I-Pigs: an Interactive Graphical Environment for Concurrent Programming

M.-C. PONG*

Computing Laboratory, University of Kent at Canterbury, England CT2 7NF

This paper describes the programming environment I-PIGS which supports the graphical concurrent programming language Pigsty. Pigsty uses graphical icons to represent processes, communication ports and links; and uses structured charts to represent the constructs of control flow in a process. I-PIGS supports the editing of a Pigsty program graphically, and executes the graphical program directly. I-PIGS guarantees that the graphical program is syntactically and semantically correct. During the execution of a Pigsty program, I-PIGS animates data communication and shows any deadlock situation on the screen. These capabilities help the user to understand the structure and the behaviour of his program.

Received September 1987, revised August 1990

1. INTRODUCTION

This paper describes the interactive graphical environment I-PIGS[1] which supports concurrent programming in a specially-designed graphical language Pigsty. ('Pigsty' stands for the *sty*le of programming using *PIGS*.) I-PIGS is an integrated environment in which a user (i.e. a programmer) can edit and debug his graphical concurrent Pigsty program directly. I-PIGS is different from other earlier environments (e.g. Cornell Program Synthesizer[2]) in supporting the execution of graphical programs. It also distinguishes itself from other graphical environments (e.g. PIGS,[3] 2-PIGS,[4] Pict,[5] PECAN[6] and GARDEN[7]) in supporting concurrent programming (including distributed and parallel programming).

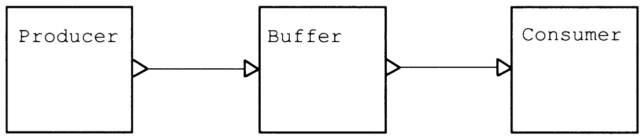
Debugging concurrent program (running in one or more processors), or distributed or parallel program (running in more than one processors) is difficult.[8] Using sequential debugger to debug individual processes of a distributed or parallel program is not satisfactory. Any attempt to gather information about a process may affect the timing and communications among the processes. This makes it difficult to understand how the communications take place. I-PIGS is not only useful for developing concurrent programs running in uniprocessor. By first develop and debug a distributed or parallel program in a concurrent programming environment such as I-PIGS, program errors in inter-process communications can be more readily located than debugging in a real parallel execution environment.

The main advantage of using I-PIGS is that it helps the user to visualise and understand his concurrent program more clearly, especially with respect to the interconnection structure of a system of processes (the *system structure*) and the data communication between processes. Fig. 1 shows a Pigsty system structure. The equivalent representation in some probable textual language is also shown. It can be seen that the graphical representation is clearer.

I-PIGS was implemented as a C program running in the PERQ workstation under the operating system PNX-2. The following is a summary of the features of I-PIGS.

* Now at: Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay Road, Kowloon, Hong Kong.

(*a*) Graphical representation.



(*b*) Textual representation.

```
Producer || Buffer || Consumer;
LINK Producer.outPort TO Buffer.inPort;
LINK Buffer.outPort TO Consumer.inPort;
```

Figure 1. Interconnection of processes.

Editing aspects:

- provides a graphical *system editor* to edit the system structure and a *chart editor* to edit the 'coding' of a process in structured chart form;
- checks that the graphical system structure and chart program are syntactically and semantically correct;

execution and debugging aspects:

- executes the system of processes by means of interpretation and simulated concurrency;
- for the process being executed, displays its variables and current values in separate windows;
- allows the user to set breakpoints;
- at a breakpoint, allows the user to examine or assign the declared variables of any process of the Pigsty program;

concurrent programming aspects:

- highlights the communication ports ready to send or receive data, and animates data communication;
- gives a diagnostic message when the program is deadlocked.

In the rest of this paper, the language Pigsty, and the editing and execution support by I-PIGS will be described.

2. THE GRAPHICAL LANGUAGE PIGSTY

2.1. Overview

Pigsty is not a language attempting to tackle all problems in concurrent programming. Rather, it was designed to test the idea that interactive graphics could be used effectively to support concurrent programming. The design of Pigsty is based on CSP[9] and Pascal.[10] The

