Journal of Visual Languages and Computing (2000) **11,** 529–548
doi:10.1006/jvlc.2000.0177 available online at http://www.idealibrary.com on IDEAL®

Smart Browsing among Multiple Aspects of Data-Flow Visual Program Execution, Using Visual Patterns and Multi-Focus Fisheye Views

BUNTAROU SHIZUKI*, MASASHI TOYODA†,
ETSUYA SHIBAYAMA§ AND SHIN TAKAHASHI§

**Institute of Information Sciences and Electronics, University of Tsukuba, 1-1-1 Tennoudai, Ibaraki, Japan, shizuki@is.tsukuba.ac.jp*
†Institute of Industrial Science, University of Tokyo, 7-22-1 Roppongi, Minato-ku, Tokyo, Japan,
§Department of Mathematical and Computing Sciences, Tokyo Institute of Technology, 2-12-1 Ookayama, Meguro-ku, Tokyo, Japan

Received 1 July 1999; accepted 31 May 2000

This paper presents a scalable visualization technique for automatic animation of data-flow visual program execution, and a software architecture to provide a scalable interface for debugging programs, which exploits a multi focus fisheye viewing algorithm in conjunction with a semantic zooming interface to show various kinds of information at runtime. The architecture also supports users' browsing with the interface by automatically assigning proper focal points, based on information embedded in the debugged programs. Thus, 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

WHEN DEVELOPING A PARALLEL PROGRAM, a programmer needs to examine various aspects of the program behaviors. Examples of 'aspects', which we refer to particular behavior of programs, include the performance of the program, the scheduling policies, the behavior of a single component and the behavior of cooperative components that communicate with each other. Since the development usually involves a cycle of coding particular processes, checking the correctness of the 'unstable' processes, and performance-tuning of the entire program by searching for bottlenecks, these aspects should be frequently monitored. Consequently, it is desirable that debuggers should provide easy access to suitable views to examine those aspects that are of interest to the programmer.

The goal of this research is to develop a visualization technique and its interface for debugging programs in parallel data-flow visual programming languages (VPLs), in which fine-grained processes are dynamically created at runtime. A summary of the requirements of the technique and the interface is presented in the following:

1045-926X/00/100529+20 $35.00/0

1. Multiple Views Facility. To examine various aspects of the execution, one or more proper views should be provided for each aspect.
2. Easy Access to Multiple Aspects of Execution of Programs. A programmer tends to browse several aspects. To this end, the programmer changes views frequently. Therefore, creating and changing views for each aspect must not impose tedious operations on the programmer. Furthermore, the interface used to specify those views should be easy and intuitive for the programmer.
3. Scalable Visualization and its Interface. The visualization and the interface for browsing the visualization must be scalable, since the number of runtime processes increases as the execution proceeds and the processes form large networks. Therefore, the interface must accept visualization of the large runtime networks.
4. Support for Multiple Focal Points. Practically, a multiple focal point facility is desirable, since the programmer usually needs to examine two or more processes (e.g. sender and receiver processes) in the runtime network, simultaneously.

The traditional approach for the first requirement is a multiple window method such as that adopted in ParaGraph [1]. ParaGraph provides various modules, each of which animates a certain behavior of a program or visualizes a measurement of performance of the program. Each of the views is displayed in one window. One merit of this method is that it allows several views of certain measurements of the execution to be examined. However, the multiple window method does not scale-up. While it is necessary to examine various aspects of the execution during the debugging, the programmer's understanding of the relationship among separated views tends to become confused by the separation, even if each view is scalable.

In the field of VPLs, the second requirement is partly achieved by the visualization technique that is used in both Pictorial Janus [2] and VIPR [3]. 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 as an animation by smoothly morphing adjacent pictures. Furthermore, VIPR partly addresses the fourth requirement, by incorporating its own single-focus fisheye viewing algorithm. While both systems achieve automatic animation of the program code being executed and data being transmitted among procedures, neither system has a system framework that provides multiple views displaying various perspectives of a programs execution.

This paper presents an extensible software architecture that provides a scalable, comprehensible and integrated visualization of program execution in data-flow VPLs by exploiting multi-focus fisheye viewing and semantic zooming [4]. We also show how we can assist a programmer's browsing tasks when examining aspects of a program. Our approach to this assistance is: (1) the system automatically generates views, each of which highlights certain aspect(s) of the execution that programmers need to monitor; and (2) interfaces are provided to change views instantly. The views are generated by first extracting various kinds of information embedded in the program, such as the design information, the creators of components, and the last modification times of the components. Then, based on the extracted information, appropriate focal points are assigned to fisheye viewing, along with appropriate views selected to emphasize certain aspects.

To show our approach and the system architecture which achieves our goals, the remainder of this paper is organized as follows. After the description of the declarative

data-flow VPLs, the languages that our technique targets, and the execution model in Section 2, Section 3 shows the technique used to animate the execution of a program written in the VPLs in a comprehensive manner and to incorporate a fisheye viewing algorithm with semantic zooming. Section 4 proposes a software architecture that achieves these issues in an extensible manner. Section 5 describes our approach to assisting the programmer's navigation of aspects of the program, and Section 6 shows the visual tracer implemented to demonstrate the assistance provided. After reviewing related works in Section 7, we summarize this paper with a discussion of future extensions of the system.

2. Declarative Data-Flow VPLs

This section describes the declarative data-flow VPLs that this paper targets, and their execution model.

A declarative data-flow VPL program is a collection of declarative rules for processes. There are two types of processes: *composite processes*, which are visually defined in data flow diagrams, and *primitive processes*.

A composite process is declared by a set of visual *composite rules*, each of which spawns child processes and forms a data-flow network of processes with data-flow links among the processes. A guard may be attached to a composite rule to enable/disable the rule conditionally at runtime. A primitive process is defined in a certain way (e.g. as state transition diagrams or in textual languages), and we omit further details from this paper.

