Visual Abstractions for Object-Based Parallel Computing

Etsuya Shibayama, Masashi Toyoda, Buntarou Shizuki, Shin Takahashi
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
2-12-1 Oookayama, Meguro-ku, Tokyo, 152-8552, JAPAN
{etsuya, toyoda, shizuki, shin}@is.titech.ac.jp

Abstract

We propose the notion of visual patterns, which describe various aspects of object-based parallel and distributed computing, and have developed a visual supporting environment for parallel programming based upon visual patterns. In this paper, we show the power of visual patterns in design, programming, and debugging processes.

1 Introduction

Object-oriented computing models inherently have visual natures: the essence of object-orientation is to model anything and any behaviors in terms of a collection of inter-related objects and interactions among them; such a collection naturally forms a general graph structure that a diagrammatic representation is best fitted for human designers and programmers to comprehend. In object-oriented software engineering community, for instance, recent experiences in object-oriented analysis and design or OOA/OOD (e.g., [RBP+91, Boo94]) proves the usefulness of various sorts of visual diagrams (e.g., class and object diagrams) that represent design of object-oriented software. Also in the object-oriented programming community, JavaBeans and some other visual programming environments are now coming to maturity. Concurrent, parallel, or distributed object-based programming could not be the exceptions and neat visual representations of parallel and distributed object-based programs/computations are desirable.

Just being visual is not sufficient, of course. In this paper, we propose new visual abstractions, *visual patterns*, that describe various aspects of parallel and distributed object-based design, programming, and debugging in a coherent manner. We also introduce a prototype parallel visual programming environment KLIEG[TST+97b] that provides a support for:

- an *object-based visual parallel programming language KLIEG*;
- visual patterns that keep information of software design and object layout on the screen.

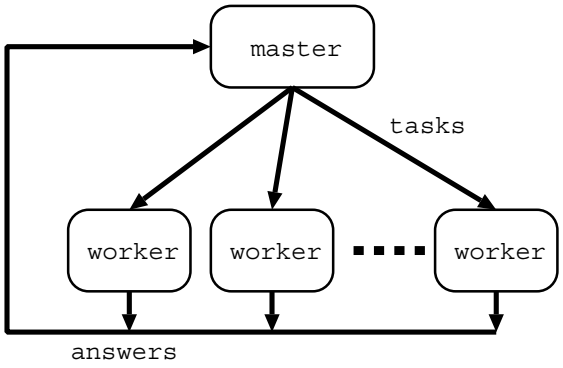

Figure 1: A diagrammatic representation of concurrent objects

The major design issues of the KLIEG environment include the following:

- visual representations vs. textual representations;
- design, programming, and debugging in the large on a relatively small screen;
- a seamless integration of design, programming, and debugging processes.

In the sequel, in Section 2 we discuss the benefits of visual representations in object-based parallel programming. Also in this section, we introduce the KLIEG language and propose the new programming methodology, that is, *pattern-oriented visual programming*. In Section 3 we briefly review a support for *visual design patterns* provided by the KLIEG programming environment. In Section 4, we discuss scaling-up issues and introduce our approach based upon distorted multi-focus zooming techniques. We compare our work and related works in Section 5 and finally summaries the current status and the future direction of our work in Section 6.

2 The Visual Language KLIEG and Pattern-Oriented Visual Programming

2.1 Why Being Visual?

As was briefly mentioned in the previous chapter, diagrammatic representations of object-oriented programs or designs are better fitted for human designers and programmers than the corresponding textual representations. This reason is simple and obvious: any forms of textual representations of general graphs or networks invented so far are not as comprehensible as standard pictorial representations.

In Figure 1, for instance, a typical diagrammatic representation of a collection of inter-related objects is illustrated. This figure represents a *master-workers* object network, in which a single master object dispatches tasks to multiple worker objects and gathers the results of the workers' computations. Either procedural or declarative, a textual representation of this sort of network is rather indirect and harder to understand. Notice that visual approaches in object-orientation are completely different from those approaches

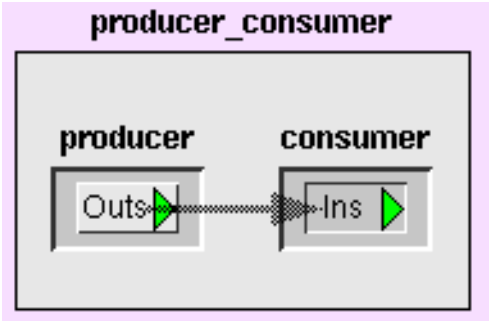

Figure 2: A producer-consumer pattern

based on (structured) flow-charts: with a little computer support for syntax-directed editing and outline processing, syntax trees in a textual form (i.e., ordinary programs) can be as comprehensible as those in a visual form.

2.2 Patterns in KLIEG

Based upon the observation in the previous subsection, we design a visual (i.e., pictorial) language KLIEG for object-based parallel computing. Programs in the language KLIEG are depicted as visual *data-flow diagrams* and, in this respect, KLIEG is similar to CODE[PJ92] and Pictorial Janus[KMK90]. One of the significant differences is that KLIEG provides a support for visual patterns and pattern-oriented visual programming.

2.2.1 Basic Usage

A visual pattern in KLIEG is represented as an object data-flow diagram with abstract objects, which are called *holes* and to be instantiated later with concrete objects. In the KLIEG environment, a visual pattern can keep design and layout information. The detail of this issue will be described in Section 3.