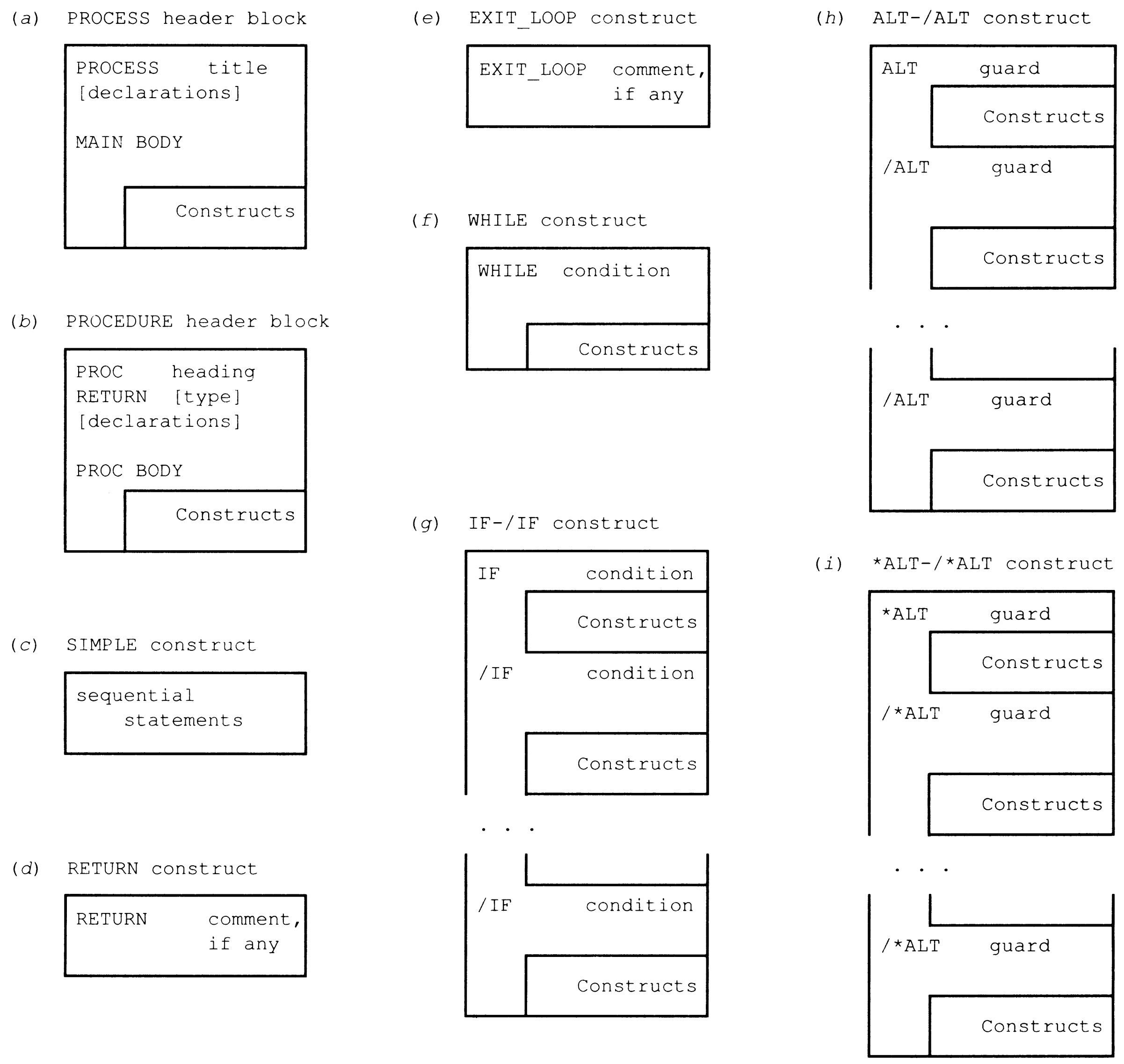


Figure 2. Pigsty structured charts.

primitive CSP synchronous communication mechanism is used.

Pigsty uses a mixture of graphics and text to represent a program. Pigsty only uses graphical representations for those aspects which are better presented graphically, viz: the *system structure* of processes and the constructs of control flow in a process in the form of *structured charts*.[3,4,11]

The pure textual part of Pigsty includes Pascal declarations, boolean conditions, sequential statements (assignments and procedure calls), and CSP-like communication commands. They are written inside the structured charts.

The graphical features of Pigsty are described below.

2.2. Pigsty processes and chart program

A Pigsty program is a system of *basic* processes and/or *high-level* processes. A high-level process is composed of a system of basic and/or high-level processes. A basic process is implemented in the form of a sequential *chart program*. Note that a basic process can be a single process or an element of an array of processes (called an *element process*).

A chart program is composed of structured chart *constructs* (Fig. 2). In Fig. 2, the positions marked with the term 'Constructs' could be any other constructs as long as they form a semantically correct program. Dijkstra's alternative **IF-FI** and repetitive **DO-OD** guarded commands[12] are included. They become the **ALT-/ALT** and ***ALT-/*ALT** constructs respectively. The use of the keywords **ALT** and ***ALT** is prompted by CSP and occam.[13] They appear to aid readability. The uniform use of '/' in the alternative choices of the constructs suggests the common alternative nature.

The most attractive features of Pigsty are related to concurrent programming. The following subsections will concentrate on these aspects.

2.3. Communications ports and links

Pigsty uses the concept of *ports* for communication.[14] A process sends or receives data to or from its ports. Ports are responsible for the actual communication. They are differentiated into *OutPorts* and *InPorts*. An OutPort can send data to an InPort only if they are 'linked' (i.e. a *link* exists between them). Synchronous communication takes place when the linked OutPort and InPort are both ready to communicate; otherwise, the ready port has to wait for its partner to become ready.

Ports do not have associated data types. Data of different Pascal pre-defined types can be transmitted through the same port. This reduces the number of ports required if data of different types are transmitted between two processes. Run-time type checking is performed when a sent value is assigned to a receiving variable.

Port names are used in the communication commands of the process, in contrast to the use of process names in the case of CSP. Thus, the syntax of the input command is

InPort-name ? receiving-variable

and the syntax of the output command is

OutPort-name ! expression

When an output command of a process is executed, the value of the expression is sent to the OutPort, which becomes ready to send. When an input command of a communicating process is executed, the InPort becomes ready to receive a value. Once a value is sent from the OutPort to the InPort via synchronous communication, it is immediately assigned to the receiving variable.

Note that port names are local to the process to which the ports belong. Using port names in communication commands offers potential to build more modular programs. Suppose that in a concurrent program, process P communicates with process Q originally. Now we want process P to communicate with process R instead. In the case of CSP, all the communication commands involving Q in the coding of P must be changed to R. In Pigsty, the communication links are hidden from the coding of the processes. The links could be changed independently. Thus, we need only delete the link between the ports of P and Q, and insert a link between the ports of P and R by using the editing commands of I-PIGS. The implementation of communication between ports would take care of the communication through the new link.

2.4. Icons

The icons representing the concurrent programming features of Pigsty are as follows (their appearances are shown in Fig. 3):

- A square box-like icon ('box' for short) enclosing a process name represents a basic process. An element process icon includes a horizontal line dividing a box icon and the element index is shown in the lower portion of the box.
- A double-lined boundary box icon enclosing a process name represents a high-level process.
- A triangle with an edge on the boundary of a box represents an OutPort of a process, and a triangle with a vertex on the boundary of a box represents an InPort of a process. Ports names are not shown by the side of the icons to avoid cluttering the screen.
- A straight line connecting an InPort icon and an OutPort icon represents a communication link between the two ports.

