プログラミング 6−5
(1996. 3. 26)

並列ビジュアルプログラミング環境 KLIEG: プロセスネットワークパターンによる柔軟な再利用機構の導入

豊田正史　志築文太郎　高橋伸　柴山悦哉
東京工業大学 情報理工学研究科 数理・計算科学専攻
{toyoda,shizuki,etsuya,shin}@is.titech.ac.jp

本稿ではビジュアルな並列プログラミングにおいて，プロセスネットワークのトポロジーおよびプロトコルを再利用する単位として，プロセスネットワークパターンという概念を提案する．プロセスネットワークパターンを用いることで，よく使われる並列プログラムのパターンを設計すること，およびパターンを様々な問題を解くために再利用することが可能になる．我々はこの概念を，プロセスとストリーム通信を基にした並列ビジュアルプログラミング環境 KLIEG に導入した．この環境ではパターンの定義および再利用を図式表現を用いて対話的に行うことができる．

KLIEG: A Visual Parallel Programming Environment Using Process Network Patterns as Flexible Reuse Units

Toyoda Masashi　Shizuki Buntarou　Takahashi Shin　Shibayama Etsuya
Department of Mathematical and Computing Sciences
Tokyo Institute of Technology
{toyoda,shizuki,etsuya,shin}@is.titech.ac.jp

We propose the notion of *process network pattern*, which is a visual abstraction representing the topology and protocol of a process network, and can be a flexible and high-level unit of reuse for visual parallel programming. In this paper, we show the power of process network pattern. With process network pattern, we can first design a common pattern of parallel programs and reuse it later for solving various problems. We introduce this notion into the visual parallel programming environment KLIEG in which we can reuse and define patterns visually and interactively with its graphical user interface.

1 Introduction

In design of parallel programs, we usually use network diagrams composed of *processes* (the units of parallel execution) and *communication channels.* It is complicated work to translate network diagrams to textual programs during the implementation, moreover, it is difficult to read such textual programs.

To solve these problems various visual programming environments have been developed, such as HeNCE[1] and CODE2.0[6], in which a parallel program is a data-flow diagram, and each node in it represents a sequential process. These environments allow using diagrams directly as programs, however, they still have the reusability problem of programs. That is, although it is possible to reuse concrete nodes and diagrams, in practice partially specified diagrams are often more reusable in parallel programs.

Now we discuss this problem in more detail. In parallel programs, we use various sorts of process networks, such as divide-and-conquer networks, and pipelined structure of processes. The topology of a network and the protocol among its processes can often be reused for solving many problems. Network topologies and protocols are some of the basic design aspects of parallel programs, and can be considered as the patterns of programs.

Recently, in object-oriented community, *design pattern* approaches have emerged to describe such basic design aspects in object-oriented systems. A design pattern is a document that consists of class diagrams, object diagrams, state transition diagrams, descriptions, and example programs. Catalogs of design patterns such as [3] have been already published. The design pattern approaches are important because they make it easier to reuse programming techniques and structures of large scale programs.

Our goal of this research is to reconstruct the notion of design patterns suited for visual programming environments. As the first step toward the goal, we introduce the notion of patterns of parallel programs called *process network patterns*[1] based on process network diagrams, and support interactive reuse and definition of patterns in a visual programming environment. We put emphasis on the patterns of network diagrams, because they correspond to class/object diagrams that are the central part of design pattern catalogs. The following is the most important design issues of process network patterns:

Reusability Patterns should be highly reusable.

Definition of Patterns Patterns should be user definable.

Dynamic Networks Not only static but also dynamic networks should be represented as patterns. Still however, the visual representations of such patterns should be static.

User Interface Patterns should be defined and reused in a visual programming environment.

[1]In the following, the simplified term "pattern" may be used instead "process network pattern".

Considering the above issues, we implemented a visual parallel programming environment KLIEG. The KLIEG environment provides a support for process network patterns. Using process network patterns, we can reuse and define the topologies of networks. We also reuse the protocols among processes to some extents.

This paper consists of six sections. In Section 2, we introduce the basic programming model of KLIEG. Section 3 explains the process network patterns. Section 4 describes the environment and its implementation. We compare KLIEG with related works in Section 5. We conclude in Section 6.

