P and Q are statements (conditional expressions) related to the program status to be satisfied immediately before (P) and after (Q) S is executed. The meaning is interpreted as follows: A is performed if P is satisfied before S is executed and Q is satisfied after S is executed.

Execution string S is defined as follows:
1. *Si*****Sj*: *Sj* is executed after *Si* is executed.
2. *Si*⟨**Sj**⟩: *Sj* is executed while *Si* is being executed.

A is a description, made by using graphic symbols, of the change in graphic elements in the chart. In A, however, the symbols used in programs that appear in P, S, and Q can be used as they are. In this case, we assume that these symbols are mapped in graphics by the mapping definition and then interpreted.

2.3 Program Design Visualization Architecture

Program design visualization displays program operation as animation on a chart. In another approach, with simulation, the program is not actually executed, and only animation is generated on the chart. In our approach, the program is actually executed, execution is extracted and analyzed, and animation is displayed on the chart.

To visualize program execution, in detail and in real time, we face the following problems:
1. Can a display be visualized simultaneously with actual execution?
2. Can the user understand such a display?

The purpose of program design visualization is to support the understanding of program operation. This means that a visualized result should be displayed at a speed at which the user can understand it. It becomes necessary to establish some sort of viable correspondence between computer execution speed and in display speed.

There are two ways of doing this:
1. Execute the program, detect execution information, and record the time series of this information as a log in the secondary storage. Then, extend the time based on this log independently of execution to allow visualization of program operation.
2. Execute the program, synchronizing it with the visualized screen, and detect execution information. Then, extend the time based on this information to allow visualization of program operation.

In the visualization of program logic, the nonuniformity of the time extension is not a serious problem. However, in the visualization of program characteristics, such as synchronization and efficiency, the time extension must be uniform. It is also necessary that visualization should not affect program execution. In this paper, we adopt the second of the above methods, with the objective of visualizing logic operation.

To visualize program execution, three basic architectures are used:
1. Set the execution environment, for example, the interpreter and simulator, for the object language, and then execute the program, obtain execution information, and analyze and visualize the information. This enables detailed information to be obtained, but an execution environment must be prepared for extracting execution information.
2. Directly embed parts of the program for visualizing the program in the object program. Visualization is done directly, on the basis of program execution.

Unlike in other methods, execution information need not be analyzed. However, this architecture cannot handle complicated execution.

3. The third architecture is tracer embedding, somewhere in between the methods above. To collect execution information on the object program, tracers are embedded as software probes in the program. The program is then executed, execution information is obtained from the tracers, and the information is analyzed and visualized. The advantage of this is that the entire execution environment need not be prepared and visualization can be done easily. The disadvantage is that the information obtained may not be very detailed in comparison with that obtained by the first architecture, for example.

We adopted tracer embedding as the basic architecture for visualization, since visualization can be provided easily and complicated program operations can be handled.

Figure 3 outlines the program design visualization system using tracer embedding. The following processing is done:

- Tracers for collecting execution information in the object program are embedded.
- Execution is detected.
- Operation is analyzed by using visualization instructions.
- Operation is displayed as animation on a screen.

To detect execution strings of the object program, the transition of control is observed at several points in the program. Therefore, the program and analysis and display are synchronized and operate as coroutines.

2.4 Chart Structure

A chart is hierarchically composed of units called elementary figures. The definitions of elementary figures are stored in the elementary figure dictionary. The definitions of charts are stored in the chart library. These definitions define the shapes, operations, and constructions of each figure.

A chart treated as a partial figure can be used for chart definition; such a chart is called a partial chart, as opposed to an entire chart.

For the elementary figure dictionary, the system provides abstract data types generally used for software design, such as stack, list, and queue, as basic elements.

In the chart library, a chart references elementary figures and other charts, that is, partial charts, as parts of itself, and is defined unrecursively. The relationship between elementary figures and charts made by the above reference relationship is called the conceptual structure. The chart at the top of this conceptual structure is called the top-level chart.

Each chart corresponds to a specific view. A view is assumed to represent the viewpoint from which the object program is viewed. For example, the views corresponding to the following charts differ according to the viewpoint:

- When the object program is viewed from the data flow.
- When the object program is viewed from the control flow.