In the following, the terms *box* and *process* might be used interchangeably, without emphasising that a box is the icon to represent the concept of a process. Also, the term *port* might be used to represent the icon or the concept of a port. It should be clear from the context that whether the icon is being referred to.

Remarks. Theoretically speaking, the definition of a language should be independent of any implementation, and we may assume that a (graphical) language can support programs consisting of arrays of processes of any dimensions. However, due to implementation limits, Pigsty programs are presently limited to consist of single processes and one-dimensional arrays of processes only.

3. EDITING FACILITIES OF I-PIGS

I-PIGS provides a *system editor* for editing system structure and a *chart editor* for editing chart program. When a user runs I-PIGS in a window of the workstation (called the *main window*), the system editor is activated; under which, the user can issue a command to invoke the chart editor. These editors are described below.

3.1. Editing of chart program

When the chart editor is invoked, the user can insert, delete, move and copy structured chart constructs and input text inside the constructs in a special *chart-program window*. The chart editor is a structured editor in the conventional sense. It guarantees that the constructs are composed correctly. For example, I-PIGS only allows the user to insert an **EXIT_LOOP** construct inside an iterative construct.

Note that the chart program created and edited for any element of an array of processes is regarded as identical for all other elements. I-PIGS maintains such consistency by actually keeping only one copy of the chart program for all elements. During execution, this copy serves as the 'reentrant code' for the element processes.

The editing of concurrent Pigsty features is supported by the system editor, which is described in the following subsections.

3.2. Editing of basic process icons

I-PIGS provides two commands for creating basic

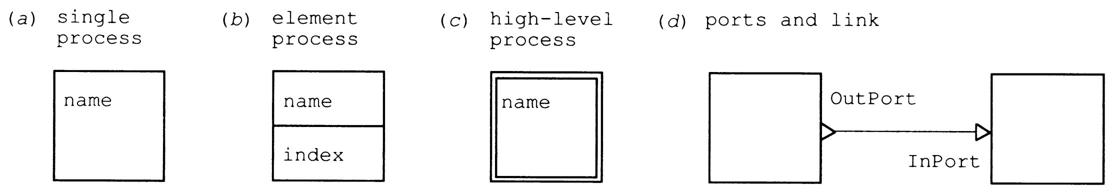


Figure 3. Pigsty icons for concurrent programming.

processes: 'create a single process' and 'create an array of processes'. Similarly, there are two corresponding commands for deleting processes.

On accepting the 'create a single process' command, I-PIGS prompts the user to fill in the name of the process and use the mouse of the workstation to position a box icon in the main window. On accepting the 'create an array of processes' command, I-PIGS prompts for the process name and the array bounds, and asks whether the boxes should be drawn horizontally or vertically. Then, after the user chooses the central position for the boxes in the window, I-PIGS displays a row or a column of boxes accordingly. I-PIGS checks that the new box(es) would not overlap any existing boxes before accepting the position indicated by the user. I-PIGS also checks that the user-supplied name has not been used before; i.e. checks against 'duplicate declaration'. Fig. 4 shows a

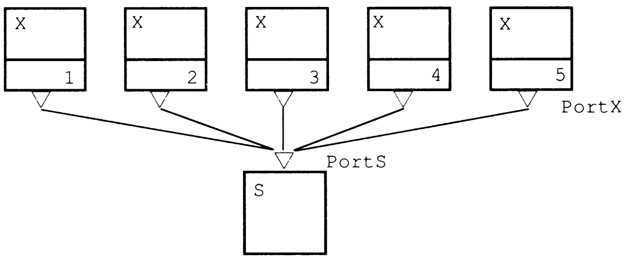


Figure 4. A single process linked to an array of processes.

single process *S* and an array of processes *X*. (Note that the port names in all Figures are shown for illustration purposes; they actually do not appear in the window to avoid cluttering.)

I-PIGS also provides editing commands for moving a single process, an element process, or a whole array of processes. (Any port icons on the box icons are moved together.) The positions of boxes are immaterial to the semantics of the program. The 'move box' commands are provided for the user to lay out the system structure nicely.

I-PIGS supports the scrolling of the whole system structure leftward, rightward, upward and downward. This scrolling capability allows the display of boxes to extend over an area bigger than the current window size.

In addition, interactive graphics techniques[15] could profitably be used to scale or rotate part or all of the display of boxes. A scaled down view of the whole system structure and another blown up view of part of the system could be displayed at the same time, possibly in different windows; though, these capabilities are not included in the current implementation of I-PIGS.

3.3. Editing of port icons

I-PIGS provides two editing commands for creating ports: 'create InPort' and 'create OutPort'; and a command for deleting a port. On accepting either 'create port' command, I-PIGS prompts the user to enter the name of the port and to position the port icon anywhere on the boundary of a box. I-PIGS checks that the user can only place a port icon on the boundary of a box, not elsewhere. The position of a port on a box is immaterial to the semantics of the program.

I-PIGS checks against 'duplicate declaration' of port names in a process on accepting a 'create port' command. To avoid cluttering the window, the port names are not shown. Editor commands are provided for examining and changing port names and process names. I-PIGS also checks against 'duplicate declaration' on accepting a 'change name' command.

If the user places the port icon on an element process icon, I-PIGS automatically displays port icons on all brother element processes at the same relative position on the boundaries of the boxes (e.g., the ports of process array *X* in Fig. 4). To guarantee a nice layout, I-PIGS checks that port icons do not overlap each other.

