Visual Patterns + Multi-Focus Fisheye View: An Automatic Scalable Visualization Technique of Data-Flow Visual Program Execution

Buntarou Shizuki Masashi Toyoda Etsuya Shibayama
Shin Takahashi
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
Oo-okayama 2-12-1, Meguro-ku, Tokyo, Japan
e-mail: {shizuki,toyoda,etsuya,shin}@is.titech.ac.jp

Abstract

We present a scalable visualization technique for automatic animation of data-flow visual program execution. We also show a framework to assist programmers' browsing tasks by automatically producing the views of execution that highlight significant aspects of the program.

The techniques described in this paper are based on the visual design patterns (VDPs) proposed in our VL'97 paper, which serve as a flexible and high-level structure for reuse of visual parallel programming. This paper shows that VDPs also serve as a basis for representing aspects of the program, with which it is possible to provide scalable views and intelligent assistance for browsing dynamically created data-flow networks.

We have incorporated these ideas into the visual tracer of the KLIEG visual parallel programming environment.

1. Introduction

In the development of parallel programs, programmers often need to examine various aspects of program behavior. We refer to the word `aspects' as particular behaviors of the program (e.g., performance of the program, behaviors of a single component, behaviors of cooperative components communicating with each other) which can be potential targets to be monitored by the programmer. For example, developing a parallel program usually involves a cycle of coding particular processes, correctness checking of the `unstable' processes, and performance tuning of the entire program by searching for bottlenecks. To provide an effective support for these tasks, a scalable and comprehensible visualizer which automatically highlights these aspects will be helpful.

In the case of visual programming languages (VPLs), the visualization technique which is used in both Pictorial Janus[6] and VIPR[3] is able to provide comprehensible views. These systems depict a state of program execution as a picture based on the shape of the program itself, and represent state transitions during execution by smooth animation of those pictures. Furthermore, VIPR partly addresses the scalability issue by incorporating its own single-focus fisheye viewing algorithm.

However, several problems still remain:

support for multiple focal points Practically, a multiple focal point facility is desirable, since programmers often need to simultaneously examine the behaviors of two or more processes (e.g., the sender and receiver processes). With single focus fisheye views, they frequently have to change focus from one process to another.

support for multiple aspects of a program
Programmers need to monitor various aspects of a program and each aspect requires its own view (or its own foci in fisheye cases). A change of aspect (or equivalently, a change of view) occurs frequently, and usually requires tedious zoom-in/out operations.

In this paper, we present a technique that provides a scalable and comprehensible visualization of program execution in data-flow VPLs by exploiting multi-focus fisheye viewing. We also show how we can support programmers' tasks to check the aspects of a program. Our approach is

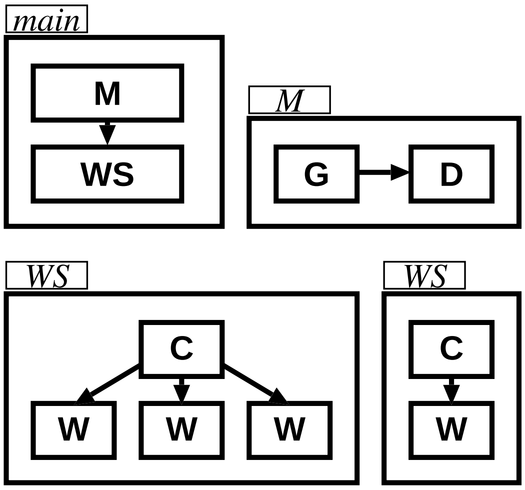

Figure 1. A program in declarative data-flow VPL

based on visual design patterns (VDPs)[14] and browsing tasks are assisted as follows: (1) automatically generated views highlighting certain aspects of the program that the programmers need to monitor; and (2) methods are provided to change views instantly. VDPs include various information, such as the design information that the designers of the VDPs present to the users, the creators, and the last modification times. With this kind of information, we can generate views of a program's execution that emphasize certain aspects of the program.

Below in Section 2 we describe the declarative data-flow VPLs that our method targets and the execution model of the languages. Section 3 shows the method used to animate the execution of programs written in declarative data-flow VPL in a comprehensive manner and to incorporate a fisheye viewing algorithm. Section 4 describes our approach to assist programmers' navigation by aspects. Section 5 shows the implemented visual tracer. After reviewing the related works in Section 6, we summarize this paper and present future works in Section 7.