In this system, charts from more than one viewpoint can be set for the program. Multiple or multidimensional views can be set for the program corresponding to these charts. The views also differ according to the difference in the degree of detail and abstraction in the chart. This system enables multiple charts to be set for the program. Hierarchical views can be set for the program corresponding to these charts.

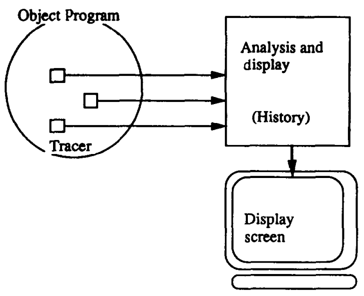


Fig. 3 Tracer embedding method.

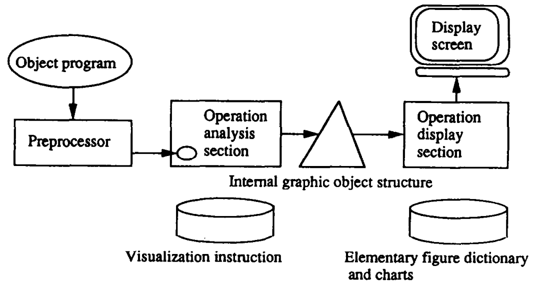

Object program
Preprocessor
Operation analysis section
Internal graphic object structure
Operation display section
Display screen
Visualization instruction
Elementary figure dictionary and charts

Fig. 4 Configuration of basic program design visualization system.

3. Program Design Visualization System

It is difficult to understand a logic program's execution merely by looking at the program, unlike descriptions in conventional programming languages. We developed a program design visualization system [7] for the object-oriented logic programming language ESP [6]. Figure 4 outlines the system's basic configuration.

The system consists of:

- Preprocessor
- Program operation analysis
- Internal graphic object structure
- Program operation display

The visualization instructions, elementary figure dictionary, and chart library explained in the previous section are given to the program as visualization knowledge.

3.1 Tracer Embedding

The visualization system does the following:
- The program operation analysis detects and analyzes the execution of the program processed by the preprocessor.
- It composes a visualization for the internal graphic object structure that models the displayed figures.
- The program operation display shows the program operation in animated form on the screen.

The preprocessor embeds the tracers required to detect operation strings defined by visualization commands. Information on the tracers embedded by the preprocessor is given to the program operation analysis.

The program operation analysis detects execution strings of the object program. When an execution string defined by a visualization command is executed, the operation analysis section composes the visualization for the internal graphic object structure.

A visualization command defines the correspondence between the execution string S in the program and visualization operation A in the graphic structure, in the format $P\{S\}Q$, A. The preprocessor embeds the tracers required to obtain execution information on P, S, and Q. The program operation analysis analyzes execution information from the tracers. When execution string S is obtained, it evaluates P and Q at the beginning and end of S under that execution string. If both P and Q are satisfied, it generates a visualization operation from A.

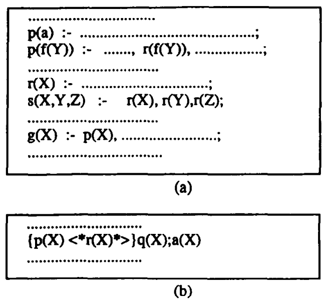
A mapping definition defines the correspondence between program element p and graphic element g in the format $p \rightarrow g$. Using this definition, the system maps the program element obtained by analysis onto the graphic element and obtains the visualization a' shown in the figure.

For the preprocessor to embed tracers, string S of the visualization command defining the execution string is analyzed, and all candidate points for embedding are listed. Then, the number of embedded tracers is minimized by deleting unnecessary points from the program control structure or by merging tracers, so that execution can be detected efficiently. After that, tracers are embedded.

The above processing is explained with reference to the example in Fig. 5. (a) shows the object program. Figure 5(b) shows a visualization command for the program. Its meaning is as follows: If $q(X)$ is satisfied immediately after $r(X)$ is performed during execution of $p(X)$, $a(X)$ is executed.