The user can also give an editing command to move a port icon to another position on the boundary of the same box. However, I-PIGS does not allow the user to move the port to another box. If a port icon on an element process icon is moved, its brother port icons on all brother element process icons are automatically moved accordingly. Moreover, any link on the port is also moved. It can result in a nicer layout of the link pattern (Fig. 5).

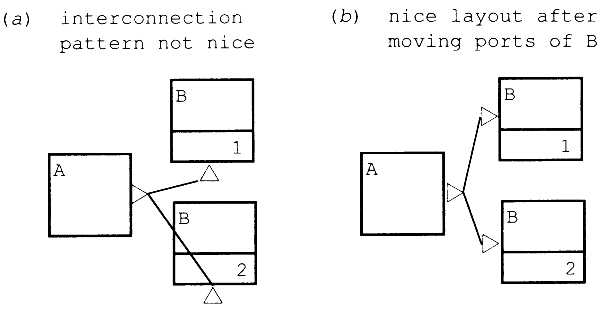

(a) interconnection pattern not nice
(b) nice layout after moving ports of B
A
B
1
B
2
A
B
1
B
2

Figure 5. Moving ports to give a nice layout.

3.4. Editing of links

The user can issue an editing command to insert a link between an InPort of a process and an OutPort of another process. I-PIGS guards against the user's (mistaken) intent to insert a link between a pair of InPort icons, or a pair of OutPort icons, or a pair of port icons on the same box. This helps to simplify the syntax and semantics analysis for execution of a Pigsty program.

I-PIGS provides three different 'insert link' commands for three different situations:

one-to-one link
one-to-many links
many-to-many links

Correspondingly, there are three 'delete link' commands for the three situations. The following discussion is about 'insert link'. The corresponding 'delete link' commands are similar.

One-to-one links can be inserted between single and single processes, element and element processes, or single and element processes. They are the simplest and need no further explanation.

One-to-many links are used to interconnect a single process and an array of processes. Fig. 4 is an example of a process *S* linked with a process array *X*. Only one single InPort, *PortS*, is placed on the boundary of box *S*. When the user links this port to the OutPort of any element process *X*, I-PIGS understands that *PortS* should be an array of ports, each element of which is linked to the OutPort of an element process *X*. *PortS* is not redrawn as five instances of InPorts on the screen, since this would make the screen become too cluttered. Multiple incidences of the links to/from a port already indicate that it is an array of ports, where each element port is connected to another port.

Many-to-many links are inserted to connect two arrays of processes, which can be the same array. Only one corresponding pair of InPort and OutPort needs to be chosen. I-PIGS will insert links throughout the ranges of elements as far as possible. Fig. 6 shows a simple case of identical number of elements in the two process arrays. Same number of links as the number of element pairs are inserted. Fig. 7 is a case of unequal number of elements in the two arrays. I-PIGS first inserts a link between the chosen pairs of ports on processes *A[2]* and *B[3]*. Then it scans the process arrays backward and puts in links for elements of decreasing indices. There is only one more link that can be inserted in the backward scan, i.e. between *A[1]* and *B[2]*. Afterwards, I-PIGS scans forward to insert links between elements of increasing indices, i.e. between *A[3]* and *B[4]*. During the backward or forward scans of elements, if I-PIGS encounters a port with a link already or comes to an extreme element, I-PIGS will stop inserting any more link.

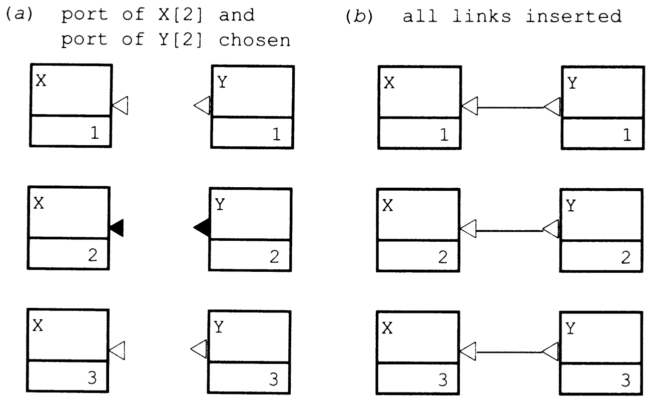


Figure 6. To insert many-to-many links between process arrays.

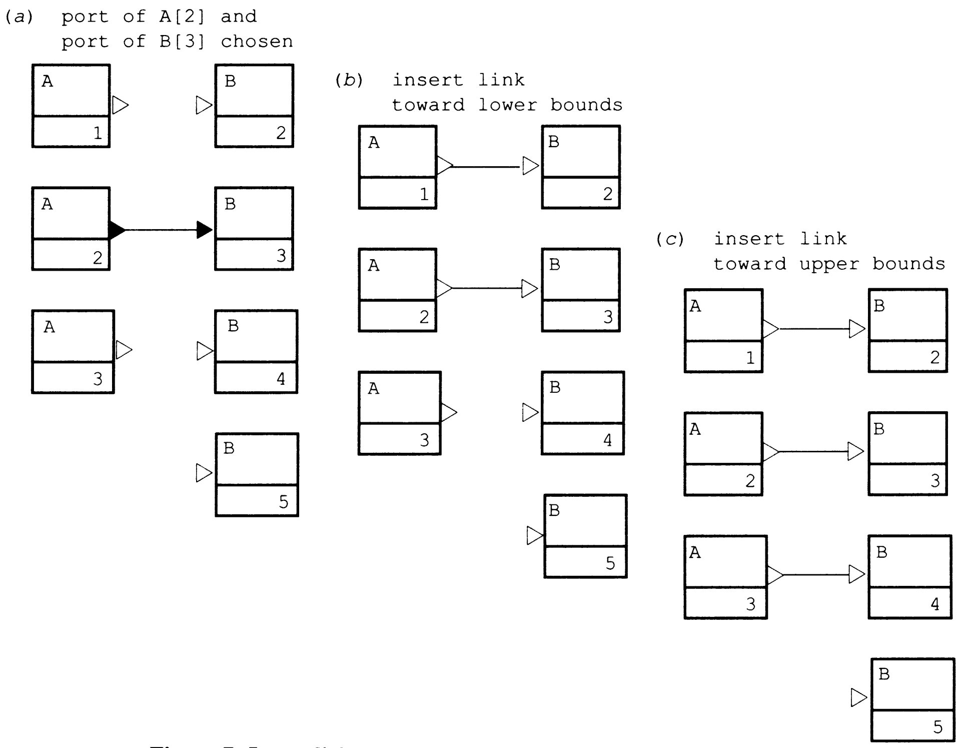
Note that only backward and forward scans for link insertion are performed, and links are not inserted automatically in a wrap-around manner. This is because the user may not want wrap-around connection. Rather, I-PIGS lets the user connect the extreme elements of a process array at his discretion. For example, in Fig. 8(*a*), the user issues the 'insert many-to-many links' command to insert links between element processes *MULTI* to form a pipeline. *MULTI[1]* has no input link and *MULTI[4]* has no output link. The user can give the 'insert one-to-one link' commands twice to insert these two links, say, to single processes *SOURCE* and *SINK* as shown in fig. 8(*b*).