Figure 1 illustrates skeletons of visual composite rules. In this figure, three processes, Main, M, and WS are defined. The Main process creates a subnetwork consisting of two instance processes; one instance of M and one instance of WS. The M process creates a sub-network consisting of G and D. Two composite rules are defined for WS. The first rule creates a network consisting of C and three instances of WS. The second rule creates

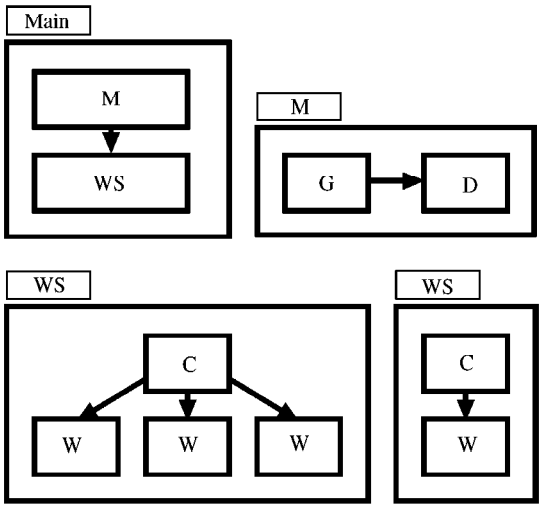


Figure 1. A program in a declarative data-flow VPL

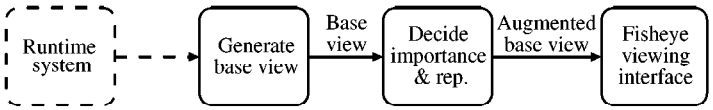


Figure 2. Calculation of the positions and representations of dynamically created processes

a network consisting of C and one W. In general, a process that has more than one composite rule will select one of the rules at runtime. Definitions of G, D, C and W, and guards of Main, M, and WS are omitted from this figure. Note that we depict processes as rectangles in this paper; however, they can be depicted as any shape, such as squares or circles, provided that they can be fitted to each other by scaling their widths and heights.

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

As a result of dynamic creation and termination of processes during execution, the process network dynamically changes its topology at runtime as the execution proceeds.

3. Basic Visualization and the Browsing Interface

This section illustrates the mechanism for constructing animations of program execution in the declarative data-flow VPLs described in Section 2. The basic animation is produced by the following four steps (Figure 2):

(1) Generate a base view of the runtime process network by calculating the positions of dynamically created processes at each step of the execution, referring to the network topologies of the composite rules to show the network in a manner that is easily recognizable to programmers.
(2) Decide the importance and representation of each process/sub-network, to select the processes to be displayed and to show appropriate information that the programmer needs to examine.
(3) Apply a fisheye viewing algorithm to the augmented network to display the network within a screen and to provide a browsing interface.
(4) Smoothly animate transitions of the network topologies, caused both by the execution proceeding and by changes of focal points in the fisheye viewing interface, not to confuse the programmer.

3.1. Define Positions of Dynamically Created Processes to Configure the Runtime Process Network

Our approach to show the network in an easily recognizable manner to the programmer is making the topologies of the network imitate the topologies of the program. To this

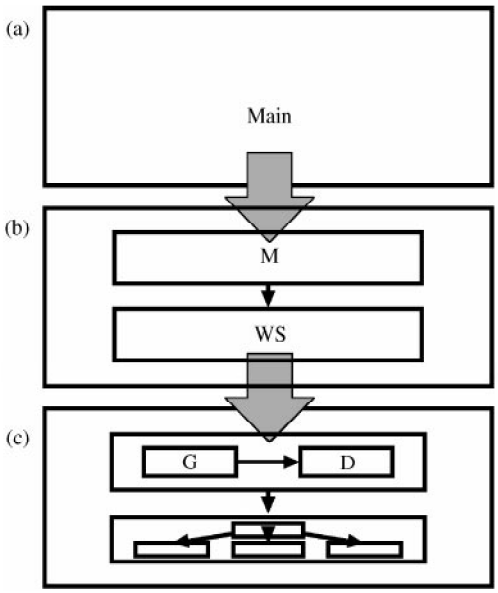


Figure 3. Configuration of newly created processes

end, the positions of newly created processes are calculated from two inputs after each application of composite rules: the location of the parent process and the network diagram of the applied composite rule. When a certain process creates its sub-network, we scale the width and the height of the applied network diagram to fit the area that the parent process occupies.

Figure 3 shows sequences of network transitions during execution of the program in Figure 1. Figure 3(a) and (b) represents the initial network topology before the program runs and the network topology after the rule of the root process has been applied, respectively. The locations of the newly created subprocesses (M and WS) are calculated from the network diagram defining the Main process (Figure 1). The calculation can be written as follows.

```
scale_h = parent.w/rule_fired.w;
scale_v = parent.h/rule_fired.h;
for each (p in processes in rule_fired){
    p_spawned = p.create_untime_process( );
    p_spawned.l = scale_h*p.l + parent.l;
    p_spawned.t = scale_v*p.t + parent.t;
    p_spawned.w = scale_h*p.w;
    p_spawned.h = scale_v*p.h;
```

The above code calculates the coordinates of one or more child processes p_spawned spawned from the runtime process parent when the rule_fired defining the parent process is applied. scale_h is the horizontal scale factor to scale the network diagram of rule_fired linearly to fit the area occupied by the parent process, and is defined by dividing the

width of the parent process parent.w by the width of rule_fired. (rule_fired.w) The vertical scale factor scale_v is derived similarly, from parent.h and rule_fired.h. Then, the position of each p_spawned (p_spawned.l, p_spawned.t), is defined by scaling the position of the corresponding process in rule_fired (p.l, p.t) and moving it by the position of the parent process (parent.l, parent.t) as the offset to the screen.