2. Declarative Data-Flow VPLs

In *declarative data-flow VPLs*, a program is a collection of declarative rules of processes. There are two types of processes: *composite processes* that are visually defined in data-flow diagrams, and *primitive processes*.

A composite process is declared by a set of visual *rewriting rules*, each of which rewrites a process into a data-flow network consisting of processes and data-flow links. A guard may be attached to a rewriting rule and enables/disables the rule accordingly at runtime. A primitive process can be defined in any way (e.g., state transition diagrams or textual languages) and we omit further details.

Figure 1 illustrates skeletons of visual rewriting rules. In this figure, three processes, main, M, and WS are defined[1] . The main process will be reduced to a network consisting of M and WS. The process M will be reduced to a network consisting of G and D. Two rewriting rules are defined for WS. The first rule will rewrite WS into a network consisting of C and three Ws. The second rule will reduceWS into a network of C and one W. Definitions of G, D, C, W, and guards of main, M, WS are omitted from this figure.

Execution of a program written in a declarative data-flow VPL is a sequence of parallel reductions from the root process (a composite main process). A process terminates when all of its subprocesses have terminated. In the above example program, an instance of process M terminates after both G and D have terminated.

In summary, the process network which is created at runtime dynamically changes its topology as the execution proceeds.

3. Animating Execution States

This section illustrates the mechanism for constructing comprehensive animations of program execution in declarative data-flow VPLs (described in Section 2). An animation can be produced by the following three steps:

1. calculate the geometries of dynamically created processes based on the network topologies of the rewriting rules

2. apply a fisheye viewing algorithm to the configured network to display the network within a screen and to provide a browsing interface

3. animate each transition of the fisheyed view after every reduction, step by step

3.1. Configuration of a Process Network

To show the network in a manner intuitively recognizable for programmers, the geometries of newly created processes are calculated from two inputs after each reduction: the location of the reduced process and the network configuration of the applied rewriting rule. When a certain process is reduced, we scale the width and the height of the applied network diagram to just fit to the area that the reduced process occupies.

[1] In this paper, we depict processes as rectangles. However, they can be depicted as any shape (e.g., squares, circles) if they can be fitted to each other by scaling their widths and heights.

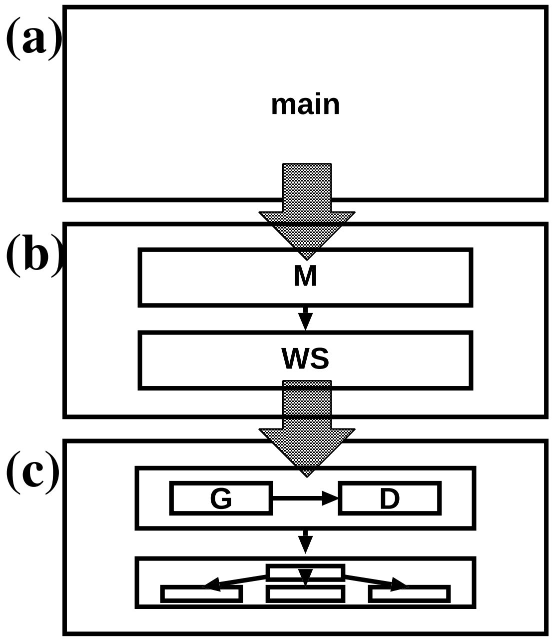

Figure 2. Configuration of newly created processes

Figure 2 shows sequences of network transitions during execution of the program in Figure 1. For example, Figure 2(b) represents the network topology after the reduction of the main process. The locations of the newly created subprocesses (M and WS) are calculated from the network diagram defining the main process (Figure 1).

By laying out the network in the above manner, we can see each snapshot of execution states, which always reflect the topologies of the network diagrams defining the visual programs.

3.2. Applying Fisheye Viewing

The programmer can easily recognize a snapshot of the runtime network configured as described in the previous section. However, the technique as described is not scalable. Even in a small network such as Figure 2(c), we cannot see the detailed behavior of processes (e.g., a C and three W processes in Figure 2(c)). A pan+zoom interface might be a partial solution for the problem. However, programmers may lose track of the currently zoomed location when they are zooming on a portion of the runtime network, particularly when many subnetworks are instantiated from one network diagram during execution.