The candidate points for embedding tracers are as follows:

1. For all predicates and methods that occur in S:
   1) Immediately before calling
   2) Immediately after calling



```
..............................
p(a) :- ...........................................;
p(f(Y)) :- ......., r(f(Y)), .................;
..............................
r(X) :- ................................;
s(X,Y,Z) :- r(X), r(Y),r(Z);
..............................
g(X) :- p(X), ........................;
..............................
```

(a)

```
..............................
{p(X) <*r(X)*>}q(X);a(X)
..............................
```

(b)

Fig. 5 Example of visualization command.

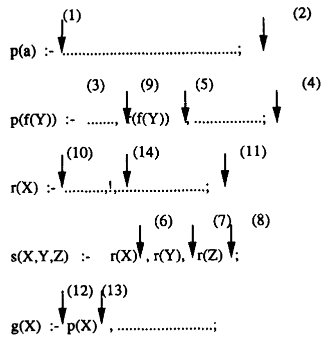

(1)
(2)
p(a) :- ....;
(3)
(9)
(5)
(4)
p(f(Y)) :- ...., r(f(Y)) ....;
(10)
(14)
(11)
r(X) :- ....!,.......................;
(6)
(7)
(8)
s(X,Y,Z) :- r(X), r(Y), r(Z);
(12)
(13)
g(X) :- p(X), ...;

Fig. 6 Candidate positions for embedding tracers.

2. For the body of each clause that defines the predicates and methods:
   1) At the beginning of the body
   2) At the end of the body
3. If there is a CUT "!" in the body:
   1) Immediately after CUT

Tracers need not be embedded at all candidate points.

The candidate points for CUTs are provided for processing Backtrack, which is a control specific to logic programs. If Backtrack occurs, the screen must be restored to the previous state. Each tracer can report an event for execution control (Backtrack occurrence). The program operation analysis has a status history that enables the screen to be restored. When Backtrack passes through a CUT, the tracers that passed between the beginning of the body and the CUT cannot report Backtrack occurrence to the program operation analysis. In this case, the tracer immediately after the CUT is substituted.

Tracers are embedded in each candidate point according to the visualization command shown in Fig. 5. An explanation is given with reference to Fig. 6. $p(X)$ is executed between the paired tracers (1) and (2), and (3) and (4). $r(X)$ ends at each of the tracers (5) to (8). If $p(t)$ is executed, $q(t)$ is evaluated when any of (5) to (8) that becomes $r(t)$ is pased between (1) and (2) and between (3) and (4). If $q(t)$ is satisfied, $a(t)$ needs to be issued. The combination of candidate points is indicated for each visualization command, and unused candidate points are excluded.

Only the calling of $r(f(Y))$ in the first clause can execute $r(X)$ during $p(X)$ execution. It is sufficient to detect it at the tracer of (5). As stated above, the program control structure is analyzed by forecasting an and-or tree, and the combinations of the candidate points that cannot occur are excluded. The values to be picked up at each candidate point are also set.

The definitions of each automaton (how transition is made by each tracer) are generated according to the combinations of candidate points for embedding. Automatons are explained in the next section. Embedding is optimized by optimizing these automatons.

Finally, embedding is optimized by moving or merging tracers within the range in which no problem occurs, in the clause. Optimization is mechanical, but optimization in its broadest aspect is not possible. Any additional optimization must be done manually. This is a problem we hope to resolve in future.

3.2 Program Operation Analysis

When control passes a tracer embedded by preprocessing, the tracer sends a message indicating that control has passed it to the program operation analysis, together with the value of the specified variable at that time. The program operation analysis disconnects this message so that control and the variable value are not influenced by analysis processing.

The program operation analysis then analyzes message information together with its history, generating an automaton for each visualization command, and conducts distributed management for the histories.

The status of the automaton corresponding to the visualization command $P\{S\}Q$; A changes when passing is reported from the tracer related to S. This automaton reaches a final state when the tracer corresponding to the end of S is passed. When each automaton reaches its final state, it evaluates P and Q. If both P and Q are satisfied, it generates a visualization operation a from A.