Figure 2 is the first and simple example of visual pattern, which represents the skeletal structure of *producer-consumer* object network. This pattern has two holes, that is, producer and consumer. In KLIEG, a recessed rectangle like producer or consumer represents a hole, which is to be instantiated with a concrete object. In this figure, the producer hole has an output port Outs and the consumer has an input port Ins. In general, an input port is depicted recessed and an output port raised. These two ports are called output and input *stream ports*, meaning that they transmit and accept, respectively, streams of messages. The arrow connecting these two ports represents a communication channel or message stream.

The KLIEG environment provides a support for definitions and use of visual patterns. For definitions of visual patterns, an editing interface similar to an draw editor is available. More advanced editing features including zooming supports will be introduced in Sections 3 and 4. For use of visual patterns, a drag-and-drop interface is provided. Holes of a pattern are instantiated with objects by dropping the icons representing the objects.

In Figure 3, the holes of producer are instantiated with two objects naturals and sum by dropping their icons onto the holes (Exactly speaking, each hole is instantiated with

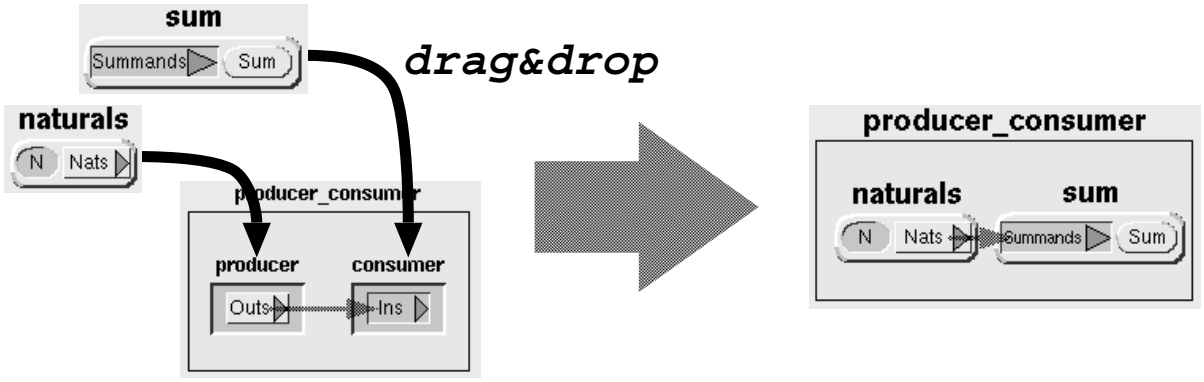

Figure 3: Hole Instantiations by drag-and-dropping objects

a copy of the object). In KLIEG, a raised round rectangle like naturals or sum is an iconic representation of an object, which depicts its signature or interface, i.e., the names and sorts of its ports. The intended behavior of this network is that, upon reception of the value N, naturals transmits $1, 2, \ldots, N$ to sum, which in turn computes the value of $1 + 2 + \cdots + N$ and finally puts it on the port Sum.

In this case, naturals has two ports N and Nats. N is an input port that accepts just a single message in its lifetime. This sort of port is called *input singleton port* and depicted as a recessed round rectangle. Similarly, Sum is an output port that transmits just a single message. This sort of port is called *output singleton port* and depicted as a raised round rectangle.

As illustrated in Figure 3, the output stream port Nats of naturals and the input stream port Summands of sum are automatically connected via a message stream upon instantiation. This connection is made by matching Nats and Summands with Outs and Ins, respectively. In contrast, N and Sum cannot match any ports in the pattern. In general, the dropped object can have more ports than the hole to which it is dropped.

The KLIEG environment calculates the most probable matching among ports of the object and the hole using the following information of each port:

- whether singleton or stream;
- whether input or output;
- the types of messages received/transmitted on the port;
- the geometry within a hole or an object.

The first two are obvious: a singleton port only matches another singleton port, and so on. The third information is exploited by the type inference algorithm that is similar to the mode analysis algorithm[UM94] for an parallel logic programming language FGHC (Flat Guarded Horn Clauses). The last information is conducted only when the other three are not sufficient for resolution of ambiguities since it is heuristic information and thus error-prone.

In case of Figure 3, the first two information is sufficient to get the correct result, i.e., Outs of producer and Ins of consumer correspond to Nats of naturals and Summands of sum, respectively. Notice that the name of a port is ignored in this matching process.

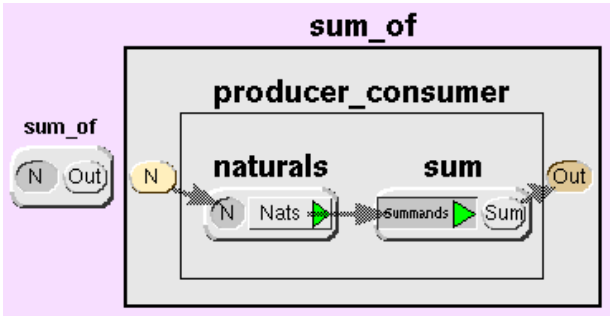

Figure 4: The interface and implementation of sum_of

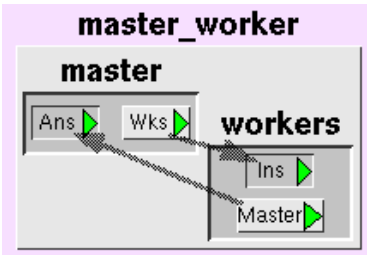

Figure 5: A base pattern for master-workers

2.2.2 Hierarchical Constructions

Even with a pictorial representation, a large and flat object network is rarely comprehensible. To overcome this difficulty, the KLIEG language/environment provides a means for hierarchical constructions of patterns and object networks.