3.5. Editing of high-level processes

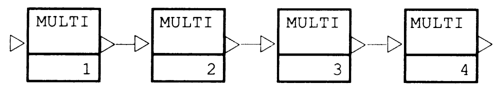
Pigsty has the feature *high-level process*. I-PIGS represents a high-level process as a *high-level box*, which is a box icon with double-lined boundary. Such icon gives a clear indication to the user that it is composed of other processes, and is not implemented as a chart program.

I-PIGS supports both the top-down and bottom-up approaches to the development of a system of processes. This is achieved by providing the 'refine box' and the 'group boxes' editing commands.

Fig. 9 is an example of refining process *A*. On accepting a 'refine box' command on a chosen box, I-PIGS creates a new *refine-box window*. The user can create a new subsystem of processes with internal links in the new window using the editing commands of I-PIGS. (The user may even create a subsystem of processes beforehand, save it in a file, and read it in at this time.)


(a) port of A[2] and port of B[3] chosen
(b) insert link toward lower bounds
(c) insert link toward upper bounds
A
B
1
2
3
4
5
Figure 7. Insert links between unequal number of element processes.

(*a*) Links between process array itself.



(*b*) Extreme element processes linked to single processes.

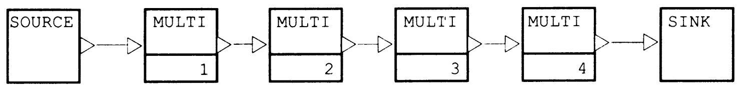
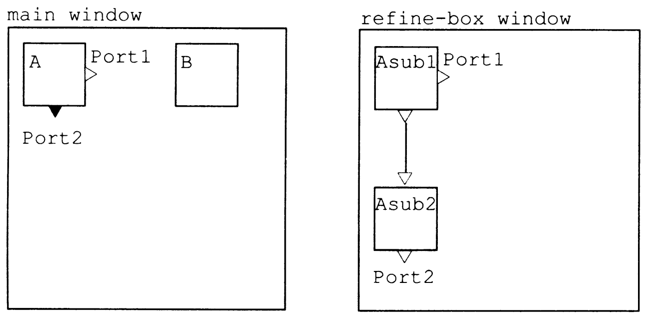


Figure 8. Link to form a pipeline of processes.

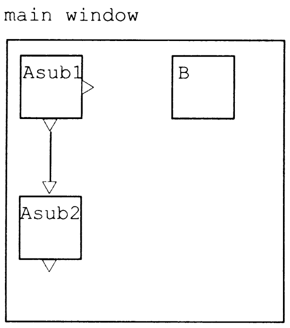
The subsystem will become the refinement of the chosen box, which becomes a high-level box.

A high-level process is linked to the outside world via its ports and the subsystem must be linked to the outside world via the same ports. Thus, when the user indicates the end of building the subsystem, I-PIGS scans through the list of ports of the high-level box and prompts the user to put the corresponding ports, one by one, on any box of the subsystem. These ports become the interface of the subsystem to the outside world (see Fig. 9(*a*)). There is no need to name these ports as they have names already.

After the refinement has been done, the refine-box window is destroyed and the refinement is shown instead of the original box in the main window. Fig. 9(*b*) is the final display. Any links of the interface ports on the high-level box will be identified by I-PIGS and the information will be recorded into the sub-level ports.



(*a*) Refine box *A* to a subsystem.
In the refine-box window, box *A* has been refined to a subsystem of boxes *Asub1* and *Asub2*. The interface port *Port1* has been placed on *Asub1*. Another interface port *Port2* (highlighted in the main window) is just placed on *Asub2*.



(*b*) End of refining box *A*.
The subsystem of boxes *Asub1* and *Asub2* are shown in the main window instead of box *A*.

Figure 9. Refine a box.

Since a hybrid of top-down and bottom-up development always occurs in practice, I-PIGS also supports the bottom-up grouping of one or more boxes. When the user issues the 'group boxes' command, I-PIGS will prompt for the top left corner and the bottom right corner of a rectangular region. Any boxes falling within this region will be grouped. However, if a single box or an array of boxes is partially inside the region, I-PIGS will give a warning message and abort the command.

If the region is acceptable, I-PIGS creates a new *group-boxes window* and displays a new high-level box in it. The user is prompted to name the high-level box, and then select port icons on the group of boxes to serve as interface ports to the outside world. On selecting an interface port, I-PIGS asks the user to place the corresponding port icon on the high-level box. I-PIGS will not prompt for new names for these ports because they must have the same names as in the subsystem. Links are allowed between the interface ports and the outside world while only internal links are allowed among the other ports of the processes in the subsystem. Fig. 10 is an example of grouping boxes *A1* and *A2* to form the high-level box *A*, with *Port1* as the only interface port.