3.2. Importance and Representation of Process and Sub-Network

Since screen space is limited and the number of processes in a runtime network increases as the execution proceeds, we have to select processes to be displayed. We also want to display in the runtime network various information about the execution, such as scheduling status and performance data. To make such a selection and display, we define two parameters for each process: a number that expresses its relative importance and a 'representation function' that decides the proper representation for the amount of space assigned to the corresponding process. Note that the effect of representation functions is similar to the semantic zooming approach [4], provided that the functions show as much detailed information as possible when enough space is assigned, while suppressing unnecessary details and showing some informative signs to indicate that 'something is there' when the assigned space is low.

3.3. Applying Multi-Focus Fisheye Viewing

The technique described in Section 3.1 is not scalable only in itself. Even in a small network such as Figure 3(c), we cannot see the detailed behavior of processes [e.g. C and three W processes in Figure 3(c)]. A pan + zoom interface might be a partial solution to the problem. It scales the original network linearly and provides scrollbars that allow the user to move part of the scaled view. However, the user may lose track of the current location when zooming on a portion of the runtime network, particularly when many sub-networks are instantiated from one rule, and are thus similar.

In contrast, focus + context approaches (e.g. [5–9]) are powerful for navigating such networks, and many variants have been developed. In particular, the Continuous Zoom [9] has the properties below and is considered a suitable algorithm. First, the algorithm supports a multiple focal point facility. Second, it guarantees the presence of paths to every network node (i.e. process), even to hidden nodes. This allows the user to examine every node. Another desirable property is that the algorithm preserves the nested structure of the network and provides a navigation interface based on that structure. Thus, the algorithm can reveal caller–callee relationships between processes in the process network hierarchy, such as depicted in Figure 3.

The Continuous Zoom algorithm can be simply applied by treating each of the processes (i.e. the rectangles depicted in Figure 3) as a node of the algorithm, and assigning the importance of the process as the degree of interest (DOI) of the corresponding node.

Note that the representation functions described in Section 3.2 should change the appearance of the corresponding node according to the area that is assigned by the viewing algorithm. Thus, it is possible to integrate a semantic zooming interface with a fisheye viewing interface.

3.4. Smooth Transitions using Animation

Abrupt transitions of view tend to confuse the user. This problem can be addressed by animating transitions smoothly. The transition of the topology of the runtime network caused by the application of rules can be animated in two steps: computing the position of each process before and after the application, and linearly interpolating the two geometries in several steps, redrawing the network at each step. Note that the positions of processes that are newly created by an application are defined as the center of the parent process.

3.5. Integration of Fisheye Viewing with Multiple Semantic Zooming: Merits and Demerits

With the visualization and its browsing interface described above, we can obtain detailed views of the dynamically created process network at execution, while seeing the shape of the whole network in a scalable manner. Moreover, multiple views can be achieved if an appropriate set of representation functions are provided, each of which shows semantic information (such as its execution status and a performance measurement) related to the specified process in a semantic zooming manner.

Suppose that a representation function places the subprocesses of a process in the same manner as that described in Figure 3, and shades each of the subprocesses based on its execution time (summation of the elapsed time of all of its descendant processes). If we specify all of the processes to be shown by the representation function, we will be able to see some 'hot' processes within the runtime network at a glance from the gradation of processes within the network. Then we can further examine those 'hot' processes, using zooming to display the same kind of details for the descendants, while viewing adjacent processes and the overall network using fisheye viewing. Thus, the most problematic processes can be probed gradually, using semantic zooming and the fisheye viewing interface.

Such performance monitoring requires integration, and cannot be achieved solely by multiple viewing (such as in ParaGraph) or by animation of the execution with a fisheye viewing interface (such as VIPR).

However, we may still have to zoom in and out of several parts of the runtime network, to select the appropriate view for each of the parts, to obtain our desired view. Moreover, changing those views is a time-consuming task, and many changes are required to examine various aspects of the program. Some supporting mechanism is therefore required.

4. System Configuration

Figure 4 illustrates the software configuration for implementing the mechanism described in Section 3.

The architecture is designed to provide a variety of views, each of which shows specific information about the execution of the program, in an extensible manner. To achieve this goal, there is a separated component *view generator*, which generates views from the output of other system components (*base view generator* and *runtime system*).

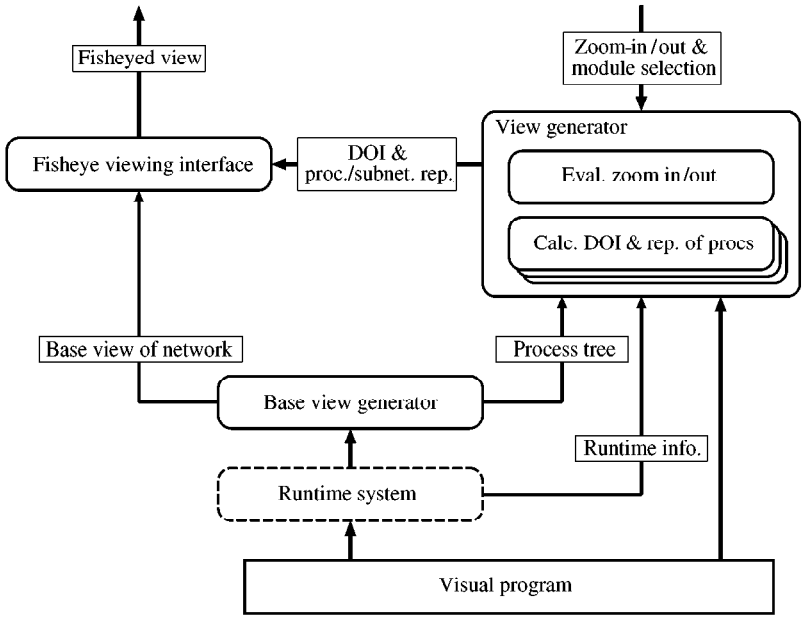


Figure 4. System architecture

The view generator has several modules as its sub-components, each of which shows specific information about part of the runtime network. Thus, we need only to add modules to the view generator, if we want to extend the system and support other kinds of views.