Firstly, an object in KLIEG can be defined hierarchically. Figure 4 illustrates a simple example: the small round rectangle sum_of represents the interface of an object sum_of; the rest of the figure represents its body or implementation. That is, an object sum_of consists of two objects naturals and sum that are embedded in the producer_consumer pattern. Notice that this picture is regarded as a rewriting rule and so the KLIEG language processor reduces the interface of an object in a program into its implementation. Notice also that the KLIEG allows the programmer to describe visual conditional rewriting rules that are as expressive as clauses of a committed-choice parallel logic language Moded FGHC[UM94].

Secondly, a visual pattern can be defined in a hierarchical manner. That is, by dropping a pattern onto a hole of another pattern, the hole can be instantiated with the former pattern. For instance, a master-workers network illustrated in Figure 1 can hierarchically be defined in KLIEG as follows:

1. **Defining the fundamental structure of the master-workers pattern**
   Figure 5 depicts the basic structure of the master_worker pattern, which has two holes, i.e., master and workers, and two arrows representing the communication channels between master and workers.

2. **Defining the master and workers patterns**

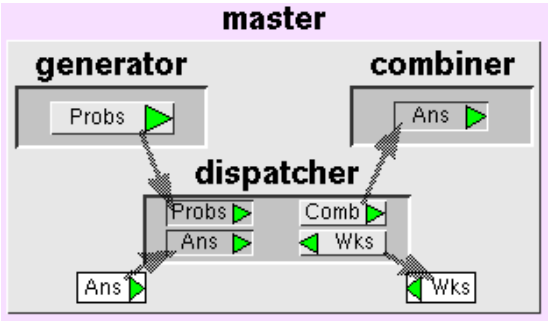

Figure 6: A master pattern

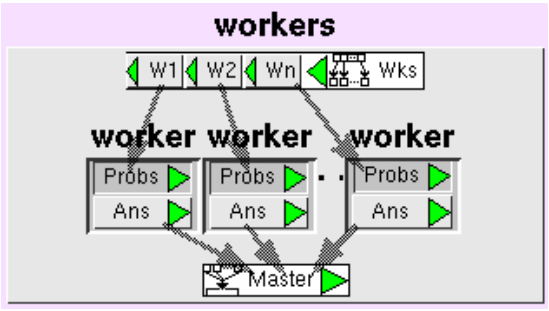

Figure 7: A workers pattern

The master and workers parts in Figure 5 should have their own micro-architectures that are best described in terms of visual patterns. Figures 6 and 7 illustrate master and workers patterns in KLIEG, respectively. Since the master-workers pattern is a canonical example throughout this paper, we resume explanations of Figures 6 and 7, later.

3. **Dropping patterns onto holes**
   By dropping master and workers patterns in Figures 6 and 7 onto the corresponding holes of the master_worker pattern in Figure 5, we get the master_worker pattern in Figure 8.

The master pattern in Figure 6 has three holes generator, dispatcher, and combiner. Generator plays a role of generating tasks that will be delivered to workers. This part heavily depends on the problems to be solved and should often be replaced. Dispatcher receives tasks from generator and deliver each of them to an appropriate worker. Also it receives the computing results of workers and send them to combiner. By replacing dispatcher and combiner, the load balancing policy and the way to combine the partial results, respectively, can be changed. The workers pattern is defined as a *replication network*, that is, workers includes a sequence of holes that are instantiated with copies of the same object. Therefore, once a single worker hole is instantiated with an object, the other worker holes are automatically instantiated with its copies. The ellipsis "$\cdots$" in the sequence of worker holes means that the length of the sequence is not yet determined. It is determined in run-time by the number of messages received at the special port Wks, which is called a

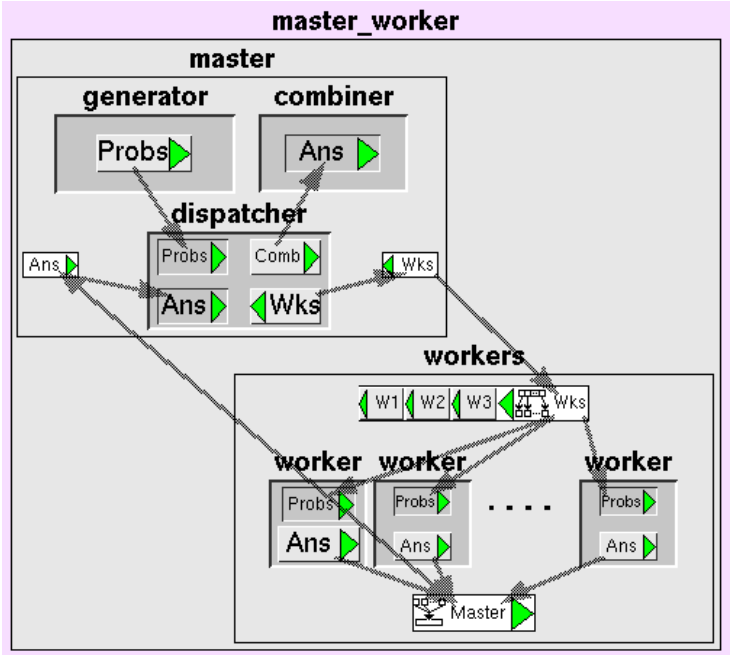

Figure 8: A master-workers pattern

map port.

Similar to Figure 3, a hierarchically constructed pattern can be used by dropping objects onto its holes. Figure 9 is an object network that is constructed of the master-workers pattern and that computes the answers of the N-queens problem.

2.3 Pattern-Oriented Visual Programming

The KLIEG environment introduces a new programming methodology, that is, pattern-oriented visual programming, which is carried out as follows:

1. Designers search for visual patterns in the pattern libraries. If any patterns appropriate for the application cannot be found, this step would be skipped.