These transitions are made by passing backward with Backtrack, as well as by passing control forward. For example, if an automaton that has changed to some state is informed that a tracer has been passed with Backtrack, this automaton restores the history by changing to the original state.

By doing repetition or recurrence with program control, more than one partial execution string that matches string S of a visualization command occurs (e.g., $s1, s2, \ldots, sn$). Multiple activation, in which strings overlap, occurs frequently. Enabling sj to be handled before the preceding si ends becomes complicated if only one automaton is generated for that visualization command.

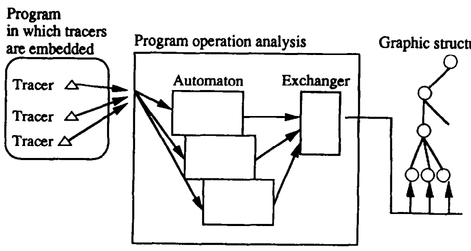

Program in which tracers are embedded
Tracer
Tracer
Tracer
Program operation analysis
Automaton
Exchanger
Graphic structure

Fig. 7 Information flow in analysis.

The system copes with multiple activation as follows:
- It defines an automaton class for each visualization command.
- For each of $s1$, $s2, \ldots, sn$, it generates an automaton of the instance when the first tracer is passed.
- Program operation analysis distributes information from the tracers to the corresponding automatons, according to the status of each automaton.

Distributing information to several automatons makes each definition simpler than when only one automaton is used.

Visualization a generated by the program operation analysis includes program elements that must be mapped onto graphic elements by using the mapping definition. The following actions are also required:
- Obtain the graphic object (in the internal graphic object structure explained in the next section) to which this visualization is to be sent as a message.
- Send the visualization to the graphic object, using the exchanger in Fig. 7, and distribute maps and messages by using the mapping definition and internal mapping table.

3.3 Internal Graphic Object Structure

Our system supports multidimensional and hierarchical views of the conceptual structure consisting of charts. On the visualized screen, the system displays the figures corresponding to each view in multiple windows in animated form through a display model called the internal graphic object structure.

The elements of this structure are active objects of graphics mutually linked in a tree structure. Elements are generated from definitions in the elementary figure dictionary and chart library. The chart of elements at the top of the structure to be displayed is selected during initialization when the system is activated. The selected chart automatically and regressively generates a graphic object in the corresponding internal graphic object structure.

The graphic object corresponding to the chart receives graphic information from a low-order object. It then constructs graphic information from the structure definition defined for it, and sends this to a high-order object (Fig. 8).

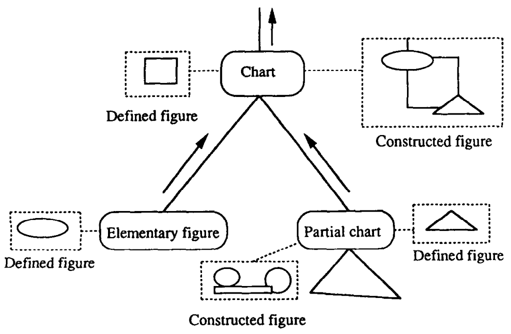


Fig. 8 Internal graphic object structure.

The graphic object corresponding to the chart can be switched so that it behaves like an elementary figure, generates graphic information from the definition of the shape defined for it, and sends information to its high-order object. For a chart on the screen, figures in the partial chart can be constructed and analyzed during the above switching.

The graphic object corresponding to the chart can also display figures constructed for each view in windows by using the animator, which is connected to by the distributer. If the animator displaying a high-order chart is made to display a low-order chart by switching with the distributer (Figs. 9(a) and 9(b)), a zoom-in is made to the low-order level.

When an animator is generated, a corresponding chart is set up and a new window is opened (Fig. 9(c)), and another window is zoomed. Detailed information can be viewed by zooming while switching with the distributer.

In the chart definition, drawings are defined unregressively, but the same drawings are shared by multiple charts. The following technique is used to make view switching and drawing analysis and composition easier:

- In internal drawing construction, each shared structure has a copied structure.
- Visualization messages for objects are copied and distributed to copied objects by the exchanger. Visualization is sent to a leaf of internal drawing construction, interpreted, and transferred to the root.