The three main components, the base view generator, view generator and fisheye viewing interface, are described below.

4.1. Base View Generator

The base view generator generates the base view of the runtime process network, as described in Section 3.1, at each step of the execution. At the same time, to convey various parameters to the view generator, it also constructs a *process tree* structure. Figure 5 depicts an example of the process tree structure, which corresponds to the network in Figure 3(c). Each node of the tree corresponds to a runtime process and stores those parameters, such as the default representation of the process and execution state: running, suspended or dead.

4.2. View Generator

The view generator assigns importance and representation to each process in the network hierarchy and feeds them to the fisheye viewing algorithm, based on the parameters stored in the nodes of the process tree. More concretely, the parameters below are assigned for each node of the process tree by modules in the view generator:

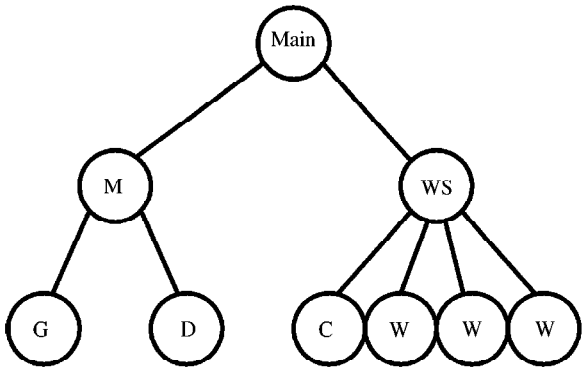


Figure 5. Process tree

- *DOI*: A number which indicates the relative importance of the corresponding process.
- *representation function*: A function that decides the proper representation for the amount of space assigned to the corresponding process.
- *visibility* (*'open'*/*'close'*/*'not-specified'*): A value that explicitly tells the fisheye viewing algorithm that the contents of the corresponding network hierarchy must be visible ('open') or invisible ('close'), or that the visibility depends on the algorithm ('not-specified').

The above parameters are assigned after a module is activated by the user. When the user specifies a module to be activated to a process (i.e. the corresponding node of the process tree), the module then re-calculates the DOIs of the specified node (and its descendent nodes if necessary) and re-assigns the representation function of the module and an appropriate visibility to the node.

4.3. Fisheye Viewing Interface

The fisheye view algorithm distorts the base view according to the DOI of each part of the network hierarchy and shows it within the screen. The appearance of each part of the network hierarchy is defined by the representation function that is set by the view generator.

5. Navigation Support by Exploiting Aspects

As mentioned in Section 1, programmers need to monitor various aspects of program execution. Although they could obtain a layout suitable for each aspect by assigning several focal points on the multi-focus fisheye viewing interface and selecting appropriate modules for semantic information, their browsing tasks can be alleviated if we can:

(1) automatically generate views, each of which highlights certain aspects to be checked (e.g. views for checking correctness of unstable processes and views for performance-tuning); and
(2) provide an interface for choosing from those views easily.

Although it is virtually impossible to generate all of the views highlighting aspects necessary for the programmer, it is possible to automatically generate several views essential to debugging. An example of those views is one that emphasizes processes considered to have bugs. Another example is a view that highlights processes that include faults and those relevant to the computation of the faulty processes.

Our approach for the automatic generation of such views is based on visual design patterns (VDPs) [10]. 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 time. From this information, we can generate views of a program's execution that emphasize certain aspects of the program.

In the following sections, we illustrate how these aspects of the program can be extracted from VDPs, after reviewing the concept of VDPs with an example. Interfaces to select one of the views generated are described later in Section 6.

5.1. Visual Design Patterns

5.1.1. Constructs of VDPs

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

As an example of a VDP, we show a rough sketch of the master-worker VDP in Figure 6, 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 generates a stream of sub-problems. The dispatcher receives the sub-problems from the generator and sends them 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.

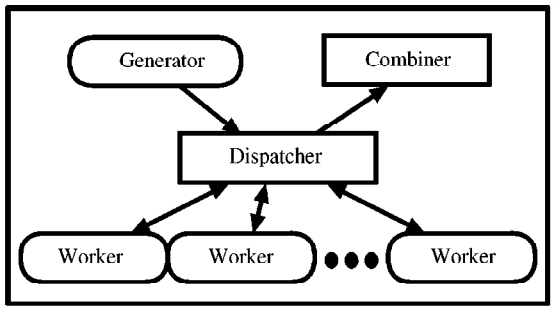


Figure 6. An example of VDP: master-worker

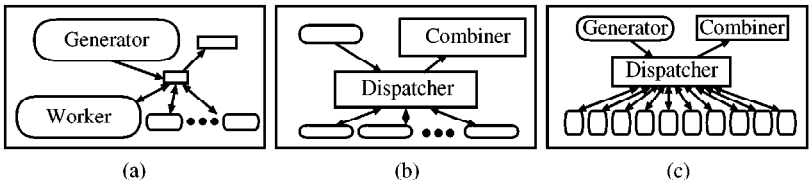


Figure 7. Master-worker pattern in three layouts

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 for reuse.

5.1.2. Layout Information of VDPs

VDPs 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 the processes that should be modified by selecting the layout with the name of the behavior.

In each of Figures 7(a) and (b), one layout of the master-worker VDP is depicted. Figure 7(a) represents the 'Problem to solve' layout. This layout emphasizes the processes that should be modified to change the problem to be solved, where the generator and the left-most worker are magnified and the others are shrunken. Figure 7(b) represents another layout, 'Treatment of answers', where the dispatcher and the combiner are magnified.

