Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment

Masashi Toyoda Buntarou Shizuki Shin Takahashi Satoshi Matsuoka Etsuya Shibayama
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
2-12-1 Oookayama Meguro-ku Tokyo 152, JAPAN
{toyoda,shizuki,shin,matsu,etsuya}@is.titech.ac.jp

Abstract

We propose the notion of visual design pattern *(VDP), which is a visual abstraction representing design aspects in parallel data-flow programs. VDP serves as a flexible and high-level structure of reuse for visual parallel programming. We introduced the support for this notion into the visual parallel programming environment, KLIEG, allowing definition and use of patterns with simple and easy interface.*

1 Introduction

Design pattern approaches have been recently proposed to describe recurring design aspects in object-oriented systems for enhancing software reusability. A design pattern is a document that describes the combination technique of abstract objects using diagrams, descriptions, and example programs. Catalogs of design patterns such as [3] have been published; with these catalogs, non-expert programmers can use well-designed combination techniques, and can construct reusable software by implementing abstract objects depicted by the patterns.

Design pattern approaches are also important in visual parallel data-flow programming (VPDP). Our goal is to formulate the notion of design patterns which is suitable for VPDP, and to make it easier to define and reuse design patterns, and also to reuse implemented programs.

It is, however, difficult to practice pattern-based design in VPDP, because normal VPDP languages such as CODE [7] do not support the notion of replaceable components. In an object-oriented design pattern, each abstract object represents a replaceable component, and a particular behavior of a program can be determined or modified by replacing components in the pattern. Such essential mechanisms to reuse designs and programs should also be supported by VPDP languages, but most lack systematic means to replace their components.

In addition, design information is important not only for documents, but also for programming environments. For object-oriented languages, there are already tools that support automatic code generation from design patterns, such as [2]. However, because they do not maintain design information in the generated program itself, it is difficult to learn the intention of the program design, such as which components can be modified to change a particular behavior.

Under these observations, we introduce reusable program structures based on data-flow diagrams, in which the designer can define replaceable components, and add design information directly to the structures. We call these structures *visual design patterns* (VDP). We directly support interactive definition, reuse, and even execution of VDPs in a visual parallel data-flow programming environment.

Here are some important design issues for a VDP system:

Management of multiple sets of processes A well-designed VDP will facilitate processes which can be implemented in several ways. However, changes to the processes must often be coordinated, i.e., a set of processes must be used at once. Therefore it is necessary for the VDP system to allow management and manipulation of a set of processes, so that the designer can allow the user to select from an abstract set of process implementations such as *default*, *sample*, and *alternatives.*

Focusing support for processes editing Since most VDPs are comprised of a numerous number of nested processes, it is important to assist the user on which parts of the given VDP he should edit on adding/modifying some new functionality. Thus, a VDP system should facilitate a feature to focus the users editing actions on the particular part of VDP subject to editing.

Visualizing execution of VDPs In order for the user to capture the behavior of a VDP, it is important for the execution of each instantiated VDP to be visualized and animated. This will manifest to the user design knowledge that dynamic and otherwise difficult to document statically.

Support for consistency checking Consistency checks on process instantiation and replacement in VDP are desirable to improve usability. For example, a user may instantiate a pattern with a wrong set of processes; by incorporating consistency checking mechanism into a VDP system, such errors can be checked, or the operation is invalidated in the first place. Furthermore, such consistency information can be employed to assist in the editing, when there are ambiguities on which inputs connect to which outputs, etc.

We have implemented a VDP system based on a visual parallel programming environment KLIEG, which answers the above issues. KLIEG itself is a visual parallel data-flow language based on a parallel logic programming language, moded FGHC [10]. We refer to a VDP in KLIEG as a KLIEG-VDP. Designers can define KLIEG-VDPs that retain design information as patterns, and then users choose a pattern from a catalog of KLIEG-VDPs in their programs and implement customized processes in the pattern. The features of the KLIEG-VDP are as follows:

- In KLIEG, a KLIEG-VDP is represented as a data-flow diagram that has some replaceable, non-instantiated processes called *holes*. Users can reuse the topology of the KLIEG-VDP by instantiating the holes with customized processes which are appropriate for the KLIEG-VDP.