2 Basic Programming Model

In KLIEG, basic programming constructs are *processes*, which are units of parallel execution. A process has input and/or output *ports*, and can have its own state. During the execution, processes compose a *process network* in which their ports are linked for representing channels, and communicate with each other via their linked ports.

A KLIEG program consists of *process network diagrams*, and definitions of process behaviors. In the following, we describe each of them.

2.1 Process Network Diagrams

Using process network diagrams, the user can describe process networks hierarchically. The largest rectangle in Figure 1 illustrates an example of a process network diagram, which computes $\sum_{n=1}^{\mathsf{N}}(n^2 + n)$. This diagram contains three processes, naturals, squares, and sum, which have their input and output ports represented by sunken and raised rectangles, respectively. The diagram also contains its own ports N and Sum for the processes to be able to communicate with outside processes. The process network is constructed by linking the ports in the diagram.

There are two types of ports, *stream ports* and *singleton ports.* Stream ports transfer a sequence of data, and singleton ports transfer only one datum. Stream ports and singleton ports are represented by rectangles and round rectangles, respectively. In Figure 1, the port N of naturals is an input singleton port, and the port Out of naturals is an output stream port. Then, the behaviors of the processes are as follows: naturals generates a stream of integers from 1 to N; squares generates the stream of the squares of the received integers; and sum computes the sum of the received integers.

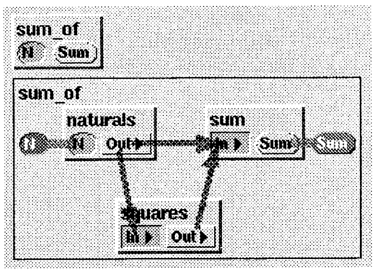

Figure 1: Process network diagram (color original)

Data can be multi-casted by linking a single output port to multiple input ports. In Figure 1, the output stream of naturals is multi-casted to the input stream ports of squares and sum. It is also possible to merge multiple streams to one stream by linking several output stream ports into one input stream port. In a merged stream, the order of data is nondeterministic, but the arrival order among data from the same output stream is preserved. In Figure 1, two output streams from naturals and squares are merged into the input stream of sum. Thus, sum computes the sum of both natural numbers less than or equal to N and their squares so that it outputs $\sum_{n=1}^{N}(n^2+n)$ to the port Sum.

The user can use a process network as a single process called a *composite process*. For example, the diagram in Figure 1 can be used as the composite process sum_of shown at the top. Using composite processes in network diagrams, the user can construct process networks hierarchically.

We can also describe dynamic changes of the topologies of networks using process network diagrams and some extra mechanisms. However, these are not the main issues of this paper, we omit these explanations.

2.2 Process Behaviors

To describe behaviors of a non-composite process, we use pictorial rules that represent state transitions of the process. The basic model of KLIEG is based on KL1[9][2], and a rule is the pictorial representation of a single KL1 clause. However, the rules are too low-level to construct large programs, so the environment of KLIEG provides higher level units such as process network patterns (See Section 3), and decrease the use of low-level descriptions. We also omit the explanation of the rules in this paper.

[2]KL1 is a parallel logic programming language developed at ICOT.

3 Process Network Patterns

Process network patterns are flexible reuse elements of process networks and protocols among processes. We can reuse a pattern, which is an incomplete process network, for solving various problems by specializing the pattern with specific processes. In KLIEG, a process network pattern is represented as a process network some of whose internal processes are not specified. We can substitute a process that works appropriately in the pattern for an unspecified part called a *hole*, so that we can reuse the network topologies. The protocol among processes can be also reused to some extent. We can also construct patterns hierarchically to define complex patterns. Moreover, we can achieve both reuse and definition of patterns simply and intuitively using the drag-and-dropping interface of KLIEG. In this way, process network patterns provide high-level abstraction of process networks.