2. They construct new patterns that are suitable for describing the software architecture of the application from existing patterns or from scratch in a hierarchical manner.

3. Programmers define and/or search for objects with which holes of the patterns shall be instantiated.

In other words, the software architecture of the application is first defined and components of the architecture will be introduced later as replaceable elements in pattern-oriented visual programming. Also the architecture can be incrementally modified by replacing component patterns (e.g., master and workers patterns in Section 2.2.2). Notice that the designers in the first two stages are expected more experienced in design and programming than those programmers in the last stage.

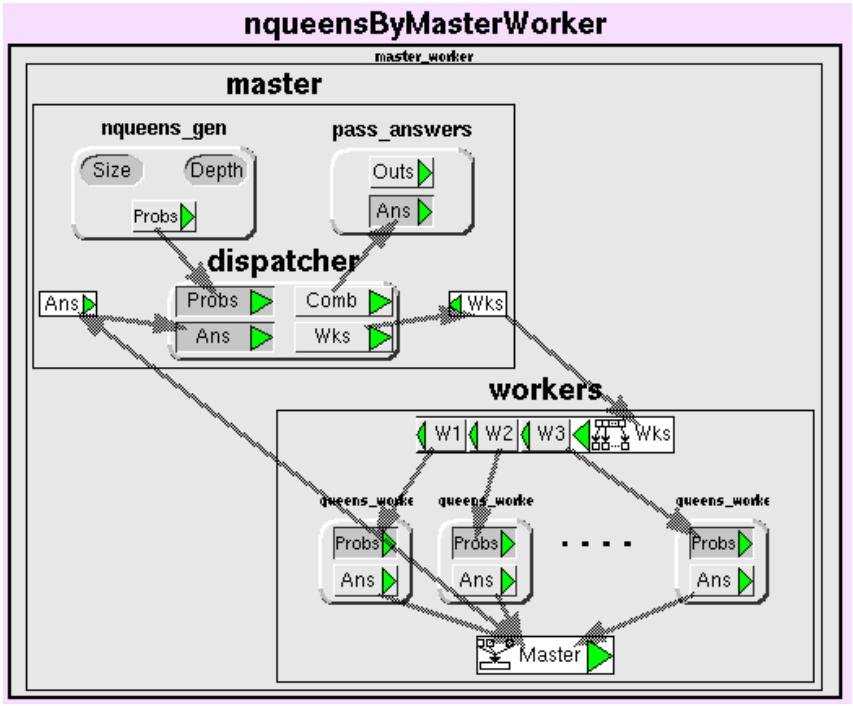

Figure 9: An N-queens program

3 Visual Design Patterns

Recently design patterns[GHJV95, Pre94] have been considered essential in design of flexible and reusable object-oriented software. The notion of design pattern is also important in object-based parallel and distributed programming.

The KLIEG environment provides a support for *visual design patterns*. In this section, we describe issues on supporting design patterns in object-based visual parallel programming environments.

3.1 Design Patterns in an Instance-Based Programming Environment

In our understanding, the significance of design patterns is its supports for flexible and reusable object-oriented software, that is, software that satisfies the following conditions:

- the software is constructed as a collection of objects;
- in order to change some aspects or behaviors of the software, it is sufficient to replace a small number of objects with those of the same roles.

Notice that our understanding is rather *instance-oriented* or puts more emphasis on *run-time structures* of software. Some people might prefer more *class-oriented* views or to pay more attentions on the *program structures*. Often in practice, both instance-oriented and class-oriented views are used in software development processes:

- in the modeling stage, first the application domain is modeled as a collection of inter-related objects;
- in the design and coding stages, the program is created as a collection of class descriptions;
- in the debugging stage, the debugger is used to capture ill-behaved objects.

One of our goals is to design a seamlessly integrated visual programming environment in which all the stages above shall share the same notions and same visual abstractions.

In general, concreteness, directness, and explicitness are important features of visual languages and thus instance-oriented approaches are more promising. In this paper, we show that our instance-oriented approach can be reasonable. For the purpose, we first reconsider the notion of design patterns from the instance-oriented point of view. In this respect, essentially what design patterns provide are:

- coding techniques to make some objects easily replaceable so as to cope with changes of specifications;
- design information including descriptions of design spaces and design decisions.

The coding techniques provided by design patterns could be replaced with language mechanisms and/or environment supports, though they might be necessary for C++ programmers. The hole mechanism of KLIEG is sufficiently expressive and it can make objects and patterns replaceable (i.e., to make software flexible). No more coding techniques are necessary.

The real issue in this section is to provide a support for design information by programming languages/environments. Design patterns are merely documents and therefore design information is rarely available in run-time systems or programming environments. Furthermore, a significant number of people consider that programming environment supports for design patterns are almost useless [1].

Given a specification change, the design information that we consider significant are those about:

- which objects shall be changed or replaced?
- what are their alternatives?
- how they behave?

The first piece of information is obviously important. If reasonable alternatives are already available, the second information is useful. Otherwise, the programmer(s) should learn the roles of the objects to be replaced and implement new alternatives. In this worst case, the last information is useful.

In our approach, these pieces of information are respectively supported by the following manner:

[1] For instance, J. Vlissides listed ten misconceptions of patterns in [Vli97] and the fourth one was "Patterns need tool or methodological support to be effective."