5.1.3. Defining a VDP

To define a VDP, the designer first defines a network diagram by placing processes and/or holes and drawing data-links between them. When the designer saves a fisheyed layout, it is possible to optionally assign one of the modules in the view generator to the processes/holes that the VDP comprises.

In the VDP depicted in Figure 7, the designer can assign to the sequence of holes [Figure 7(c)] the module that has a representation function that colors the associated process base on its execution at runtime. The resulting layout will help the users of the VDP to examine whether load-balancing works correctly at runtime, as described later in Section 5.2.

5.1.4. Programming using VDPs

Programming with VDPs usually starts with understanding the behavior of existing VDPs by instantiating the holes with sample processes provided as a system library. After understanding the behavior, the programmer instantiates holes with the processes that the programmer developed. After instantiating all of the holes, the programmer can use the resulting network in a composite rule (described in Section 2).

5.2. Generating Views from Information in VDPs

Currently, two kinds of views based on the information in VDPs are generated: views generated by referring to layout information in the VDPs and views that reflect the quality of the processes.

5.2.1. Views from Layouts in VDPs

Our experience in using VDPs suggests that a layout defined by the designer of a VDP often corresponds to an aspect that the user of the VDP wants to examine during execution. Since programming using well-designed VDPs consists mainly of instantiating their holes with appropriate processes, the magnified portions in VDP layouts represent the processes just instantiated, which are usually unstable. Therefore, by referring to the layouts, a visualizer can highlight certain parts (in the runtime network) that are relevant to the code that the user needs to examine, thus assisting by greatly reducing the tasks required to assign focal points in the fisheye viewing interface.

In the case of using the master-worker VDP, the user often has to examine a generator and a worker process simultaneously, after they have finished coding their own 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. Since the scheduling of worker processes is controlled by the dispatcher, the 'Treatment of answers' layout is useful, where both the dispatcher and the combiner are magnified. Zooming-in on the dispatcher would be sufficient for this purpose.

5.2.2. Layouts by Quality of Components

The quality of components (i.e. processes in the data-flow VPLs) can be used as a tool to make our visualizer enhance the network view, because of the two observations below:

- In order to recognize the behavior of a VDP, the user of the VDP often wants to monitor the overall behavior of the VDP, rather than the details of each process.
- After instantiating holes within the VDP, the user usually needs to examine the detailed behavior of the processes that instantiate the holes, since these processes (which are under development) can be considered unstable and are the main targets to monitor and debug.

In programming using reusable components (i.e. the processes in this paper), a program is composed mostly of reused components, which are considered stable. Consequently, the programmer usually has to check only the few components that the programmer has developed thus the qualities are unknown. Therefore, by automatically emphasizing the components whose qualities are unknown (possibly low), it is possible to reduce the programmer's navigation tasks for checking the correctness of the components.

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

- The code currently being developed is considered unstable and is the main target to monitor and debug. Therefore, as far as possible, the visualizer shrinks the sub-networks created from stable (well-debugged or old) processes and magnifies the sub-networks that are relevant to unstable processes.
- The sub-networks created from the processes that instantiate the holes within 'VDPs from the system library' will be monitored by the programmer, since the VDPs in the system libraries can be considered stable, like the C standard libraries.

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

6. Implementation in KLIEG VPL

We have implemented the architecture described in Section 4 as the visual tracer of the KLIEG VPL [10, 11]. Figure 8 shows a snapshot of the animation representing the execution of an N-Queens program. The program is derived by instantiating the generator and the worker holes of the master_worker KLIEG-VDP [Figure 9(a)] with an nqueens_gen that generates sub-problems of N-queens and nqueens_workers that solve the problems, respectively.

In this section, we first describe the KLIEG-VDP implemented in KLIEG VPL, and next show the browsing assistance based on aspects implemented in the KLIEG visual tracer.