Focus+context approaches (e.g. [5][7][8][10]) are suitable for navigating such networks, and many variants have been developed. However, all of the focus+context views are not sufficient for displaying runtime process networks and providing navigation interfaces, because several issues described below should be considered in a visualization of program execution. Firstly, any distortion imposed by the algorithms should be minimized, since we want the execution view to be as similar as possible to the shape of the visual program; we want the view to be easily recognized by the programmer. Secondly, we also want to preserve the nested structure of the network and to provide a navigation interface based on that structure, since the nests represent caller-callee relationships between processes, and can be considered a suitable abstraction for navigation. It is also desirable to guarantee the presence of paths to every process, to enable the programmer to examine every live process.

3.3. Smooth Transitions by Animation

Abrupt transitions of view tend to confuse users. This problem can be addressed by animating transitions smoothly. The transition of the topology of the runtime network caused by a reduction can be animated by two steps: computing the geometry of each process before and after the reduction, and linearly interpolating the two geometries in several steps and redrawing the network at each step.

4. Navigation Support by Aspects

As mentioned in Section 1, programmers need to monitor various aspects of program execution. Although they could obtain a layout suitable for each purpose using multi-focus fisheye viewing, their browsing tasks can be reduced if we can:

1. prepare both the view for correctness checking of unstable processes and the view for performance checking, and
2. provide a way to switch views instantly.

In this section we first review the concept of VDPs with an example. Next, we illustrate how aspects of the program can be extracted from VDPs. We will describe how these aspects support their navigation tasks later in Section 5.

4.1. Visual Design Patterns

A VDP is a user-definable data-flow network diagram that has *holes* as parameters and that maintains design information such as described later in Section 4.2. A hole can be instantiated with concrete processes by the user of the VDP. The user can use the network diagram in a rewriting rule (described in Section 2) after instantiating all of the holes.

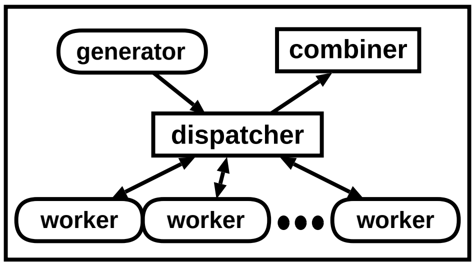

Figure 3. An example of VDP: master worker

As an example of VDPs, we show a rough sketch of the master_worker VDP in Figure 3, which implements a simple load balancing scheme that involves a generator process and a collection of worker processes. The master_worker VDP has some holes (represented as ovals), which are instantiated with concrete processes depending on the problem to be solved.

The network is composed of the generator hole and the combiner process (represented as a rectangle), the dispatcher process, and several worker holes. The Generator simply generates a stream of sub-problems. The dispatcher receives sub-problems from the generator and sends the sub-problems to idle workers. It also receives answers from the workers and forwards the answers to the combiner. Each worker process receives sub-problems, solves them, and returns the answers to the dispatcher. The Combiner receives the answers from the dispatcher and computes the final answer.

By providing appropriate generator and worker processes, we can use this VDP to solve various parallel programming problems, such as ray-tracing and search problems. In this respect, VDPs are appropriate units of reuse.

4.2. Support by Aspects in VDPs

To assist programmers' navigation tasks, we construct process network views, each of which highlights a certain aspect of the program. This is achieved by automatically extracting two types of information: the *layout information* involved in the VDPs used in the program, and the *quality of components*.

Support by Layout Information

The original motivation of VDPs was to visually represent design information, for instance, "which processes (or holes) should be modified to change a particular behavior?" For this purpose, the designer of a VDP can save fisheye-viewed layouts of the VDP with appropriate names. The user can easily discover processes that should be modified by selecting the layout with the name of the behavior.

In each of Figure 4(a) and (b), one layout is depicted of the master_worker VDP. Figure 4(a) is `The problem to solve' layout. This layout emphasizes the processes that should be modified to change the problem to solve, where the generator and the left most worker are magnified and the others are shrunken. Figure 4(b) is another layout `The treatment of answers', where the dispatcher and the combiner are magnified.