- a visual pattern can have *multiple aspects*, each of which has its own layout information; by selecting an appropriate aspect, the objects to be replaced are displayed with emphasis.
- a hole in a visual pattern can keep more than one object, i.e., *multiple implementations*; each object in a hole may be regarded as default, sample, or alternative implementation;
- the *KLIEG tracer* visualizes a computation using the layout information of a visual pattern provided by its creator using the KLIEG editor.

3.2 A Support for Multiple Aspects

A hierarchically constructed pattern can become large and may have more than one aspect or behavior to be changed. For instance, it is desirable for the master_worker pattern in Figure 8 to have the following aspects:

- the problem to be solved;
- the load balancing policy;
- the way to combine the computed results by the workers.

These three aspects are almost orthogonal, though in practice they can be inter-related.

The KLIEG editor provides a multi-focus distorted zooming interface, called Mochi Sheet[TST+97a], similar to the continuous zoom[BHDH95] in order effectively to display each aspect of a pattern. Figure 10 illustrates two aspects of the master-workers pattern. In the left diagram, holes related to the problem to be solved, i.e., generator and a worker, are magnified and other holes are shrunken. In this manner, holes and objects that should be instantiated and replaced are visually emphasized and so design information concerning "*which objects shall be changed?*" are effectively provided. In the right diagram, the objects related to "*the way to combine computed results by the workers*" are emphasized.

The zooming interface is tightly embedded into the KLIEG editor. On one hand, the designer of a visual pattern can freely change the size and position of any visuals in the pattern and register any layout as a new aspect. On the other hand, a user of the pattern can choose any registered aspect with a dialog box. A change of the aspect is smoothly animated like *morphing*.

3.3 A Support for Multiple Implementations

In KLIEG, more than one object can be dropped onto a single hole of a visual pattern, or the hole can keep more than one object at a time. This mechanism is useful for the designer of a visual pattern to provide several kinds of implementations including:

- the *default implementation* that the user most likely to use;
- *sample implementations* that tell the user the role of the hole;
- *alternative implementations* that the user can choose and customize for building applications.

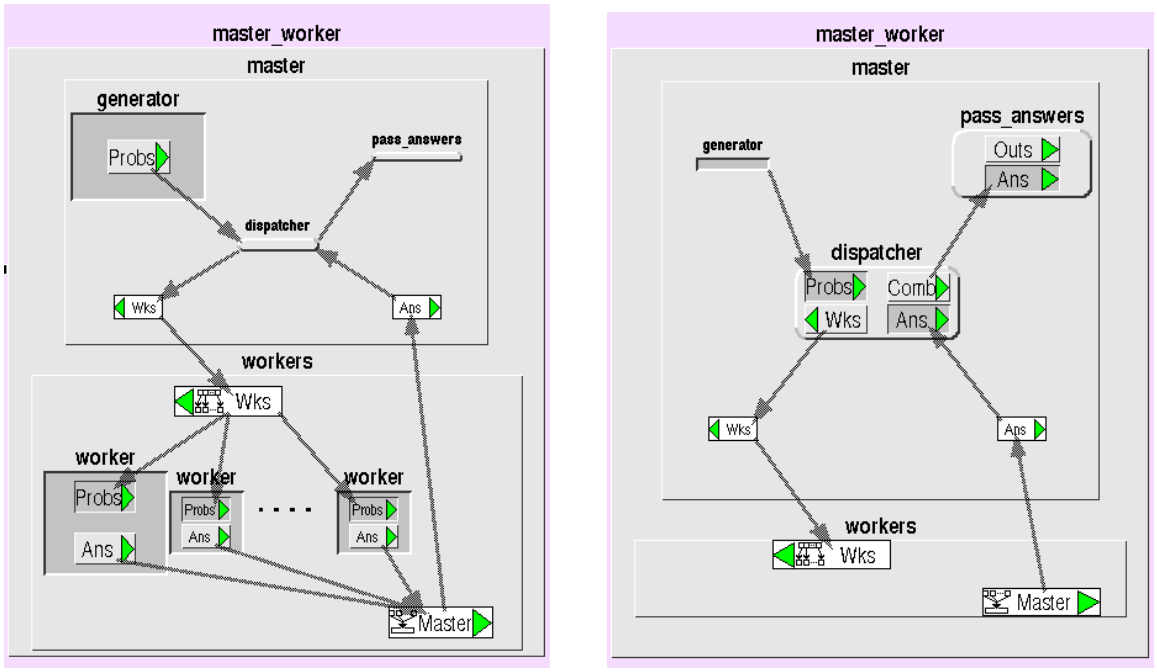

Figure 10: Two aspects of the master-workers pattern

The user of the pattern, on the other hand, can choose an appropriate implementation of a hole via a dialog box. If the default implementation is general enough, what a novice user normally does is just to choose it. With a sample implementation, the user can learn the basic role of the hole, possibly with a help of the KLIEG tracer that visualizes the behavior of the implementation. If a number of alternative implementations cover most areas of the design space, it is sufficient to choose the most eligible one.

Notice that the KLIEG environment has not yet succeeded to effectively provide trade-off information among those alternatives in a visual manner. Currently *written documents* are the only solution to this problem[2].

3.4 Visualizing Program Behaviors

The KLIEG tracer visualizes and animates program execution. Figure 11 is a snapshot of the KLIEG tracer, which are currently executing the N-queens program defined in Figure 9. This picture automatically generated by the tracer is more or less similar to Figure 9: the relative positions and sizes of pass_answers, dispatcher, and a number of nqueen_worker are almost the same. Notice that in this figure the generator has already finished its work and becomes a small rectangle on the top left of the master rectangle.