The animator displays animations in windows. It can detect when a drawing has been picked up on the screen and feed it back so that zoom switching and drawing analysis and composition can be done by selecting a drawing on the display screen.

Displaying multiple windows by zoom switching is effective in helping users understand program operation, but it complicates screen display. Therefore, it is done only when the user needs it. Whether a window is displayed depends on the generation or erasure of the window. The animator is also generated or erased simultaneously.

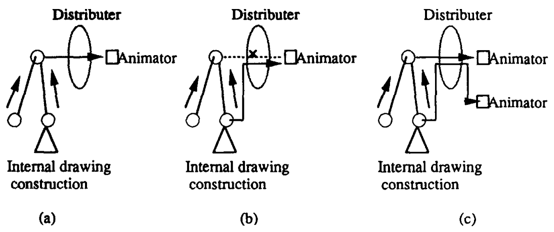

Distributer
Animator
Internal drawing construction
(a)
Distributer
Animator
Internal drawing construction
(b)
Distributer
Animator
Animator
Internal drawing construction
(c)

Fig. 9 View switching with the distributer.

If frequent requests are made by a user for a window to be displayed and erased, the overhead become a consideration. The system reduces this overhead as follows: Once a window or animator has been generated, the window is no longer displayed when nondisplay is requested, but the animator and the window are saved. If display is requested again, the corresponding window is displayed, so a new window need not be generated.

4. Prototype and Experiment

The system prototype we developed is used for programs in the object-oriented logic programming language ESP on a PSI workstation, which we used for an experiment on visualization. The system is written in ESP, and the size of the source program is about 76 K bytes.

The KL1 software simulator [8] for parallel logic programming language execution was used as the object. In the experiment, we used charts actually intended for design. The size of the source program is about 18 K bytes. It requires 28 visualization instructions and displays 11 charts. There are 69 embedded tracers.

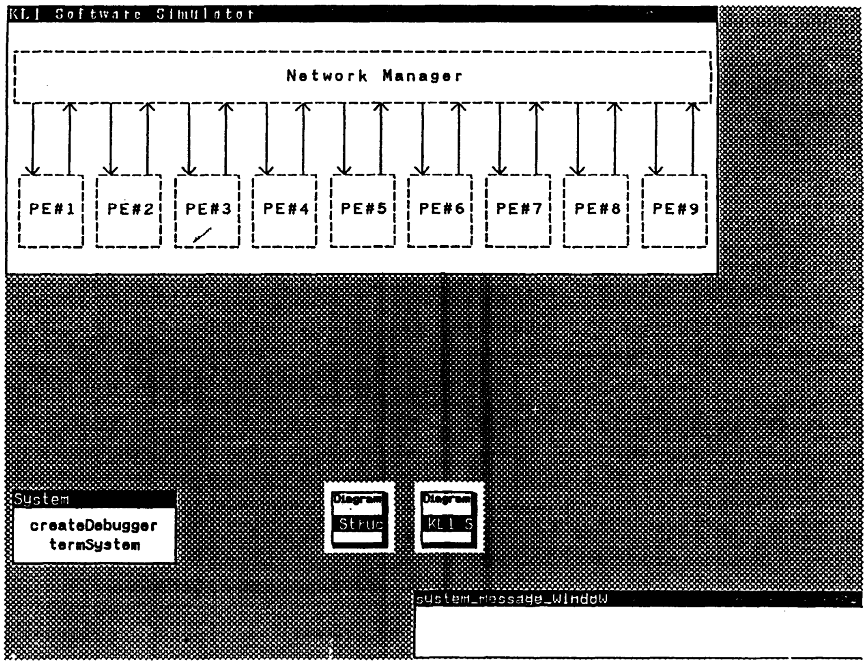
Figure 10 shows a part of the execution example.

(a) The chart of the highest level module configuration for the KL1 software simulator is displayed immediately after activation. In this chart, nine processor elements (PEs) are connected to one network manager in both directions.