- A hole is allowed to hold multiple processes, one of which is valid at a given time. This allows the designer to provide a default implementation, several alternatives, and sample code for a hole in the KLIEG-VDP. The user can select an appropriate implementation of the hole from these alternatives.

- For highlighting replaceable processes, KLIEG-VDPs support multi-focus fisheye-like viewing. The designer magnifies the processes which should be replaced together for changing a particular behavior, and saves the layout with an appropriate name. The user can easily determine the replaceable processes by selecting the layout.

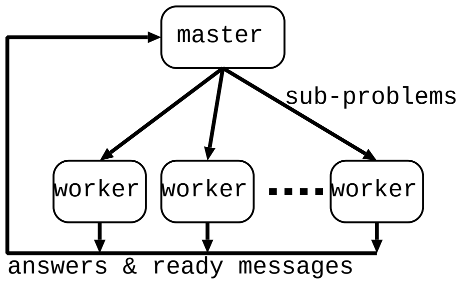

Figure 1: Master worker pattern

- To show the dynamic behaviors of processes, KLIEG provides an execution *tracer*, which visualizes and animates the execution of the program. The user can see the behaviors by executing the sample code.

- KLIEG-VDP system facilitates type checking and inference algorithm on communication ports for consistency checking. By checking types, KLIEG presents only the appropriate processes for a hole from multiple implementation choices. KLIEG also links replaced processes as automatically as possible.

2 Visual Design Patterns on KLIEG

In this section, we describe the details of KLIEG-VDPs. A KLIEG-VDP is a data-flow network diagram that has holes as parameters and that maintains the design information described in the previous section. Data-flow network diagrams are components of KLIEG programs, and consist of *processes* with input or output *ports* and *links* that connect input ports and output ports. A hole also has ports, and can be instantiated with processes that have at least the ports of the same type. A network diagram may be constructed hierarchically from multiple networks.

In the following, we show an example of KLIEG-VDP, and then describe how the design information is maintained in KLIEG-VDPs, and how to define and reuse them.

2.1 An Example

We show the *master-worker* pattern, which provides a simple load balancing scheme that involves a master process and a collection of worker processes. Figure 1 illustrates the concept of the master-worker pattern. The master partitions a problem into sub-problems and sends them to workers ready to compute. The workers are responsible for computing sub-problems. A worker returns the solution(s) of a sub-

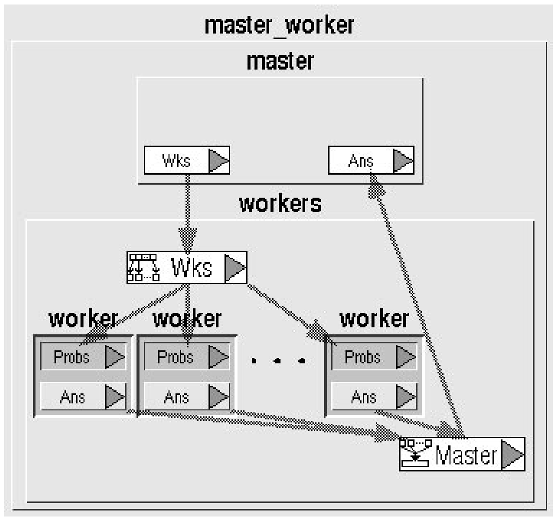

Figure 2: Master-worker pattern in KLIEG

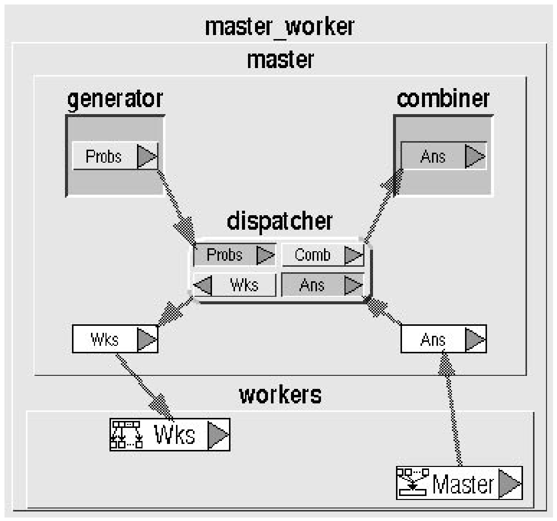

Figure 3: Detail of master network

problem to the master. When a worker completes the computation of a sub-problem, it notifies the master that it is ready to compute again. By providing appropriate masters and workers, we can use this pattern for solving various parallel programming problems, such as ray-tracing and search problems.