The KLIEG tracer utilizes the layout information of visual patterns during visualization. In case of Figure 11, for instance, this picture is generated with the layout information of master_worker pattern provided by its designer and without any optional information. This visualization technique using layout information of the visual program is similar to the one employed by Pictorial Janus[KMK90]. In addition, the KLIEG tracer

[2] We have a plan to extend Mochi Sheet to support hyper links.

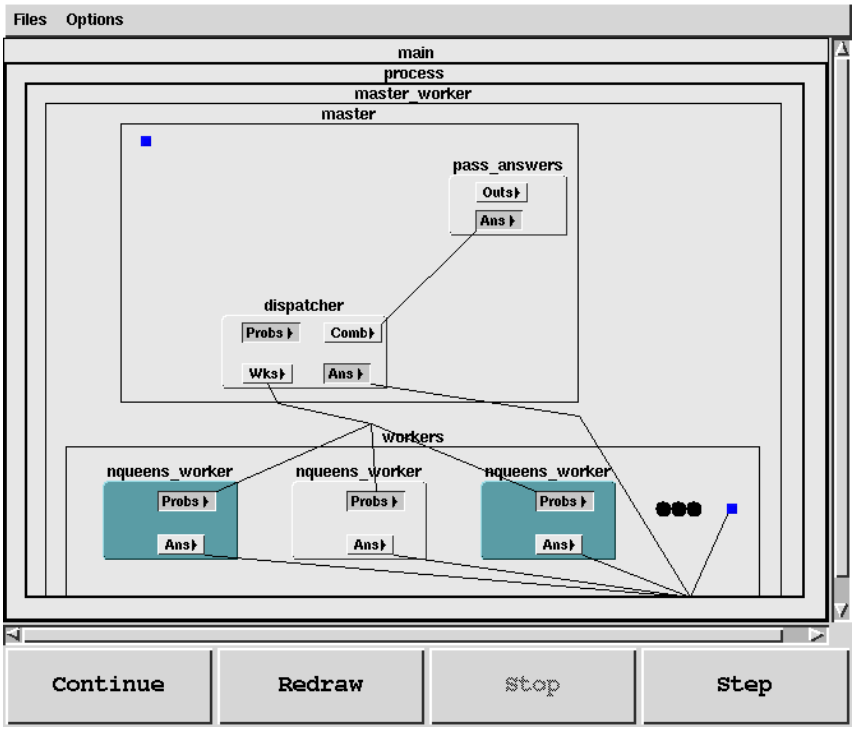

Figure 11: The KLIEG tracer

has a unique feature: it is integrated with the KLIEG editor. The tracer also supports multiple aspects of a visual pattern and the user can interactively change its aspect during execution. For the purposes, the tracer also uses a multi-focus distorted zooming algorithm.

4 Scaling-up Issues

For a long time, visual languages have been considered only useful for toy problems or end users' programming. However, recent advancements of visual technologies can make visual languages more practical. The KLIEG environment provides zooming interfaces for the user to manipulate dataflow diagrams that are too large to fit in a single computer display of a typical resolution (e.g., 1024×768). In this section, we demonstrate how the zooming interfaces are incorporated into the KLIEG environment.

On one hand, a snapshot of a computation depicted by the KLIEG tracer is usually much larger than its corresponding source program. Even if a source program is small, the number of objects created during the execution can be large. Therefore, the KLIEG tracer should provide a sophisticated browsing interface so that only the portions in considerations and their related contexts be displayed. Notice that not only visual but also textual tracers/debuggers developed so far rarely provide such sophisticated browsing interfaces.

On the other hand, the KLIEG editor should support not only browsing but also editing. The zooming interfaces with editing are an important research area that most

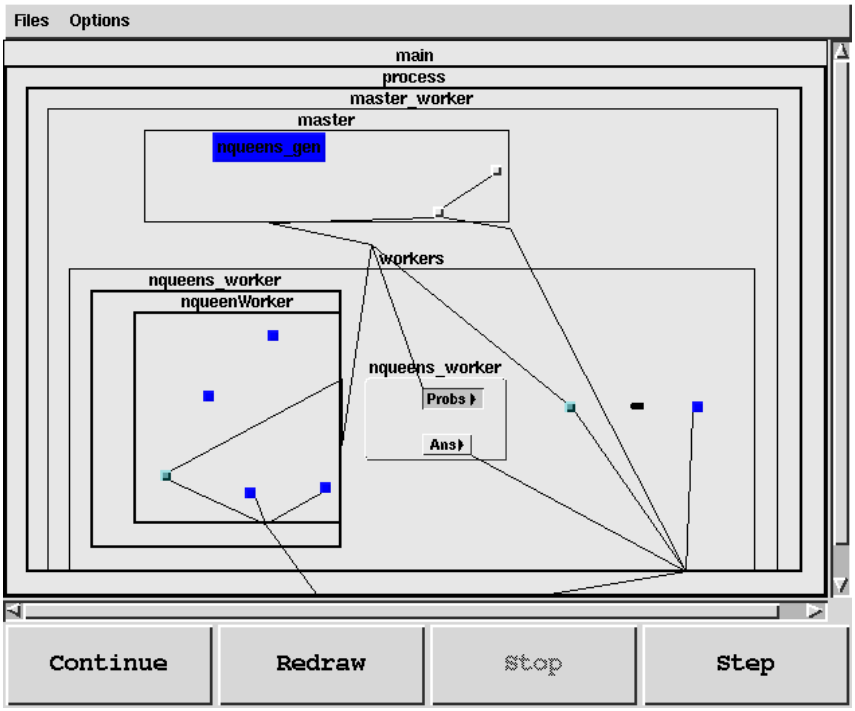

Figure 12: A zooming image of the KLIEG tracer

people do not notice.