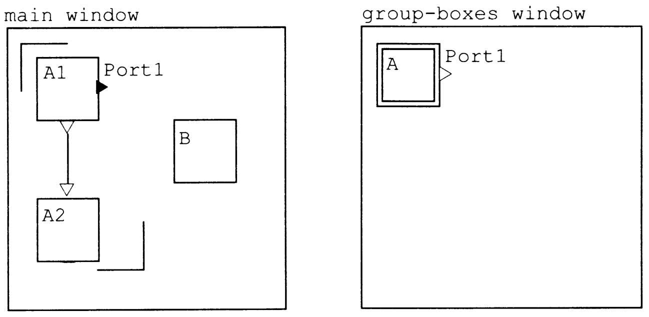
Notes. In order to enforce a consistent and disciplined editing habit, I-PIGS only allows the insertion and deletion of ports and links in the bottom-most level of a tree of box refinements. I-PIGS will update the corresponding port and link information in the high-level boxes automatically.

Insertion and deletion of boxes can be done at any level, but the deletion of a high-level box (requires confirmation by the user) will also remove all its sub-level boxes.

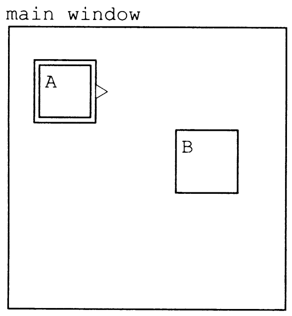
4. EXECUTION OF PIGSTY PROGRAM

I-PIGS makes heavy use of the multiple windows of the modern graphics workstation. During execution of a Pigsty program, tables of variables and a record of the execution history are displayed in different windows.

Before actual execution of a Pigsty program, the text in the chart program of each basic process is parsed first and virtual machine instructions are generated. If the user chooses to see the chart program of any process, it will be shown in the chart-program window. The text which has been scanned is converted to reverse video. Thus if a syntax error occurs, the user would know that it occurs at the last lexical token in reverse video.



(*a*) Group boxes *A1* and *A2* to become high-level box *A*.
In the main window, the corners of the marked region enclosing boxes *A1* and *A2* are shown; and the chosen interface port *Port1* is highlighted.
In the group-boxes window, interface port *Port1* is just placed on the new high-level box *A*.



(*b*) End of grouping boxes *A1* and *A2*.
The high-level box *A* is shown in the main window in place of the grouped boxes.

Figure 10. Group boxes.

I-PIGS checks that the port name used in a communication command of a process is that of a port icon on the boundary of the corresponding box. It also checks that the port has a link to a communication partner. The graphical system editor of I-PIGS has guaranteed that if a link exists, it must be between a pair of InPort and OutPort.

After all the processes have been parsed successfully, I-PIGS will execute the Pigsty program by interpreting the generated virtual machine instructions. Since I-PIGS runs in a uniprocessor workstation, it executes the concurrent Pigsty processes via a time-slicing mechanism to achieve simulated concurrency.

The icon of a process being executed is shown in reverse video. If the user chooses to see the chart program, I-PIGS will highlight the constructs being executed in reverse video in the chart-program window. This highlighting shows the control flow of the chart program.[3]

I-PIGS highlights ready-to-communicate port icons in reverse video and animates data communication between processes in the main window. For example, Fig. 11 (*a*) shows the appearance of the main window when the output command '*PortQ! data*' of process *Q* is being executed, where *data* has the value 1. The value to be sent is displayed in box *Q* and its OutPort icon *PortQ* is highlighted to indicate ready to communicate. If the communicating process *P* is already waiting for input data, the value 1 is sent immediately to *P* and displayed in box *P* (Fig. 11 (*b*)). If *P* is not yet ready to communicate, *Q* has to wait until I-PIGS executes the corresponding input command of *P*. At then, the value 1 is sent to *P* and displayed in box *P* (Fig. 11 (*b*)). After data communication, the displayed data values in the boxes are cleared. Thus, data communication is animated on the screen as the showing of the data value in the output box and then in the input box.

To help debugging, the user can require a breakpoint to occur before a chosen structured chart of a process is executed, or after a process becomes ready to communicate. At a breakpoint, I-PIGS allows the user to examine or change the variables of any process, to step-execute the stopped process, or to continue the execution.

5. AN EXAMPLE

Since I-PIGS highlights port icons waiting to communicate, whenever I-PIGS detects a deadlock, the situation is shown clearly. This is illustrated with an example – the Dining Philosophers Problem.

The problem is to simulate the behaviour of five philosophers who spend their lives thinking and eating spaghetti. They eat at a circular table in a dining room.