According to our experience in using VDPs, a layout of VDPs often corresponds to aspects that the user wants to examine during execution. Since programming with well-designed VDPs only requires instantiating their holes with appropriate processes, the magnified portions represent the processes just instantiated, and they are usually unstable. Therefore, by referencing the layout information, a visualizer can highlight certain parts (in the runtime network) that are relevant to the code that the programmer wants to examine.

In the case of using the master_worker VDP, programmers often have to examine a generator and a worker process simultaneously, after they have finished the coding of the actual generator and worker, in order to check whether each sub-problem is produced and solved correctly. `The problem to solve' layout directly answers this requirement. In turn, they may want to monitor the scheduling of worker processes to discover bottlenecks, for performance tuning. Since the scheduling is controlled by the dispatcher, `The treatment of answers' is useful where both the dispatcher and the combiner are magnified. Therefore, another zooming-in to the dispatcher would be sufficient.

Note that this mechanism may be used in another way. The designer/user of VDPs can also define layouts that are expected to be used mainly in checking the behavior of the VDPs. For example, Figure 4(c) shows a layout that sets a visualizer to display all of the worker processes without abbreviation. Thus, the VDP's layout information can serve as a medium that is used by the user to inform the visualizer of the aspects that he/she wants to monitor during program execution.

Support by Quality of Components

Browsing assistance only by layout information does not sufficiently support programmers in all phases of programming using VDPs.

Programming using VDPs usually starts with an understanding of the behavior of a VDP by instantiating the holes with sample processes provided as a system library. At this stage, programmers often want to monitor the overall behavior of the VDP, rather than the details of the sample processes, in order to recognize the behavior of the VDP.

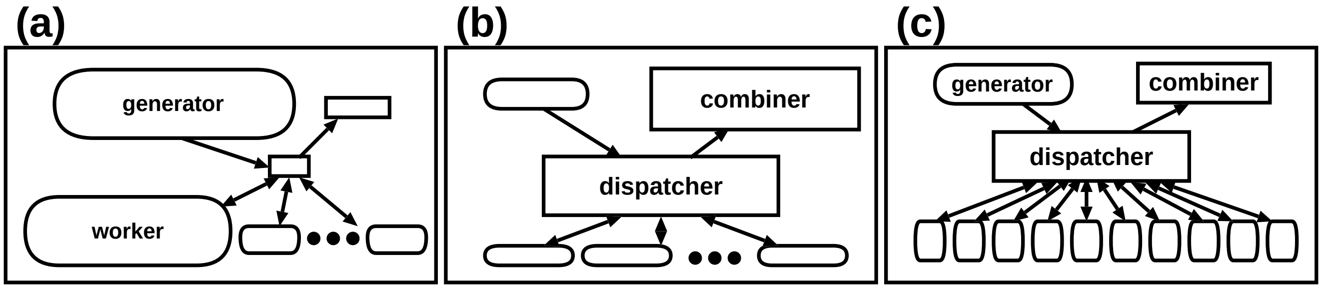

Figure 4. Master worker pattern in three layouts

Next, they instantiate the holes with the processes that they develop. At this stage, they usually need to examine the detailed behaviors of the processes that instantiate the holes, since these processes (which are under development) are unstable and are the main targets to be monitored and debugged.

The quality of components (processes in the data-flow VPLs) serves as a good guide for a visualizer to enhance the network view, by automatically emphasizing several components with low quality, thus reducing programmers' navigation tasks in correctness checking. This is because, in programming with reusable components, a program is mostly composed of reused components, that are considered stable. Consequently, programmers usually have to check only those few components that they have developed.

The quality of a process can be calculated and reflected to the network view by using the heuristics below:

- Codes being currently developed are considered unstable and the main targets to be monitored and debugged. Therefore, the visualizer shrinks the subnetworks created from stable (well-debugged or old) processes to magnify the subnetworks relevant to unstable processes as much as possible.
- The subnetworks created from the processes which instantiate the holes within `VDPs from the system library' will be monitored by the programmer, since the VDPs are the system libraries and can be considered stable, like the C standard libraries.

For this calculation, the VDPs and processes hold the creator and the last modification time.