4.1 The Zooming Interface of the KLIEG Tracer

During execution of a KLIEG program with the tracer, the user can magnify any portions of the object dataflow diagram generated by the program execution. For instance, by magnifying the nqueen_gen and the leftmost nqueen_worker in Figure 11, the user can get the image like Figure 12. The KLIEG tracer employs the continuous zoom algorithm[BHDH95] and semantic zooming for this purpose.

Although solely the continuous zoom algorithm usually works well, it sometimes fails. A typical example is a SPMD (single program, multiple data streams) computation, in which a number of objects of the same type work together in parallel. Their behaviors are essentially the same but they may have different data. If the number of the parallel objects becomes large (e.g., > 100), the continuous zoom algorithm allocates each object an equally small area like Figure 13, or otherwise it allocates sufficiently large spaces for a small number of fixed objects. The point is that the continuous zoom algorithm is designed as domain-independent and does not assume any domain specific knowledge. It cannot well handle a large number of similar objects (or nodes) that share the same parent node in the hierarchy.

Since SPMD computations often appear in practical settings, the KLIEG language and tracer provides a special support for them using domain specific information. First, the KLIEG language provides the notion of *replication pattern* which represents a number of objects of the same type. An example usage of a replication pattern is already introduced in the workers pattern in Figure 7. Second, the KLIEG tracer provides a special browsing

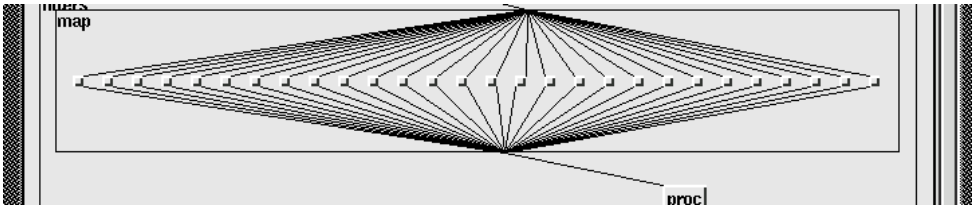

Figure 13: Normal zooming of a replication pattern

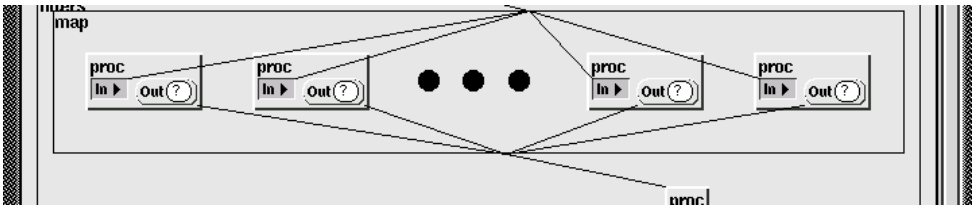

Figure 14: Semantic zooming of a replication pattern

facility for replication patterns. Namely, any object in a replication pattern can selectively be magnified and the details of the others can be omitted simultaneously. Omitted objects might be represented as "$\cdots$" and they still can be accessible by moving the foci by mouse operations. Figure 14 illustrates an example of semantic zooming provided by the KLIEG tracer. This figure and the previous one depict the same snapshot of a computation with different zooming techniques.

The reason why an object dataflow diagram representing a snapshot of a computation can significantly be larger than the source program is that sub-diagrams occurring in the source program can be copied many times during execution. Significant parts of these copy processes can often be represented by replication patterns.

4.2 The Zooming Interface of the KLIEG Editor

A KLIEG program consists of one or more modules, each of which is a collection of object and pattern definitions. For instance, Figure 15 illustrates a program consisting of modules whose names are qsort, append, and primes. Each module have small rectangles representing object interfaces, object implementations. In order to edit a program which is under development, the user first magnifies the portions that are soon to be edited and/or referenced. Figure 16 is a typical layout example, in which the qsort module is magnified.

During an editing session, any visuals might be created or deleted. This means that the default position of each node might be changed frequently during an editing session. Without some reasonable constraints, re-computation of the layout would take a long time and it would be difficult to achieve interactive responses.

Mochi Sheet [TST+97a] that provides the zooming interface of the KLIEG editor assumes a simple constraints based on griding for rapid re-computation of the layout. This is the reason why modules and definitions in Figures 15 and 16 are regularly aligned. We consider that the griding constraint of Mochi Sheet is reasonable compromise between