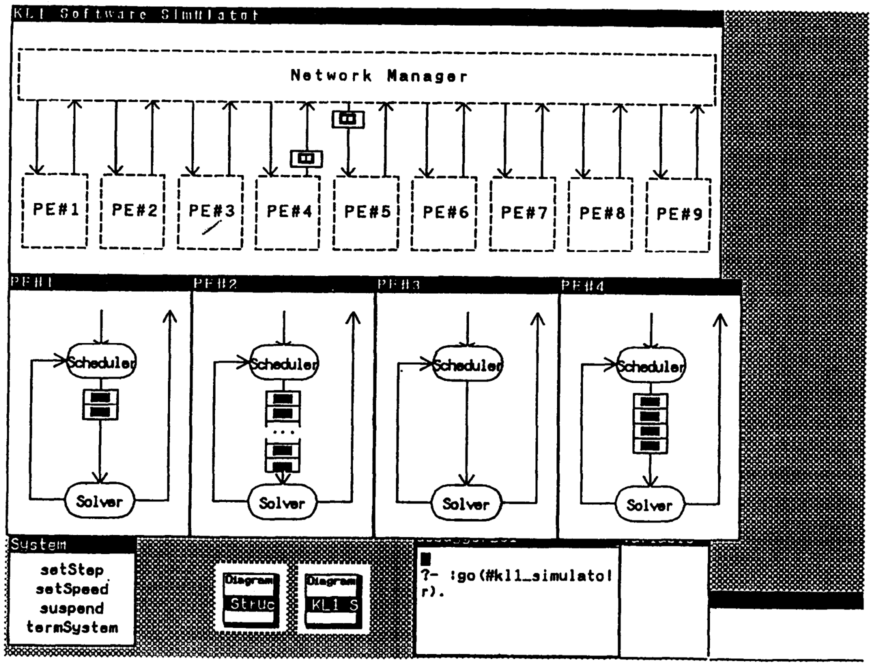
(b) Charts of the internal configuration of some processors are displayed in separate windows. Each processor element consists of a scheduler and a solver. Each scheduler is connected to the corresponding solver by a queue. Data are queued between the scheduler and solver. An abbreviated display (. . .) is used when the amount of data is large.

(c) The queue for which display is abbreviated is zoomed, and its contents are displayed. Because there is no mapping definition to distinguish data, the same drawing is used in the display. However, there is a definition to distinguish communicated data, and therefore different drawings appear in the overall configuration chart.

(d) Charts of the internal processor configuration are displayed in the overall configuration chart, not in

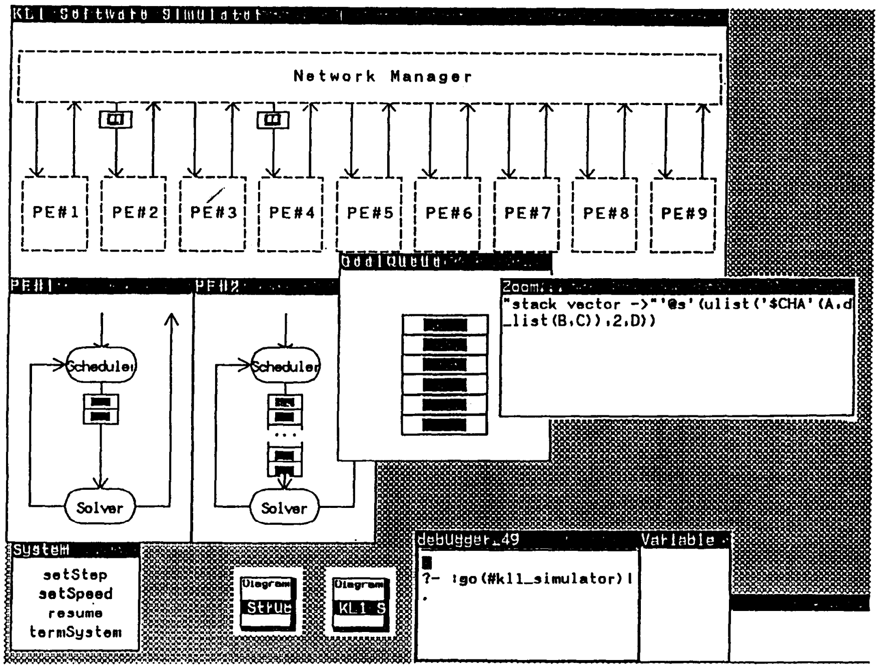


(a)
Fig. 10 Example of visualization on the prototype.



```
?- :go(#kl1_simulato
r).
```