5. Implementation in KLIEG VPL

We have implemented the supporting mechanism presented in Section 4 into a visual tracer of the KLIEG VPL [14, 12]. Figure 5 shows a snapshot of the execution of an

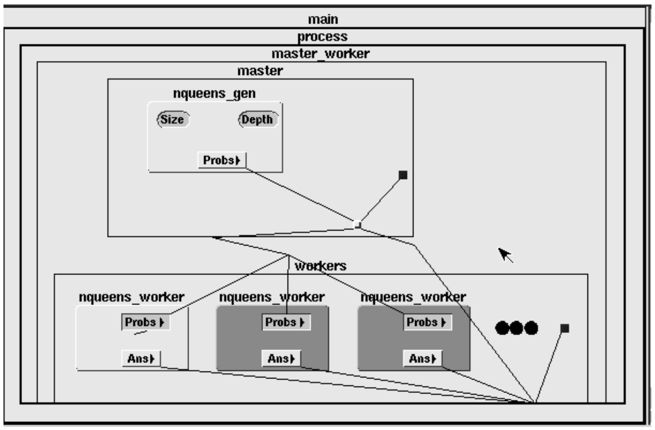

Figure 5. A Snapshop of Visualization on the KLIEG tracer

N-Queens program. The program is derived by instantiating the generator and the worker holes of the master_worker KLIEG-VDP (Figure 6(a)) with an nqueens_gen that generates sub-problems of N-queens and nqueens_workers that solve the problems, respectively.

In the following, we first describe the VDP implemented in KLIEG VPL, and next show the zooming facilities that we utilize. Finally, we describe the browsing assistance implemented in the KLIEG visual tracer.

5.1. VDPs in KLIEG VPL

Here, we briefly describe the master_worker VDP in KLIEG VPL. The master_worker KLIEG-VDP (shown in Figure 6(a) and (b)[2]) is a network constructed from two networks, master and workers.

[2] Both Figure 6(a) and (b) are fisheye-viewed layouts defined in the master_worker KLIEG-VDP, and the internal structure of the workers network hidden in Figure 6(b).

These networks have ports (represented by white rectangles) to communicate with each other. An arrow linking two ports represents a stream that is a continuous data-flow between the ports. For example, master has two ports Wks and Ans to communicate with workers.

The master network is composed of the generator hole and the combiner process, and the dispatcher process as shown in Figure 6(a).

Note that workers is defined using a *replication network* that replicates processes dynamically, and connects those processes. The replicated processes in workers are represented by three holes (recessed rectangles labeled worker), and an ellipsis, which abbreviates a set of processes. Each worker hole has an input port (the recessed rectangle labeled Probs) and an output port (the raised rectangle labeled Ans). Each replicated worker process receives subproblems from the Probs port, solves them, and returns the answers to the master via the Ans port.

A replication network has some special ports that determine the number of replicated processes and the topology of the network. For example, the Wks port in workers is a *map port* that determines the number of processes to generate by the number of received elements from the port, and maps each element to each process. Master (at the bottom of Figure 6(a)) is a merge port that merges the output streams of all the processes.