Figure 2 depicts the KLIEG-VDP that represents the master-worker pattern. The master_worker pattern is a network constructed from two networks, master and workers, that correspond to the master process and the collection of worker processes, respectively. Both master and workers have some holes that represent processes, which depend on the problem to solve[1].

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 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 sub-problems from the Probs port, solves them, and returns answers to the master via the Ans port.

[1] Master is shrunken, so the details are hidden.

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-right of Figure 2) is a merge port that merges the output streams of all the processes. Besides these ports, we can use *broadcast ports* that broadcast received elements to all the processes.

In Figure 3, the structure of the master hidden in Figure 2 is expanded by zooming. The master network is composed of the generator and combiner hole, and the dispatcher process. Generator simply generates a stream of sub-problems. Combiner receives answers from dispatcher and computes the final answer. The dispatcher process is the default implementation of the dispatcher hole. It receives sub-problems from generator and messages from workers (ready and answer). Then it sends the sub-problems to ready workers, and sends the answers to the combiner.

In addition, we should also mention that KLIEG-VDPs can be combined hierarchically for constructing large-scale programs from smaller ones. To construct KLIEG-VDPs hierarchically, we merely instantiate a hole with an entire KLIEG-VDP. In fact, master_worker is constructed in this way, i.e., master_worker has two holes, and these holes are instantiated with the master and workers networks.

In the following sections, we will show the details

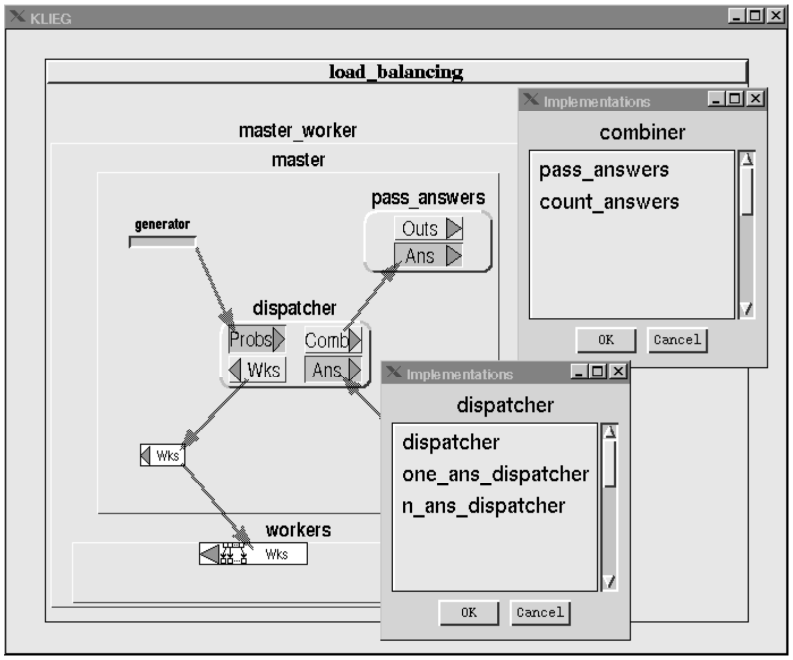

Figure 4: Alternative implementations

of the definition and the use of the master_worker pattern. Definition of network diagrams can be easily performed by normal graph editors, so we will concentrate on issues on definition and use.

2.2 Management of Multiple Implementations

The designer can store a hole with multiple implementations by repeated dragging and dropping icons of appropriate processes onto the hole. Using this interface, the designer can provide a KLIEG-VDP with different kinds of implementations:

Sample implementations By executing these implementations on the tracer, the user can understand the behavior of the KLIEG-VDP.

Default implementations The most likely implementations the user is likely to use.

Alternative implementations The implementations that serve as the basis of user customization.