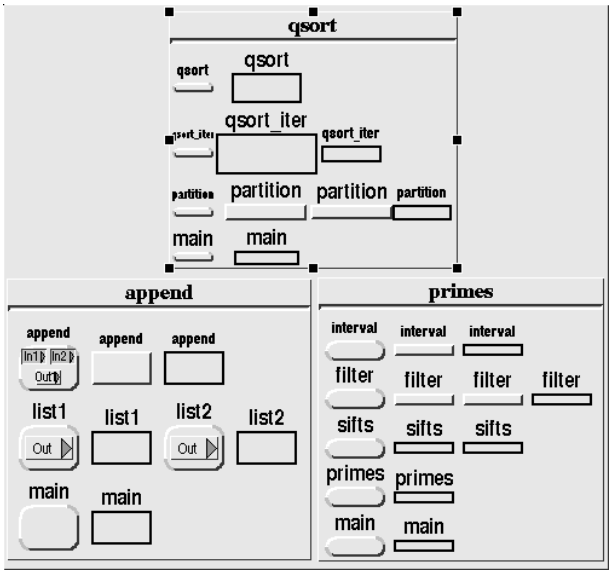

Figure 15: An initial image of the KLIEG editor

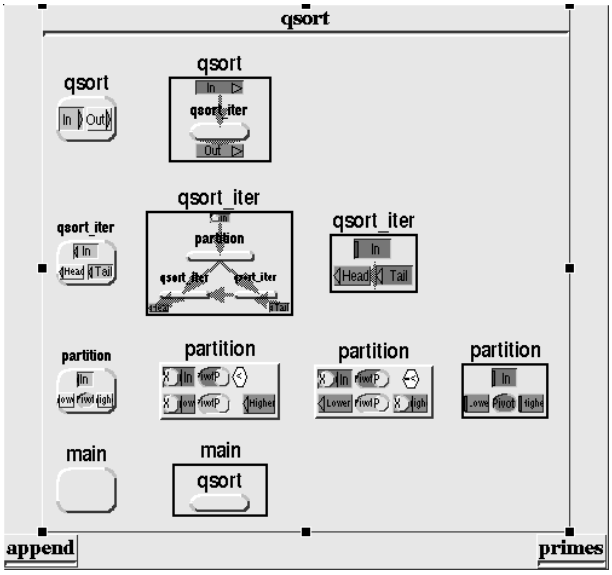

Figure 16: Magnifying a module

freedom of the layout and interactive responses.

After each editing session, a typical user changes the layout of the program to her or his most familiar one. To support user's preferences of the default layout, Mochi Sheet provides a *resize operation* so that the previous layout can be quickly recovered by simple mouse operations. In addition, the history of the layouts are kept in the system and any of them can be also recovered by mouse operations.

5 Related Work

Until now, a lot of visual parallel programming languages have been proposed including CODE[PJ92] and Pictorial Janus[KMK90]. However, most of them do not provide any mechanism for replaceable objects or processes. Therefore, it is difficult (or impossible) to explicitly define reusable software architectures nor replaceable components in these languages.

VISTA[SF94] is one of the exceptions and provides the notion of *public processor*. Although public processors in VISTA are replaced with other compatible processors, no design information is available in the VISTA programming environment. In this respect, KLIEG provides a deeper support for pattern-oriented visual programming.

For scaling up, VIPR recently introduced a fisheye zooming interface[CS96]. Still, however, it supports only a single-focus zooming. In our experiences, multi-focus zooming is better suited in editing and debugging object-based visual parallel programs partly because, in editing and debugging, we often would like to see the sender and the receiver objects simultaneously. Also, since KLIEG provides a support for multiple aspects of a visual pattern, a single-focus zooming interface is insufficient for our purpose.

6 Conclusion

Currently, visual patterns in KLIEG can have the following information of object-oriented parallel software:

- Design pattern information for parallel and distributed software;
- Layout information of objects for software visualization.

Visual patterns and supports for pattern-oriented visual programming in KLIEG integrate the design and coding stages of program development in a seamless manner. That is, both a program and its design information are represented as the same collection of visual patterns and visual objects. There are no essential differences between them.

Layout information in visual patterns are useful in particular in visual debugging. The KLIEG tracer animates a computation by generating successive images, each of which is a snapshot of the computation, using the layout information of visual patterns provided by their designer(s). In this manner, almost the same pictorial images are available not only in the design and coding states but also in the debugging stage. This sort of seamless integration is important in visual programming environments.

Our future work includes pattern-directed compilation technologies. For instance, since the master-workers pattern introduced in this paper implements a typical load balancing schema, it seems promising to attach to the pattern optional information (e.g., information for analysis and code translation) for the compiler. That is, if the compiler can recognize the master-workers pattern in a program, it might be possible to generate better codes. For the purposes, we also have to consider visual representations of compile-time metalevel architectures.

References

[BHDH95] 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*, pages 207–215, November 1995.

[Boo94] Grady Booch. *Object-Oriented Analysis and Design with Applications, Second Edition.* The Benjamin/Cummings Publishing, 1994.

[CS96] Wayne Citrin and Carlos Santiago. Incorporating fisheying into a visual programming environment. In *Proc. 1996 IEEE Symposium on Visual Languages*, pages 20–27, 1996.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[KMK90] Vijay A. Saraswat Kenneth M. Kahn. Complete Visualizations of Concurrent Programs and their Executions. In *Proc. 1990 IEEE Workshop on Visual Languages*, October 1990.

[PJ92] P.Newton and J.C.Browne. The CODE 2.0 Graphical Parallel Programming Language. In *Proc. ACM Int. Conf. on Supercomputing*, July 1992.

[Pre94] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development.* Addison-Wesley, 1994.

[RBP+91] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-Oriented Modeling and Design.* Prentice-Hall, 1991.

[SF94] Stefan Schiffer and Joachim Hans Fröhlich. Concepts and Architecture of Vista - a Multiparadigm Programming Environment. In *Proc. 1994 IEEE Symposium on Visual Languages*, 1994.

[TST+97a] M. Toyoda, B. Shizuki, S. Takahashi, , and E. Shibayama. Mochi sheet: Integration of zooming and layout editing. In *Proceedings of Interaction'97*, pages 79–86. Information Processing Society of Japan, February 1997. (In Japanese).

[TST+97b] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, and E. Shibayama. Supporting design patterns in a visual parallel data-flow programming environment. In *IEEE Symposium on Visual Languages*. IEEE, September 1997.

[UM94] Kazunori Ueda and Morita Masao. Moded Flat GHC and Its Message-Oriented Implementation Technique. *New Generation Computing*, 13(1):3–43, 1994.

[Vli97] John Vlissides. Patterns: The top 10 misconceptions. *Object Magazine*, 1997. http://www.sigs.com/publications/docs/objm/9703/9703.vlissides.html.