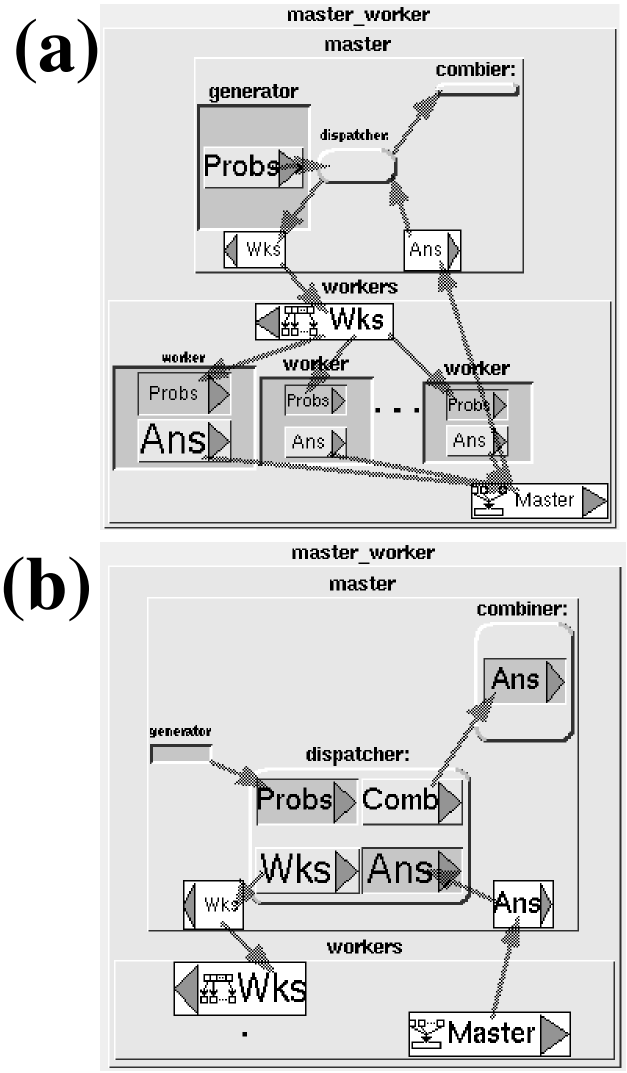

Figure 6. Layouts in KLIEG-VDP

5.2. Visualization on the KLIEG Tracer

Using the Continuous Zoom Algorithm

As shown in Figure 5, the network view, which is distorted by a fisheye viewing algorithm, preserves the topology of the network diagrams depicted in Figure 6(a). Under the observation described in Section 3.2, we have selected the Continuous Zoom[2] as a fisheye zooming algorithm. The continuous zoom is a variant of Furnas's fisheye method[5] and manages 2D hierarchically nested networks consisting of rectangular-shaped nodes and links connecting nodes as shown in Figure 2. The algorithm also has many features: multi-focal points are supported, relative locations of nodes are well preserved, and the presence of a path to every node is guaranteed.

Note that the continuous zoom uses the same operation to animate the transitions caused by users' zoom-in/out operations as the animation described in Section 3.3. Therefore, we can integrate the two animations together. This integration greatly simplifies the implementation.

Semantic Zooming Approach

To avoid drawing unnecessary details of the network and to enhance both readability and drawing performance, we have adopted the semantic zooming approach[4] to display the runtime process network.

The KLIEG tracer shows processes in three manners: rectangles with ports and process names, rectangles with process names, and a mere square. Representation of each process is selected according to the screen space that is assigned to each process/subnetwork by the fisheye viewing algorithm. Actually, processes shown on the tracer represent some portions of the network hierarchy. For example, each nqueens_worker in Figure 5 clusters its subprocesses and represents them as an icon of nqueens_worker. The user can observe the internal networks by zooming-in.

Moreover, we color each process of the network in order to help programmers discover bottlenecks within the network as a whole. Processes are colored to reflect the current state of each process. Runnable, I/O-blocked and dead processes are colored gray, green and black, respectively[3]. In Figure 5, three nqueens_workers are shown on the tracer. The left one is currently running, and is colored light gray. On the other hand, the other two are waiting for another sub-problem, and are colored green. Using this colored network view, programmers can easily monitor the scheduling status of the program.

5.3. Browsing Assistance on the Tracer

Browsing Assistance in the KLIEG Tracer

Figure 7 depicts the browsing assistance facility implemented in the KLIEG tracer. As shown in the center of this figure, we can change the tracer's view by selecting a layout name from the menu.

In this figure, the N-queens program, which is monitored on the tracer, contains the nqueens_gen process that the programmer has modified recently. Thus, the tracer reflects the low quality of the process on the default view of the tracer (shown in the center of this figure) by magnifying an instance of the nqueens_gen process, so the internal subnetwork of the process is shown.

We can adjust the network view easily by menu operations. The center of Figure 7 shows the situation when the user is selecting `The problem to solve' from the menu. The menu operation causes the tracer to change the view as shown at the left part of the figure, by referencing the layout saved by the designer of the master_worker VDP. We can now get the runtime network view in which both the generator and the workers are magnified, and we are ready to check the behavior of the two kinds of processes. If performance checking should be done, we select `The treatment of answers' from the menu. This magnifies both the dispatcher and the combiner, so another single zooming-in

[3] In black and white, these colors correspond to light gray, dark gray and almost black, respectively.

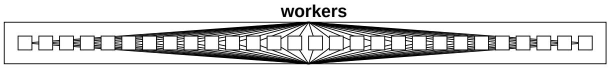

Figure 8. a sequential network created from a replication network

to the dispatcher is enough to obtain the layout shown at the right part of Figure 7.