Implementations in a hole are chosen with a dialog box. Using the dialog box, the user can not only select an implementation, but also change a particular specification of an existing program[2]. We found this is easier than dragging an implementation from the other module every time the user replaces an implementation.

[2]A well-designed VDP should be able to change its specification by replacing implementations in holes.

As an example, we show how to provide multiple implementations for the master-worker pattern to solve search problems. When solving search problems with the master-worker pattern, combiner might be implemented independent of the problem to solve. Thus we provide two implementations to combiner (Figure 4). On the top-right of Figure 4, the Implementations dialog box is shown. The pass_answers process passes through the answers from dispatcher to the output, and the count_answers counts the number of answers.

In addition, we can change the treatment of answers from finding all answers to finding the fixed number of answers. This is done by changing the dispatcher process to the process that terminates all the workers when it receives the fixed number of answers from workers. To perform this, we only add an implementation of a new dispatcher to dispatcher hole. On the bottom-right of Figure 4, a dialog box shows the one_ans_dispatcher process for finding single answer, and the n_ans_dispatcher process for finding n answers.

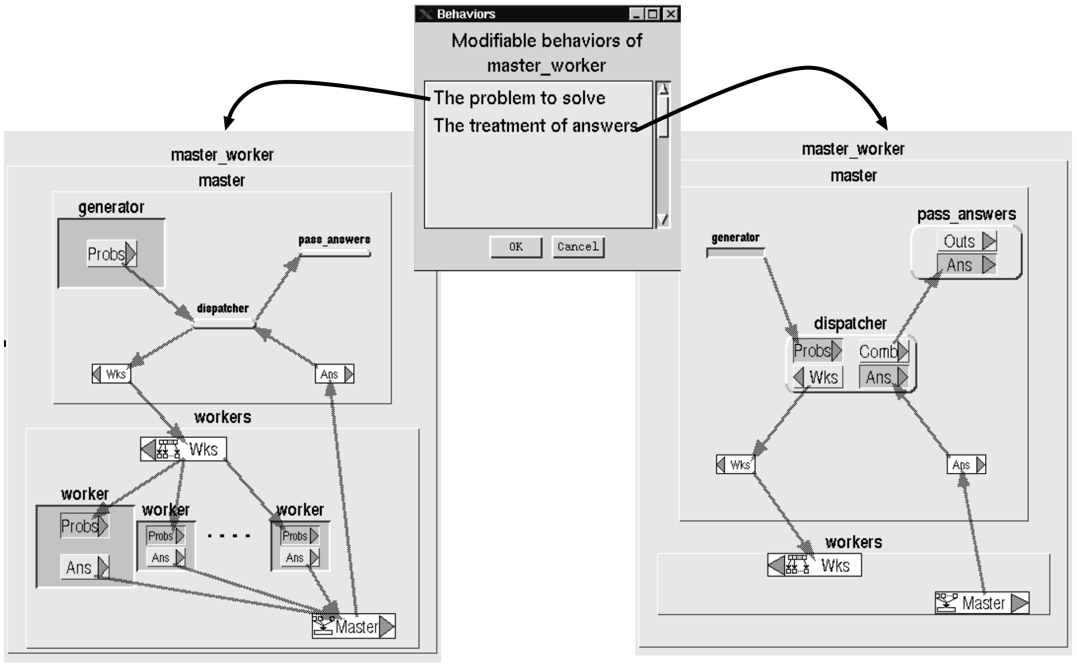

Figure 5: Changing layout for modifying behaviors

2.3 Focusing Support for Processes Editing

The designer can show which processes should be modified to change a particular behavior of a KLIEG-VDP by saving a fisheye-viewed layout of the KLIEG-VDP with an appropriate name. The user can select a layout by the behavior name and can easily find out which processes should be modified.

Using multi-focus fisheye viewing, we can obtain a layout in which all related components are magnified in a single view, even if they are separated from each other on screen. In such a view, we can see the details of the components and the overview of the network at the same time. In addition, the editor of KLIEG animates transition from one layout to another layout, so that the user is not confused even if the layout drastically changes.

When the user intends to change a particular behavior of a program, the user selects an appropriate layout and replaces the magnified processes. The changes to a KLIEG-VDP in one layout view are reflected in other layouts.