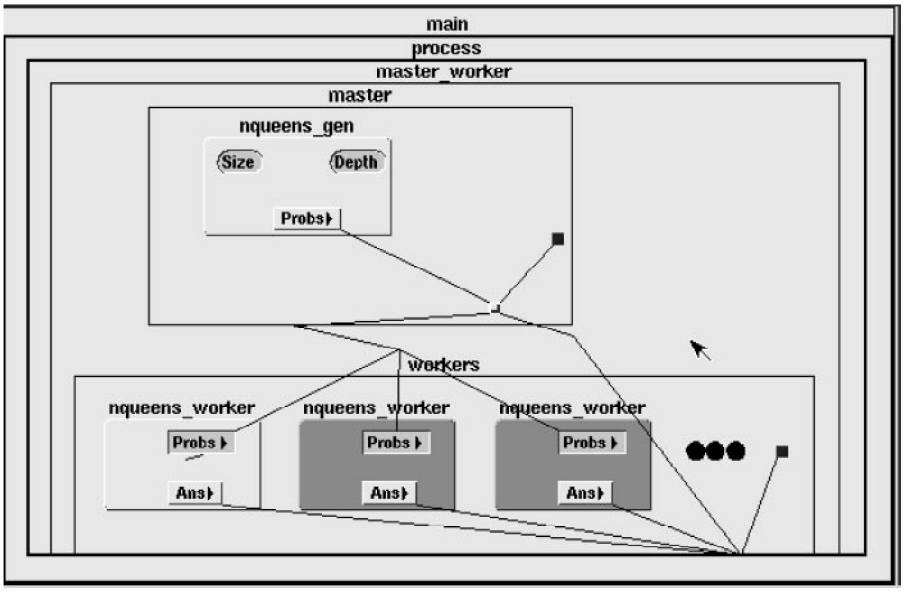
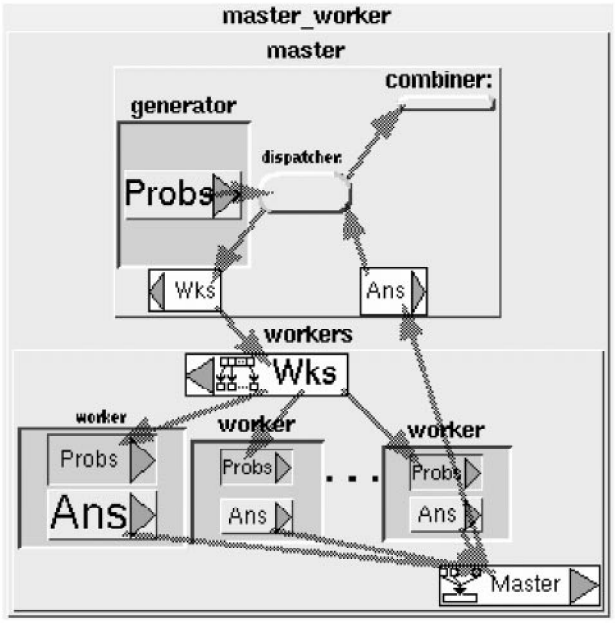
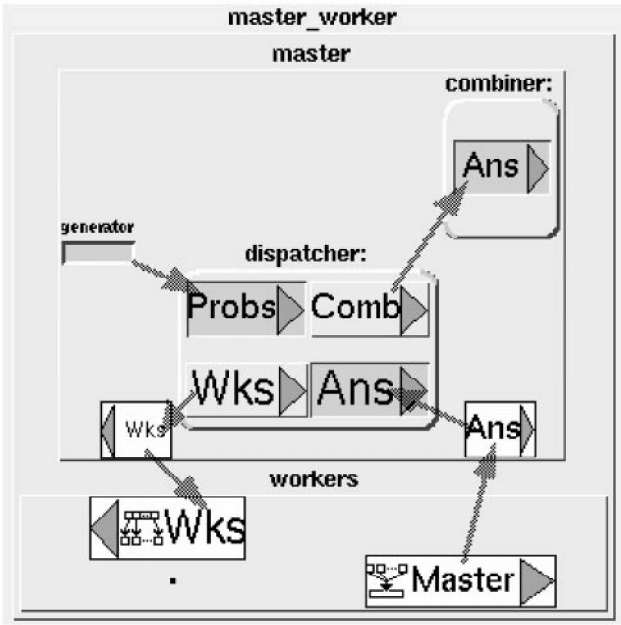


Figure 8. A snapshot of visualization on the KLIEG tracer



(a)



(b)

Figure 9. Layouts in KLIEG VDP

6.1. VDPs in KLIEG VPL

The master_worker KLIEG-VDP [shown in Figure 9(a) and (b)] is a network constructed from two networks, master and workers. 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, the combiner process, and the dispatcher process as shown in Figure 9(a).

Note that workers are 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 sub-problems 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 9(a)] is a merge port that merges the output streams of all the processes (details are described in our VL97 paper [10]).

6.2. Zooming Interface of KLIEG Tracer

As shown in Figure 8, the network view on the visual tracer, which is distorted by the Continuous Zoom algorithm, preserves the topology of the network diagrams depicted in Figure 9(a).

Note that the continuous zoom uses the same operation to animate the transitions caused by the user's zoom-in/out operations as the animation described in Section 3.4. Therefore, we can integrate the two animations together. This integration greatly simplifies implementation.

6.3. Modules in the View Generator

Currently, there are four modules in the view generator: 'Evaluate zoom in/out', 'Restore saved layouts', 'Calculate code quality' and 'Terminate Processes.' The module 'Evaluate zoom in/out' is activated by users' zooming in/out operations (currently, left button for zooming in, and right button for zooming out). Other modules, which construct views reflecting certain kinds of aspects, are activated in response to users' selections (currently, pop-up menus on visible processes). One exception is 'Terminate Processes.' The module is activated when a process terminates and it changes the representation of the process to indicate that the process is now dead.

More detailed descriptions of the modules are as follows:

- The '*Evaluate zoom in/out*' module increases and decreases the DOI of the node in response to the user's zoom-in and out, respectively. The visibility and the representation of the process are not changed.

- '*Restore saved layouts*' receives the name of saved layouts of a VDP and resizes runtime processes by distributing the DOI of the manipulated process to the descendant processes. The DOIs of each of the descendant processes are assigned to be proportional to the area of the corresponding process in the saved layout.
- '*Calculate code quality*' calculates the code quality of the descendant processes of the selected process, as described in Section 5.2.2. The DOIs of each of the descendant processes are assigned to be proportional to the code quality of the processes.
- '*Terminate processes*' shrinks a dead process by assigning 0 (the minimum value of DOI in the current implementation) as the DOI, to give as much screen space as possible to other live sibling processes.

Besides those modules, parameters for a newly created node are defined by the system as follows:

- *DOI*: 1.
- *visibility*: 'not-specified'.
- *representation function*: the default representation function determines the representation of the associated process according to the screen space that is assigned to the process by the fisheye viewing algorithm. When the process is assigned enough space to show the children of the process, the function shows the children. Otherwise, it shows one of three icons: a rectangle with the ports and the name of the process, a rectangle only with the name of the process, and a simple square. Moreover, in order to help the programmer easily discover bottlenecks within the network, the function colors the icon to show the scheduling status of the process: runnable, I/O-blocked and dead processes are colored gray, green and black, respectively. (When viewed in black and white, these colors correspond to light gray, dark gray and almost black, respectively.)

In Figure 8, three nqueens_workers are shown on the tracer. The one on the left is currently running, and is colored light gray, whereas the other two are waiting for another sub-problem, and are colored green. From this default colored network view, the programmer can briefly monitor the scheduling status of the program.