Support by Semantic Browsing

An ellipsis, which is shown at the bottom half of Figure 5, abbreviates a sequence of worker processes (such as the sequence shown in Figure 8) created from the replication network mechanism in KLIEG-VDPs.

Users can change the abbreviated part within the sequence by mouse dragging, and can easily browse all of the abbreviated processes, one by one. Browsing such a sequence of nodes (as depicted in Figure 8) with fisheye viewing usually requires many zoom-in/out operations and tends to be a tedious task. This shows that the idiom of the underlying VPL (the replication network mechanism in KLIEG VPL) can serve as a template to provide hybrid browsing interfaces along with a fisheye view and can successfully achieve more scalability in browsing. Such hybrid interfaces are called *semantic browsing* interfaces.

6. Related Work

PP[13] is another programming environment that directly displays the execution of a visual program as with both Pictorial Janus[6] and VIPR[3]. Our visualization of program execution is basically the same as those systems, in the sense that an animation of program execution is represented based on the shape of the program. However, we provide high level browsing assistance to support programmers' navigation tasks based on programs' aspects.

The visualization technique described in Section 3 can be applied to visualize other data-flow VPLs that resemble that mentioned in Section 2. Examples of such VPLs include CODE[9], Meander[15], VISTA[11], and V[1].

There have been many efforts to automatically visualize the creation of objects and the messages passed between objects in parallel object-oriented systems. Many studies have also been made for monitoring the performance of the system automatically, without code-instrumentation or annotations. Our visualization technique can be integrated into these visualization systems to provide another comprehensive view of the execution for visual data-flow VPLs.

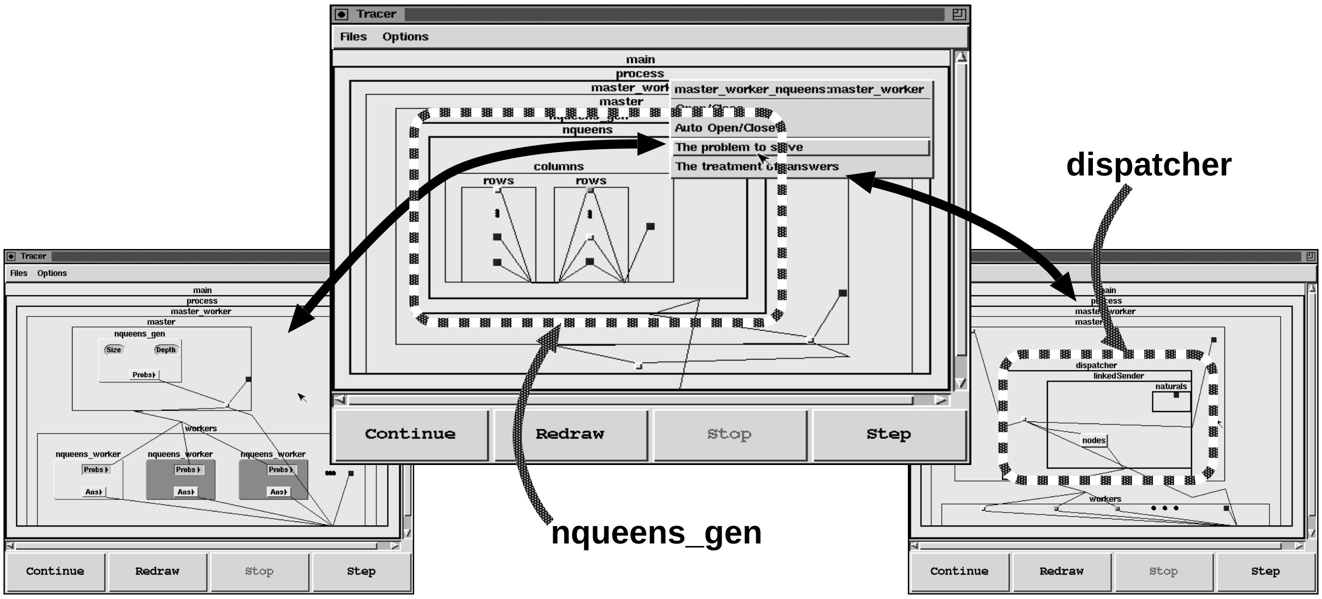

Figure 7. Changing layout of the network

7. Summary and Future Work

This paper presents a visualization technique to automatically animate the execution of the declarative data-flow VPLs. We also show the framework to provide intelligent assistance for browsing the display of execution states based on the aspects. This framework, with a multi-focus fisheye viewing interface, successfully enhances scalability in the visualization of program execution and provides simple but intelligent assistance for the user's navigation tasks.