3.1 An Example

As a first 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. The master partitions a problem into sub-problems, and each worker computes sub-problems in parallel. Figure 2 depicts the master-worker pattern in KLIEG. A pattern is represented by a large sunken rectangle that includes a process network. The pattern master_worker [3] consists of the master pattern and the workers pattern. The master pattern is composed of two processes, generator and dispatcher [4]. The generator process simply generates a stream of sub-problems. The dispatcher process receives sub-problems from generator and messages from workers, and sends the sub-problems to ready workers. It also sends all the solutions from workers to Answer. The dispatcher process makes streams corresponding to the workers, and sends them via the Workers port to communicate with the workers. The number of streams is determined by the input port WorkerNum. Then, dispatcher forwards the sub-problems via corresponding streams. The workers pattern represents multiple workers, and the number of workers is dynamically determined by the number of streams in Ws. The workers pattern is defined by a *sequence pattern* that represents such a sequence of processes (See also Section 3.3.2). Each worker receives sub-problems, and solves them, then returns answers to the master.

[3]The first letter of the name of a process, a hole, or a pattern is lower case, and the name of a port begins with an upper case letter.

[4]A process can be iconified to save the space. The dispatcher process is iconified, so its ports are hidden.

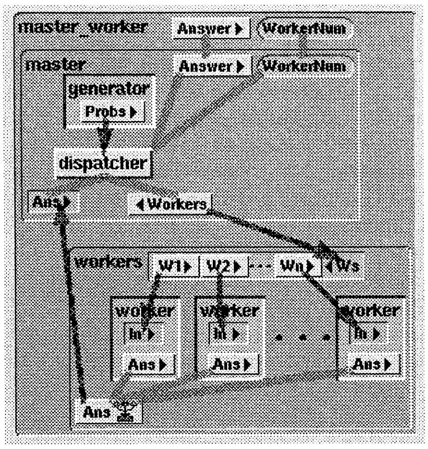

Figure 2: Master worker pattern in KLIEG

As shown in Figure 2, a pattern has its own ports as a process does, so they can be used as processes. For example, master has the ports Answer, Ans, Workers, and WorkerNum. To represent holes, we use sunken rectangles that may include ports. For example, there are holes named generator and worker in the master_worker pattern.

3.2 Reusing Patterns

We can reuse topologies of networks simply by substituting specific processes for holes of patterns by drag-and-dropping processes to the holes. Ports in a hole restrict processes that can be embedded in the hole. That is, when a process is substituted for a hole, it must have at least the same type ports defined in the hole. When a process is substituted for a hole, the ports of the process are automatically linked as the ports of the hole were connected. If there is ambiguity, the ports of the process may not be linked properly as the user intended. In that case, the user can change links among them correctly.

The master_worker pattern can be used for solving many problems, such as ray-tracing computations and various search problems. For example, we solve N-Queens problem by AND-parallel search using the master_worker pattern. Figure 3 depicts the program for the N-Queens problem. To construct this program, we only have to drag-and-drop the nqueensGenerator and the nqueensWorker to the appropriate holes and to add necessary ports to the pattern. The nqueensGenerator generates partial solutions whose first columns are fixed, and each nqueensWorker solves those sub-problems sent to it. When all the holes in a pattern are specified, the pattern can be treated as a concrete process. To represent this change visually, the patterns become raised

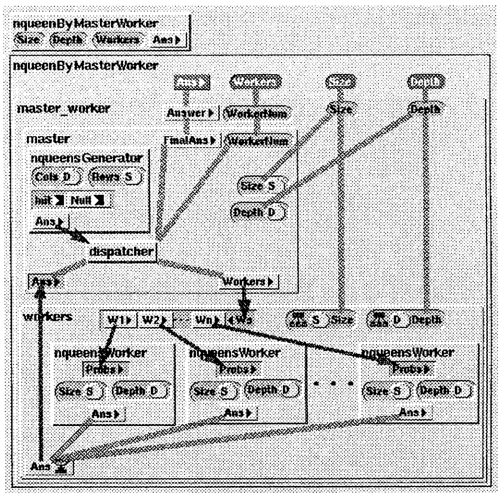

Figure 3: N-Queens by the master worker pattern

shapes. The port Size and Depth are added to designate the size of the chess board and the number of columns fixed by the nqueensGenerator, respectively.

We can also reuse the protocol among processes to some extent. When some of processes in a pattern represent a part of the protocol, and are common to many problems, we can reuse the pattern whose holes are substituted by those processes. For example, dispatcher in the master_worker pattern represents a part of the protocol among master and workers so that it is specified in Figure2. Note that the user can change the protocol of a pattern by changing specified processes. For example, when we reuse the master_worker pattern for OR-parallel search problems, we change the dispatcher process to a process that terminates all the computation when it receives one answer from workers.