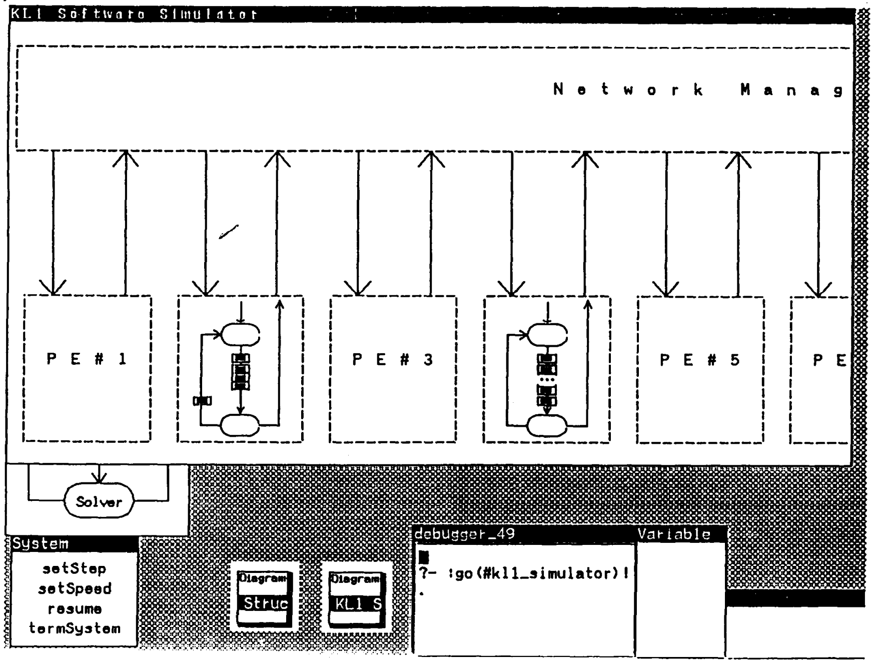
(b)
Fig. 10 (Contd.)



```
"stack vector ->"'@s'(ulist('$CHA'(A,d
_list(B,C)),2,D))
```