Note that the user can examine hidden parts within the network by enlarging the part with zooming-in. For example, each nqueens_worker in Figure 8 clusters its sub-processes and represents them as an icon of nqueens_worker. The user can observe the internal networks by zooming-in. In this way, the default representation function is effective in preventing unnecessary details of the network from being drawn, thus enhancing both the readability and performance of drawing.

6.4. Browsing Assistance in the Tracer

Figure 10 depicts the browsing assistance facility implemented in the KLIEG tracer.

The N-queens program, which is monitored by the tracer, contains the nqueens_gen process that the programmer has modified recently. As a result, the unknown (possibly low) quality of the process is reflected in the tracer's view (shown in the center of this figure) by magnifying an instance of the nqueens_gen process, so that the internal sub-network of the process is shown.

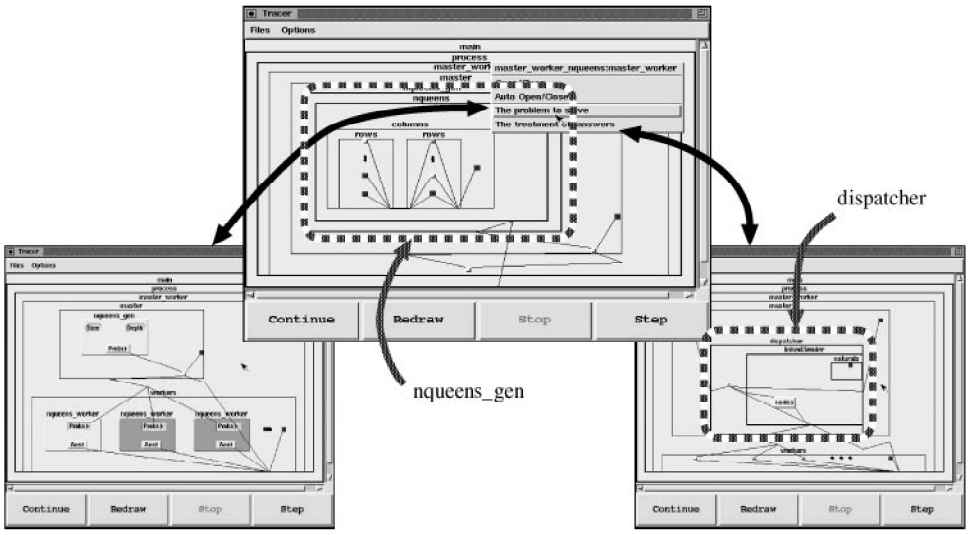


Figure 10. Changing the layout of the network

We can easily adjust the network view by selecting items from the menu as shown in the center of Figure 10. The figure shows the situation when the user is selecting 'Problem to solve' from the menu. The menu operation causes the tracer to change the view as shown in the left part of the figure, by referring to the layout saved by the designer of the master_worker VDP. Now we can get a runtime network view where both the generator and the workers are magnified, and we are ready to check the behavior of the two kinds of processes. If the performance is required to be examined, we select 'Treatment of answers' from the menu. This selection magnifies both the dispatcher and the combiner, so a further single zoom-in to the dispatcher is enough to obtain the layout shown in the right part of Figure 10.

6.5. Support by Semantic Browsing

Although it is possible to browse any shape of network using a fisheye viewing interface with semantic zooming, it is also possible to combine other kinds of interfaces that are suitable for the specific shape of networks with fisheye viewing. Furthermore, the idiom of the underlying VPL (the replication network mechanism in KLIEG VPL) can serve as a template to provide such 'hybrid browsing interfaces'.

An ellipsis, which is shown in the bottom half of Figure 8, is an example. The ellipsis abbreviates a sequence of worker processes (such as the sequence shown in Figure 11) created from the replication network mechanism in KLIEG-VDPs. Browsing such a sequence of nodes only with fisheye viewing usually requires many zoom-in/out operations and tends to be a tedious task, since the user has to enlarge each node one by one. On the visual tracer, the user can change the abbreviated part within the sequence by dragging with the mouse, and can easily browse all of the abbreviated processes.

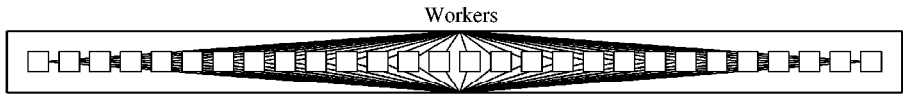


Figure 11. A sequential network created from a replication network

We call such hybrid interfaces *semantic browsing* interfaces. Note that semantic browsing interfaces should be designed in such a way that they do not interfere with the properties of the Continuous Zoom algorithm, such as the multiple focal points and paths to every process, described in Section 3.3. The above semantic browsing interface satisfies these requirements by providing: (1) the function to enlarge two or more processes in a sequence and (2) the function to view the contents of enlarged processes by zooming in on those processes.

7. Related Work

PP [12] is another programming environment that directly displays the execution of a visual program, as do Pictorial Janus [2] and VIPR [3]. Our visualization of program execution is basically the same as in those systems, in the sense that animation of program execution is represented as pictures that are derived from the shape of the program. However, we provide multiple views, using semantic zooming interfaces together with advanced browsing assistance, to support the programmer's navigation tasks based on a program's aspects.

The visualization technique described in Section 3 can be applied to visualize other data-flow VPLs that are similar to that mentioned in Section 2. Examples of such VPLs include Prograph [13], CODE [14], Meander [15], VISTA [16], and V [17].

There have been many attempts to automatically visualize the creation of objects and the messages passed between objects in parallel object-oriented systems. Many studies have also been made of monitoring the performance of these systems 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.

8. Discussion

Although we have concentrated on visualizing runtime networks in this paper, the visualization could be extended to integrate a data browsing facility. This can be achieved by making a stream port expand and the data of the stream be visible within the expanded port when users zoom in on the stream port.