3.3 Defining patterns

The user can define patterns by constructing networks with holes and ports, and by substituting some of their holes. It is also possible to construct large and complex patterns hierarchically by substituting a pattern for a hole. This makes it possible to reuse large and complex networks flexibly, and provides some scalability to programs. Currently, the user can define two kinds of patterns, *static patterns* and *sequence patterns.* A static pattern represents a static network whose holes and ports are freely arranged by the user. A sequence pattern represents a dynamic network composed of a sequence of processes in one static diagram. In this section, we describe, as an example, how to construct the master_worker pattern in Figure 2.

3.3.1 Static Patterns

Static patterns represent static process networks. The user can define static patterns by arranging holes and ports freely, and by linking ports. For example, we define a base pattern of the master_worker pattern as a static pattern. Figure 4 illustrates the base pattern of master-worker pattern in KLIEG.

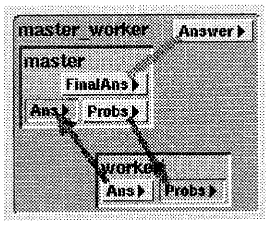

Figure 4: Base of master-worker pattern

3.3.2 Sequence Patterns

A sequence pattern represents a dynamic network composed of a sequence of processes sharing the same definition in one static diagram. The number of the processes and their topologies are dynamically determined. The internal processes in a sequence pattern are arranged in a sequence, and each of them may or may not communicate with its neighbors and with outside processes by the pattern's own ports.

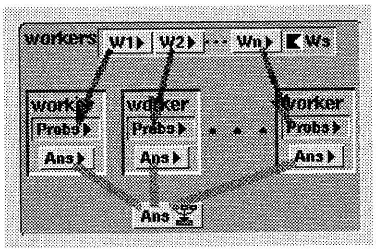

Figure 5: Multiple workers

Figure 5 depicts the workers pattern which represents the worker part of the master_worker pattern. A sequence pattern includes three holes in the sequence with an ellipsis $(\cdots)$ between the second and the third holes. Each hole represents the first, second, and the last process respectively. Changes to one of these holes are propagated to all the other holes. For example, when the user specializes one of these holes by a process, all the other holes are automatically specialized by the processes with the same definition.

The user can use three kinds of special ports to define the topologies of sequence patterns. The following is the list of special ports. A description in parentheses represents available types for each port.

Map port (input/output, stream) An input map port maps its elements to processes. An output map port sends outputs of all processes in the same order of the processes.

Broadcast port (input, stream/singleton) It broadcasts its data to all processes.

Merge port (output, stream) It merges output streams of all processes.

For example, the workers pattern includes two special ports. The port Ws is a map port. Ws includes a sequence of the stream ports that is represented by three ports with an ellipsis $(\cdots)$. Each port is linked to the Probs ports of the corresponding worker. Another port Ans in workers is a merge port. It merges the outputs from the Ans port of the all processes.

It is still the problem how many processes should be created in the pattern. In the workers, it is controlled by the input map port Ws. That is, the number of internal processes is adjusted to the number of elements of the port.

Sequence patterns can represent various topologies. Pipelined and bi-directional communication structures can be described by linking the ports of neighbors. Divide-and-conquer structures can be described by combinations of the special ports. It is also possible to mix these communication structures.

3.3.3 Hierarchical Composition of Patterns

To define larger and more complex patterns, KLIEG allows hierarchical compositions of patterns, which provides some scalability to programs. We can achieve this by substituting a pattern for a hole by drag-and-dropping the pattern to the hole. For example, we can substitute the workers pattern in Figure 5 for the worker hole of the base master_worker pattern in Figure 4. We can define the master_worker pattern in Figure 2 by substituting, in addition, an appropriate pattern for the master hole.

4 Environment and Implementation

The KLIEG environment consists of three main components: *editor*, *translator*, and *tracer*. Figure 6 shows the snapshot of the KLIEG environment. In this Figure, the user has completed the program on the editor (the left window) and is viewing the execution on the tracer (the right window).