We use a zooming algorithm that resembles the continuous zoom algorithm[1], which supports multi-focus fisheye viewing of hierarchically-organized networks such as KLIEG programs. We cannot use the continuous zoom algorithm directly, because it lacks the facility for editing such as avoiding overlapping of graphical objects during layout modification. Our zooming algorithm avoids overlapping automatically, and also supports saving and recovery of layouts.

As an example, we show two layouts of the master-worker pattern for search problems. One is for changing the problem to solve, and the other is for changing the treatment of the answers. Figure 5 depicts two layouts and a dialog box for changing layouts. The layout on the left is for changing the problem. To change the problem, the user must change the generator and the worker magnified in the editor. Another layout on the right is for changing the treatment of the answers. Similarly, the treatment of the answers can be changed by modifying combiner or dispatcher magnified in the editor.

2.4 Visualizing Execution of KLIEG-VDPs

The user can observe the behaviors of KLIEG-VDPs by executing a sample program with the tracer. As an example, we show the sample program that solves the N-Queens problem using the master_worker pattern, and its execution on the tracer. The program in Figure 6 was constructed by selecting the layout for changing the problem (Figure 5), and instantiating the

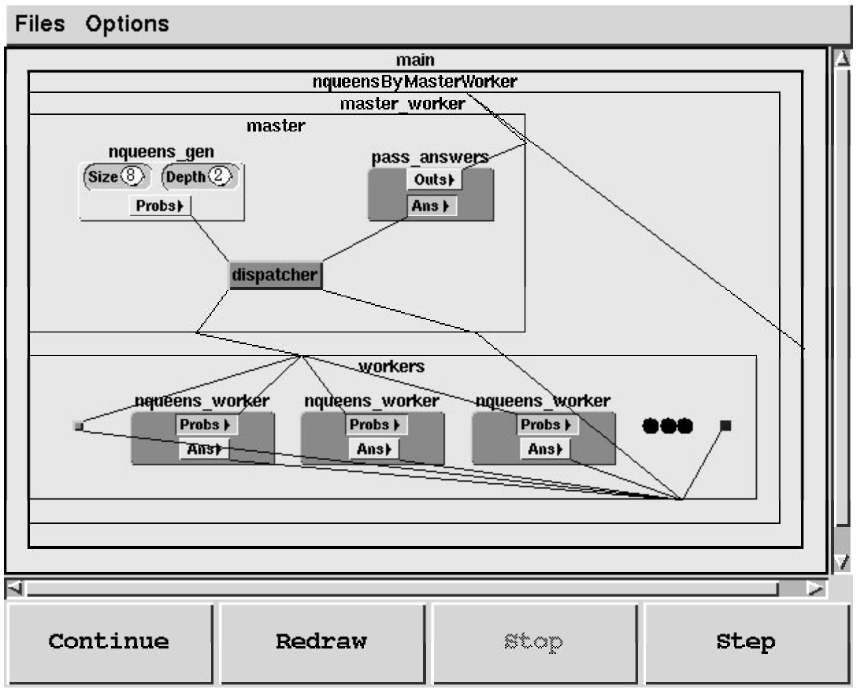

Figure 7: Executing the N-Queens program on the tracer

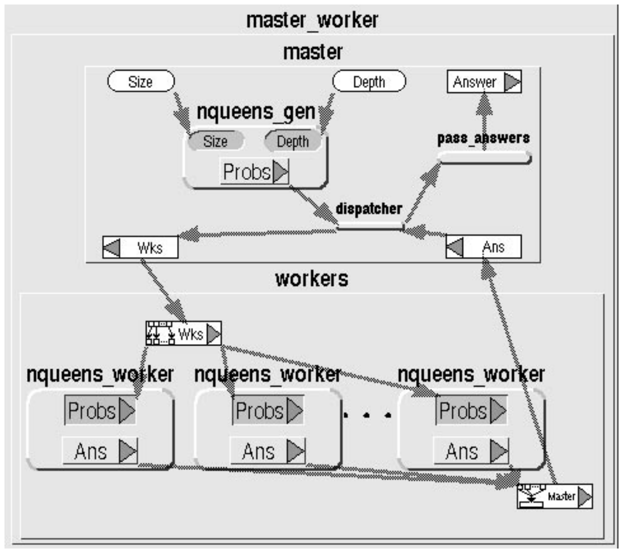

Figure 6: Solving N-Queens problem using master_worker

holes with processes for the N-Queens problem. In the master network, the ports Size and Depth[3] are added for inputting the search parameter, and the Answer port is added to output answers.