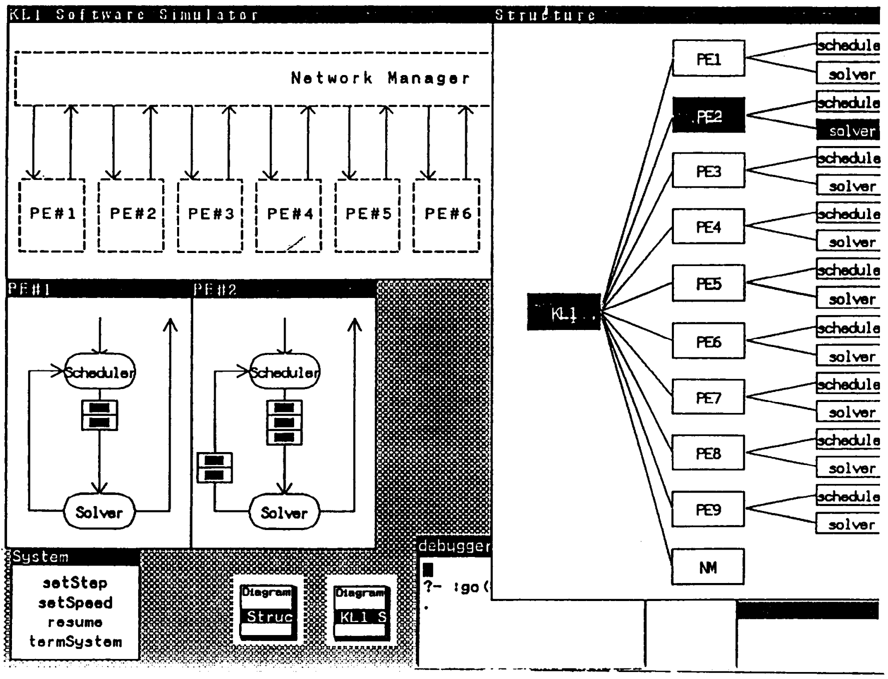
```
?- :go(#kl1_simulator)!
.
```

(c)
Fig. 10 (Contd.)



```
?- :go(#kl1_simulator)!
.
```

(d)
Fig. 10 (Contd.)



(e)
Fig. 10 (Contd.)

separate windows. To make the display easier to understand, the display scale and range of the chart are changed and magnified.

(e) A processing status chart is displayed for Fig. 10(c), indicating the following hierarchy of processing modules:

- PE#1 to PE#9 and the network manager (NM) under KL1, and
- Individual PEs with schedulers and solvers.

The module currently executed is displayed in reverse video. The entire configuration and internal processor configuration are displayed from the viewpoint of the data flow. The processing status is indicated from the viewpoint of the control flow.

Our experiment confirmed that the method described in this paper enabled program operations to be correlated with operations on charts using visualization instructions, even for a practical-scale program.

It also confirmed that visualization of program execution on charts corresponding to multidimensional and hierarchical views made program operation easier to understand.

5. Conclusion

This paper described a system that uses tracer embedding as the basic architecture for program design visualization.

An experimental system that visualized the operations and structure of an ESP program confirmed that program design visualization and the support of multidimensional and hierarchical views were useful in helping users to understand program operation.

Once the validity of the visualization system and visualization instructions has been confirmed, the system can be used to detect errors; that is, an abnormal display indicates an abnormality in the object program. Currently the validity of visualization instructions must be confirmed manually. We assume that the information displayed by visualization instructions is sufficient to help users understand program operation, even though it does not include all the operations of the object system.

In future, we intend to accomplish the following:

- Automatic generation of valid visualization instructions in linkage with a program design system
- Extension of the object language to a parallel processing language.

Acknowledgments

This study was part of the 5th generation computer project. We thank Mr. Hasegawa, head of the First Research Laboratory ICOT, and all who provided us with their valuable support.

References

1. Grafton, G. B. and Ichikawa, T. Visual Programming, *IEEE Computer*, Special issue on visual programming (Aug. 1985), 6-9.
2. Moriconi, M. and Jare, D. F. Visualizing Program Design Through PegaSys, *IEEE Computer*, Special issue on visual programming (Aug. 1985), 72-83.
3. Isoda, S., Shimomura, T. and Ono, Y. VIPS: A Visual Debugger, *IEEE Software* (May 1987), 8-19.
4. Morishita, S. and Numao, M. Prolog Computation Model BPM and Its Debugger PROEDIT2, *Logic Programming '86* (Proceedings of the 5th Conference Tokyo, Japan, June 1986), ed. Wada, E., Lecture Notes in Computer Science 264, Springer-Verlag, Berlin (1987), 147-158.
5. Brown, M. H. Exploring Algorithms Using Balsa-2. *IEEE Computer*, **21**, 5 (May 1988), 14-36.
6. Chikayama, T. Unique Features of ESP, *Proc. of FGCS '84* (1984), 292-298.
7. Ichikawa, I., Ono, E., Mohri, T. et al. Program Design Visualization System for Object-Oriented Programs, *SIGPLAN NOTICES*, **24**, 4 (Proceedings of the ACM SIGPLAN Workshop on Object-Based Concurrent Programming, San Diego, Sept. 20-27 (1988), ed. G. Agha, P. Wegner, and A. Yonezawa (April 1989), 181-183.
8. Ohara, S., Tanaka, J. et al. A Prototype Software Simulator for FGHC, *Logic Programming '86* (Proceedings of the 5th Conference Tokyo, Japan, June 1986), ed. Wada, E., Lecture Notes in Computer Science 264, Springer-Verlag, Berlin (1987), 46-57.