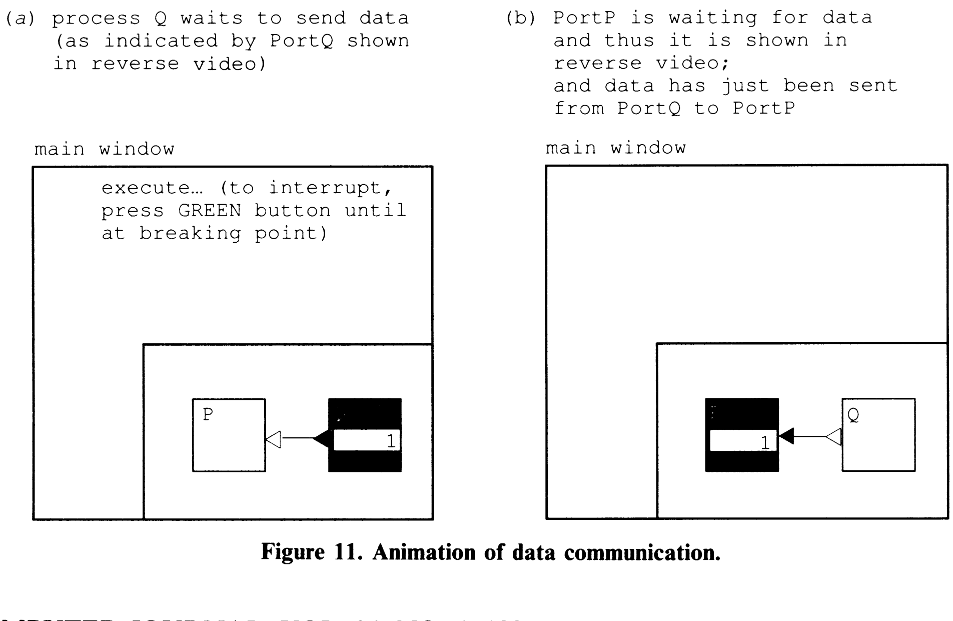
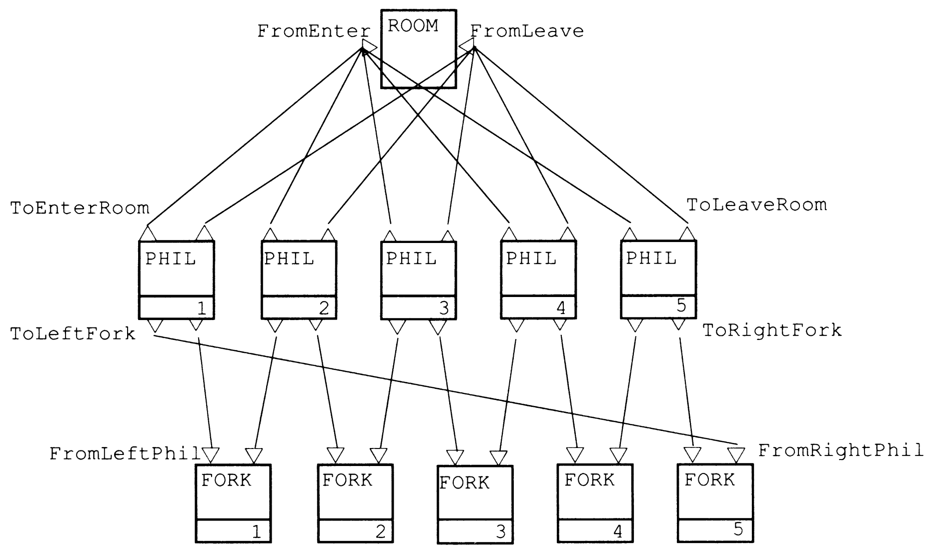

(a) process Q waits to send data (as indicated by PortQ shown in reverse video)
main window
execute... (to interrupt, press GREEN button until at breaking point)
P
1
(b) PortP is waiting for data and thus it is shown in reverse video; and data has just been sent from PortQ to PortP
main window
1
Q

Figure 11. Animation of data communication.

(*a*) system structure of processes



(*b*) chart program for process PHIL

```
PROCESS PHIL
COMMENT     following
            constants
            are used as
            signals
CONST    Enter = 1
CONST    Leave = 1
CONST    PickUp = 1
CONST    PutDown = 1
MAIN BODY
  WHILE      (* ALIVE *)
    (* THINK *)
    ToEnterRoom ! Enter
    ToLeftFork  ! PickUp
    ToRightFork ! PickUp
    (* EAT *)
    ToLeftFork  ! PutDown
    ToRightFork ! PutDown
    ToLeaveRoom ! Leave
```

(*c*) chart program for process ROOM

```
PROCESS ROOM
COMMENT   'Enter' & 'Leave'
          are used as signals
VAR    Enter, Leave : integer
VAR    occupancy, i : integer
MAIN BODY
  occupancy := 0
  *ALT
          FromEnter(i) ? Enter
    occupancy := occupancy +1
  /*ALT   FromLeave(i) ? Leave
    occupancy := occupancy -1
```

(*d*) chart program for process FORK

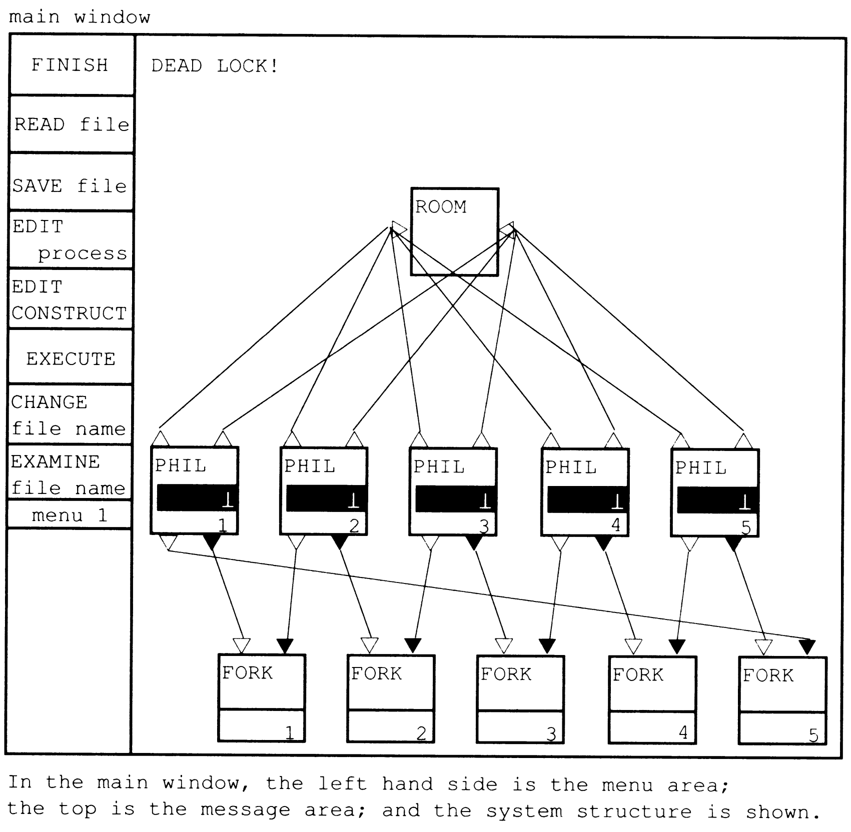
```
PROCESS FORK
COMMENT   'PickUp' & 'PutDown'
          are used as signals
VAR PickUp, PutDown : integer
MAIN BODY
  *ALT     FromLeftPhil ? PickUp
    FromLeftPhil ? PutDown
  /*ALT   FromRightPhil ? PickUp
    FromRightPhil ? PutDown
```