Figure 7 shows a snapshot of the execution of the N-Queens program in the tracer. The tracer animates the transition of the network during the execution, while maintaining the topology of the pattern in the program. This is possible because the design information of the KLIEG-VDP can be referred from the runtime system of KLIEG. The tracer can also show the contents of streams. Thus the user can easily recognize the KLIEG-VDPs used in the program, and can observe the behaviors from the animation and the contents of streams. The tracer also supports fisheye viewing, so the user can navigate through large-scale networks using automated zooming of important parts of the program.

2.5 Checking Consistency Using Types

To further improve the user interface of KLIEG-VDPs, we employ type checking and inference of ports of processes. Using type information, KLIEG can restrict the possible candidates for a hole, and can connect ports of processes automatically.

[3]They are singleton ports that receive only one datum

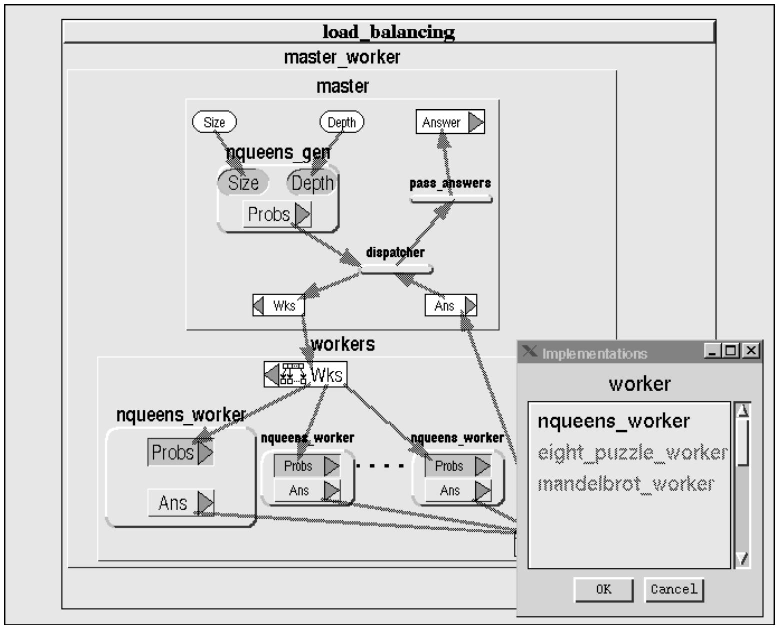

Figure 8: Checking port types of the worker

As an example, Figure 8 shows the situation where the generator hole has been instantiated with nqueens_gen process that generates sub-problems for the N-Queens problem. In this case, processes that have no relevance to the N-Queens problem should be disabled when the user instantiates the worker hole or selects the worker process. To detect this, the port type of Probs of nqueens_gen is propagated to dispatcher and then to the worker holes by the type inference algorithm, so that processes except for nqueens_worker are disabled in the Implementations dialog box (at the bottom-right of Figure 8).

As another example, consider the case the user replaces dispatcher with another implementation. It would be impossible to connect ports automatically without type information, because dispatcher has two input ports and two output ports, and can be linked in different ways. In this case, KLIEG automatically connects the ports correctly if the types of all ports are detected. In this way, the user does not have to reconnect each port every time when replacing the processes.

To implement these type checkings, we use a constraint based type analysis. The base language of KLIEG is moded FGHC [10], and our analysis technique is also based on the mode-analysis algorithm of moded FGHC. We omit the details of the algorithm for brevity; interested readers are referred to [10].