The proposed architecture can be easily extended to provide other kinds of views. One example is a view that highlights processes relevant to the value of a port specified by the user. This can be achieved by adding another module in the view generator, which computes a slice of the program by interprocedural slicing such as in [18, 19], and increases the DOIs of the processes in the computed slice. The user only needs to select

the port of a runtime process to obtain a fish-eyed view that highlights processes in the runtime process network that might affect the value of the port selected.

The proposed architecture can be extended to show other kinds of measurements, such as the memory usage of processes and the amount of data transmitted via stream links. Two approaches to presenting such measurements can be considered: presenting a measurement as a fish-eyed view, or as a graphical diagram instead of a process icon. The former approach can be achieved by assigning measurements as DOIs of processes. The latter can be done by assigning a representing function that shows a diagram of the measurement. Both would work well, but the strategy for selecting an appropriate approach would depend on the properties of the measurements. Further research on this topic is necessary.

9. Summary

This paper presents a scalable visualization technique to animate the execution of declarative data-flow VPLs using the Continuous Zoom algorithm. We show a software architecture to provide a scalable interface for debugging, which exploits a multi-focus fisheye viewing algorithm in conjunction with a semantic zooming interface to show various kinds of information at runtime. The architecture also supports the user's browsing tasks by automatically generating views highlighting certain aspects, based on information embedded in debugged programs.

This framework relieves us of the load both of focusing tasks on the fisheye viewing interface and selection of appropriate modules, thus achieving greater scalability in browsing.

References

1. M. T. Heath & J. A. Etheridge (1991) Visualizing the performance of parallel programs. *IEEE Software* **8,** 29–39.
2. K. M. Kahn & V. A. Saraswat (1990) Complete visualizations of concurrent programs and their executions. In: *Proceedings of IEEE Workshop on Visual Languages*, Skokie. Illinois. IEEE Computer Society Press.
3. W. Citrin & C. Santiago (1996) Incorporating fisheying into a visual programming environment. In: *Proceedings of IEEE Symposium on Visual Languages*, Boulder, Colarado. IEEE Computer Society Press, pp. 20–27.
4. K. M. Fairchild, S. E. Poltrock & G. W. Furnas (1988). SemNet: three-dimensional graphic representation of large knowledge bases. In: (R. Guindon, ed.) *Cognitive Science and its Applications for Human–Computer Interaction*. Lawrence Erlbaum Associates, Hillsdale, NJ, pp. 201–233.
5. G. W. Furnas (1986) Generalized fisheye views. In: *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, Boston, Massachusetts. Association for Computing Machinery, ACM Press. pp. 16–23.
6. J. D. Mackinlay, G. G. Robertson & S. K. Card (1991) The perspective wall: detail and context smoothly integrated. In *Proceedings of ACM CHI'91 Conference on Human Factors in Computing Systems*, New Orleans, Louisiana, ACM Press. pp. 173–179.
7. K. Misue & K. Sugiyama (1991) Multi-viewpoint perspective display methods: formulation and application to compound graphs. In: *Proceedings of the 4th International Conference on Human–Computer Interaction*, Stuttgart, Germany, **2,** pp. 834–838.

8. M. Sarkar, S. S. Snibbe, O. J. Tversky & S. P. Reiss (1993) 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*, Atlanta, Georgia. Association for Computing Machinery, ACM Press, pp. 81–91.
9. L. Bartram, A. Ho, J. Dill & F. Henigman (1995) 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*, Pittsburgh, Pennsylvania. Association for Computing Machinery. ACM Press, pp. 207–215.
10. M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, & E. Shibayama (1997) Supporting design patterns in a visual parallel data-flow programming environment. In: *Proceedings of IEEE Symposium on Visual Languages*, Capri, Italy. pp. 76–83.
11. E. Shibayama, M. Toyoda, B. Shizuki & S. Takahashi (1999) Visual abstractions for object-based parallel computing. In: *Object-Oriented Parallel and Distributed Programming*, Hermes Science Publications, Paris, France, pp. 113–132.
12. J. Tanaka (1994) Visual programming system for parallel logic languages. In: *The NSF/ICOT Workshop on Parallel Logic Programming and its Program Environments.* The University of Oregon, pp. 175–186.
13. P. Cox, F. Giles & T. Pietrzykowski (1989) Prograph: a step towards liberating programming from textual conditioning. In: *Proceedings of IEEE Workshop on Visual Languages*, Rome, Italy, IEEE Computer Society Press, pp 150–156.
14. P. Newton & J. C. Browne (1992) The CODE 2.0 Graphical Parallel Programming Language. In: *Proceedings of ACM International Conference on Supercomputing*, Washington, U.S.A. ACM Press.
15. G. Wirtz (1994) Modularization and process replication in a Visual Parallel Programming Language. In: *Proceedings of IEEE Symposium on Visual Languages*, St. Louis, Missouri, IEEE Computer Society Press, pp. 72–79.
16. S. Schiffer & J. H. Fröhlich (1995) Visual programing and software engineering with Vista. In: *Visual Object-Oriented Programming: Concepts and Environments*. Chapter 10, Manning Publications Co., Greenwich, pp. 199–227.
17. M. Auguston & A. Delgado (1997) Iterative constructs in the visual data flow language. In: *Proceedings of IEEE Symposium on Visual Languages*, Capri, Italy. IEEE Computer Society Press, pp. 152–159.
18. S. Horwitz, T. Reps & D. Binkley (1990) Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems* **12,** 26–60.
19. M. Kamkar, N. Shahmehri & P. Fritzson (1992) Interprocedural dynamic slicing. In: *Programming Language Implementation and Logic Programming* (*PLILP '92*), in Lecture Notes in Computer Science, Vol. 631 Springer-Verlag, Berlin, pp. 370–384.