The editor is used for constructing processes and patterns and composing them up into a KLIEG program. The user can edit some *modules*, a set of processes and patterns related to each other, in multiple windows. The translator parses a pictorial KLIEG program and translates it into KL1 codes. The translated code has information about position and size of processes in the program as comments for the tracer. The tracer visualizes the creation

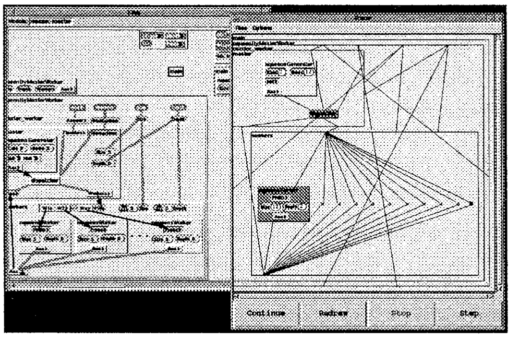

Figure 6: Snapshot of the KLIEG environment

of processes and dynamic changes of the network topology during the execution of a program. On the tracer, processes are automatically layouted based on the geometric information, such as the position and scale of processes, in the code. In addition, it can display the contents of streams and zoom in/out a part of the network in which the user is interested. To execute translated KL1 codes, we use KLIC[2], a portable compiler of KL1 developed at ICOT.

All components of the KLIEG environment are implemented mainly in C++ and Amulet[5] user interface development environment.

5 Related Work

In this section, we compare KLIEG with several related works, and discuss about them.

VISTA[7] is a visual multiparadigm programming environment. VISTA's programming constructs are processors, which may have an internal network of processors. A processor can have some internal processors called public processor that can be substituted by another processor having a compatible interface. A public processor can be substituted dynamically, but the topology of the network is static. In KLIEG, process cannot be substituted dynamically, but the topology of the network can be dynamically changed. Therefore, KLIEG provides more intuitive diagrams of dynamic networks. In addition, A processor substituted for a public processor must have a compatible interface in VISTA. In KLIEG, a process doesn't have to have a compatible interface of a hole, that is, the process can have more ports than the hole. Thus KLIEG provides higher-level abstraction for process networks than VISTA.

There are also some visual languages based on parallel logic programming language, such as Pictorial Janus[4] and PP[8]. Both Pictorial Janus and PP provides a single form of pictorial rewriting rules to visualize clauses of parallel logic programming languages, so that it is difficult to construct large scale programs. KLIEG provides higher-level abstraction such as processes, streams, and process network patterns, so that it is easier programming in the large.

6 Conclusion

In this paper, we have proposed the notion of process network pattern, which is flexible and high-level element of reuse, and introduce this notion into the visual parallel programming environment KLIEG. Process network patterns provide high-level abstraction of process networks. They make it possible to define and to reuse the topologies and the protocols of process networks. The user can reuse patterns visually and interactively by drag-and-dropping processes or patterns to their holes.

In future work, we will extend the process network patterns more flexibly and expressively for more general network and protocol reuse. In addition, we will make the catalogs of various patterns using process network patterns.

References

[1] A. Beguelin, J. J. Dongarra, G. A. Geist, R.Manchek, and V. S. Sunderam. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing 91*, pp. 435–444, 1991.

[2] Tetsuro Fujise, Takashi Chikayama, Kazuaki Rokusawa, and Akihiko Nakase. KLIC: A Portable Implementation of KL1. In *International Symposium on Fifth Generation Computer Systems 1994*, pp. 66–79. ICOT, December 1994.

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

[4] Kenneth M. Kahn. Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pp. 943–950. ICOT, 1992.

[5] Richard McDaniel and Brad A. Myers. Amulet's Dynamic And Flexible Prototype-Instance Object And Constraint System In C++. Technical Report CMU-CS-95-176, Carnegie Mellon University School of Computer Science, 1995.

[6] P.Newton and J.C.Browne. The code 2.0 graphical parallel programming language. In *Proc. ACM Int. Conf. on Supercomputing*, July 1992.

[7] 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.

[8] 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.

[9] Kazunori Ueda and Takashi Chikayama. Design of the Kernel Language for the Parallel Inference. *The Computer Journal*, Vol. 33, No. 6, pp. 494–500, December 1990.