3 Programming with KLIEG-VDPs

For users, programming using KLIEG-VDPs is simple and easy. It can be performed using the same interfaces for definition, namely dragging and dropping interface and dialog boxes:

1. Select a necessary KLEG-VDP from a library, and drag and drop it on to one's program.
2. If a sample code exists, execute the code with the tracer, and confirm the behaviors of the KLIEG-VDP.
3. Select a layout corresponding to the behavior to be modified from the dialog box.
4. Leave the default implementation for a hole as is, if it is sufficient. If not, select an appropriate implementation for each hole from its dialog box. If there are no appropriate implementations, implement necessary processes for unspecified holes, and instantiate the holes with the processes using drag and drop.
5. Add necessary ports and links that are not defined in the KLIEG-VDP.
6. When changing the other behaviors of the program, repeat the steps 3 to 5.
7. Execute the program and for change in behavior go back to step 3.

4 Related Work

There are many visual parallel data-flow programming environments such as CODE[7], Meander[11], and SAA[6]. There are also environments based on parallel logic programming languages like Pictorial Janus[4], and PP[9]. Because these environments lack the support for VDPs, it is difficult to replace a set of components in data-flow diagrams. The user must manually delete a set of processes, create new processes, and link processes appropriately. This is a tedious and time-consuming task. In KLIEG, holes can be easily instantiated with dragging and dropping with automated support for linking ports, etc.

In VISTA[8], *processors* may have an internal network of processors. In particular, internal processor called *public processor* can be replaced by another processor with a compatible interface. Although similar to VDPs, public processors in VISTA are merely replaceable, thus the user must search appropriate processors from libraries. In KLIEG, a hole can have alternative processes, so the search for processes is unnecessary. In addition, since VISTA shows only the list of public processors, it is difficult to know which processors need to be changed when multiple processors should be changed to obtain desired behavior.

Holon/VP[5] uses an object sharing technique to enhance reusability of program. This technique allows multiple networks sharing the same process as their components. This is usable for customizing existing programs by adding functions, but difficult for customizing by *replacing* the functions, which is easy with KLIEG. Furthermore, we can add functions to a process by adding another process that implements the function and intercepts input ports of the process.

5 Conclusion

We have proposed the notion of visual design patterns, which is flexible and high-level structure of reuse, and introduce this notion into the visual parallel programming environment KLIEG. A VDP maintains design information through out all programming stages. Checking consistency improves usability and reusability of components in visual parallel data-flow language. In addition, designers and users can perform each task using the same simple user interfaces.

In future work, we plan to support interaction diagrams and protocol checking on VDPs, to represent more information of designs.

References

[1] Lyn Bartram, Albert Ho, John Dill, and Frank Henigman. The Continuous Zoom: A Constrained Fisheye Technique for Viewing and Navigating Large Information Space. In *Proceedings of UIST '95*, pp. 207–215, November 1995.

[2] F. J. Budinsky, M. A. Finnie, J. M. Vlissides, and P. S. Yu. Automatic code generation from design patterns. *IBM Systems Journal*, Vol. 35, No. 2, 1996. – Object technology.

[3] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[4] Vijay A. Saraswat Kenneth M. Kahn. Complete Visualizations of Concurrent Programs and their Executions. In *Proc. 1990 IEEE Workshop on Visual Languages*, October 1990.

[5] Yuichi Koike, Yasuyuki Maeda, and Yoshiyuki Koseki. Enhancing Iconic Program Reusability with Object Sharing. In *Proc. 1996 IEEE Symposium on Visual Languages*, pp. 288–295, 1996.

[6] Keng Ng, Jeff Kramer, Jeff Magee, and Naranker Dulay. A Visual Approach to Distributed Programming. In A. Zaky and T. Lewis, editors, *Tools and Environments for Parallel and Distributed Systems*, chapter 1. Kluwer Academic Publishers, February 1996.

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

[8] Stefan Schiffer and Joachim Hans Fröhlich. Visual Programing and Software Engineering with Vista. In *Visual Object-Oriented Programming: Concepts and Environments*, chapter 10, pp. 199–227. Manning Publications Co., 1995.

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

[10] Kazunori Ueda and Morita Masao. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, Vol. 13, No. 1, pp. 3–43, 1994.

[11] Guido Wirtz. Modularization and Process Replication in a Visual Parallel Programming Language. In *Proc. 1994 IEEE Symposium on Visual Languages*, pp. 72–79, 1994.