Although the process network view shown in this paper can easily be extended to incorporate a data browsing facility (for example, zoom-in operation on a stream port expands the port and the received data stream is then shown within the expanded port), we have to explicitly specify a stream link in order to display the data communicated via the stream link in the current implementation. It is, therefore, necessary to develop assistance for data browsing based on the aspects of the program.

References

[1] M. Auguston and A. Delgado. Iterative Constructs in the Visual Data Flow Language. In *Proceedings of 1997 IEEE Symposium on Visual Languages*, pages 152–159. IEEE, IEEE Computer Society Press, 1997.

[2] L. Bartram, A. Ho, J. Dill, and F. Henigman. The Continuous Zoom: A Constrained Fisheye Technique for Viewing and Navigating Large Information Space. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 207–215. Association for Computing Machinery, ACM Press, November 1995.

[3] W. Citrin and C. Santiago. Incorporating Fisheying into a Visual Programming Environment. In *Proceedings of 1996 IEEE Symposium on Visual Languages*, pages 20–27. IEEE, IEEE Computer Society Press, 1996.

[4] K. M. Fairchild, S. E. Poltrock, and G. W. Furnas. SemNet: Three-Dimensional Graphic Representation of Large Knowledge Bases. In R. Guindon, editor, *Cognitive Science And Its Applications For Human-Computer Interaction*, pages 201–233. Lawrence Erlbaum Associates, 1988.

[5] G. W. Furnas. Generalized Fisheye Views. In *Proc. of CHI'86*, pages 16–23. Association for Computing Machinery, ACM Press, 1986.

[6] K. M. Kahn and V. A. Saraswat. Complete Visualizations of Concurrent Programs and their Executions. In *Proceedings of 1990 IEEE Workshop on Visual Languages*. IEEE, IEEE Computer Society Press, October 1990.

[7] J. D. Mackinlay, G. G. Robertson, and S. K. Card. The Perspective Wall: Detail and Context Smoothly Integrated. In *Proc. of CHI'91*, pages 173–179, 1991.

[8] K. Misue and K. Sugiyama. Multi-Viewpoint Perspective Display Methods: Formulation and Application to Compound Graphs. In *Proc. of the Fourth International Conference on Human-Computer Interaction*, volume 2, pages 834–838, 1991.

[9] P.Newton and J.C.Browne. The CODE 2.0 Graphical Parallel Programming Language. In *Proc. ACM Int. Conf. on Supercomputing*, July 1992.

[10] M. Sarkar, S. S. Snibbe, O. J. Tversky, and S. P. Reiss. Stretching the Rubber Sheet: A Metophor for Visualizing

Large Layouts on Small Screens. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 81–91. Association for Computing Machinery, ACM Press, 1993.

[11] S. Schiffer and J. H. Fröhlich. Visual Programing and Software Engineering with Vista. In *Visual Object-Oriented Programming: Concepts and Environments*, chapter 10, pages 199–227. Manning Publications Co., 1995.

[12] E. Shibayama, M. Toyoda, B. Shizuki, and S. Takahashi. Visual Abstractions for Object-Based Parallel Computing. In *Proc. of France-Japan Workshop on Object-Based Parallel and Distributed Computing*, 1997.

[13] J. Tanaka. Visual Programming System for Parallel Logic Languages. In *The NSF/ICOT Workshop on Parallel Logic Programming and its Program Environments*, pages 175–186. the University of Oregon, 1994.

[14] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, and E. Shibayama. Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment. In *Proceedings of 1997 IEEE Symposium on Visual Languages*, pages 76–83, September 1997.

[15] G. Wirtz. Modularization and Process Replication in a Visual Parallel Programming Language. In *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 72–79. IEEE, IEEE Computer Society Press, 1994.