Figure 12. A Pigsty solution to the dining philosopher problem.

The table is surrounded by five chairs, each belonging to one philosopher, and five forks are laid circularly on the table in between the chair positions. When a philosopher feels hungry, he enters the dining room, sits in his own chair, picks up the fork on the left and then the fork on the right before he can eat. After eating, he puts down both forks and leaves the room.

Fig. 12 is a Pigsty program which implements a solution to the problem. Each *PHIL* process simulates a philosopher. Each *FORK* process simulates a fork. The *ROOM* process simulates the status of the dining room. This Pigsty solution is nearly identical to the CSP version.[9] The major difference is that port names instead of process names are used in the communication commands. (Note that the fork process icons, *FORK*, and the philosopher process icons, *PHIL*, were not laid out in a circle to avoid the mess up of the communication links over the box icons.)

The dining philosopher problem illustrates many of the problems encountered in concurrent programming, particularly resource sharing and deadlock. The forks are the shared resources. Controlled access to each fork is necessary. Each fork is simulated by a *FORK* process which ensures that the fork is picked up and then put



In the main window, the left hand side is the menu area; the top is the message area; and the system structure is shown.

Figure 13. Illustration of deadlock of the dining philosopher problem.

down by the same philosopher. Deadlock occurs if all five philosophers enter the room, each picks up his left fork and wishes to pick up the right fork, but cannot do so because it is in another philosopher's hand. The result is that all philosophers will be starved to death. Fig. 13 illustrates this deadlock situation: all ports *ToRightFork* are shown in reverse video indicating that all the philosophers wish to pick up the right forks, and all ports *FromRightPhil* are shown in reverse video indicating that each fork must first be put down by the philosopher to its right. Thus, through I-PIGS, the user can see what has happened and infer why it has happened.

6. CONCLUDING REMARKS

I-PIGS serves to support the idea that interactive graphical support for concurrent programming is feasible and effective. The graphical representation of a system of communicating processes and animation of data communication enables the user to understand the structure and the behaviour of a concurrent program more easily and more clearly.

Though Pigsty uses primitive message passing commands for interprocess communication, the idea of interactive graphical support for concurrent programming can also be used to support languages with other communication mechanism. For example, on a remote procedure call, the calling process can be highlighted to indicate waiting for results, and the sending of the invocation message and the reply of the results can be animated on the screen. To support languages which allow dynamic creation and deletion of processes, and which allow dynamic change of the links between processes, the display of the box icons and links can be changed accordingly during execution.

I-PIGS is just a small step towards employing interactive graphics to support concurrent programming. I-PIGS has not yet fully exploited the use of icons and animation. Since using appropriate (dynamic) graphics increases the communication bandwidth between human and human/computer, with the growing popularity of workstations, we expect that interactive graphical support for programming would become more effective and more widely acceptable.

Acknowledgement

I would like to thank Professor P. J. Brown of the University of Kent at Canterbury for his guidance and reading earlier versions of the paper, and Dr N. Ng of the University of Hong Kong who introduced to me the idea of interactive graphical support for programming. The comments by the referee helped much to improve the presentation of the paper. My research study was supported in part by a Hong Kong Li Po Chun Charitable Fund Postgraduate Scholarship in 1983–85 and in part by a U.K. Overseas Research Students Award in 1984–85.

REFERENCES

1. M. Pong, Interactive Graphical Support for Sequential and Concurrent Programming. Ph.D. thesis, University of Kent at Canterbury, U.K. (1985).
2. T. Teitelbaum and T. Reps, CPS – the Cornell Program Synthesizer. *Communications of ACM* **24** (9), 563–573 (1981).
3. M. Pong and N. Ng, PIGS – a system for programming with interactive graphical support. *Software – Practice and Experience* **13** (9), 847–856 (1983).
4. M. Pong, 2-PIGS: an interactive graphical programming environment with mixed interpretation and compilation. *Proc. Int. Computing Symposium 1985, Florence, Italy, March 1985*. North-Holland, Netherlands (1985).
5. E. P. Glinert and S. L. Tanimoto, Pict: an interactive graphical programming environment. *IEEE Computer* **17** (11), 7–25 (1984).
6. S. P. Reiss, Graphical program development with PECAN program development systems. *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Pittsburgh, U.S.A., April 1984*, as *ACM SIGPLAN NOTICES* **19** (5), 30–41 (1984).
7. S. P. Reiss, GARDEN tools: support for graphical programming. *Proc. Int. Workshop on Advanced Programming Environments, Trondheim, Norway, June 1986*, 63–80 (1986).
8. C. E. McDowell and D. P. Helmbold, Debugging concurrent programs. *ACM Computing Surveys* **21** (4), 593–622 (1990).
9. C. A. R. Hoare, Communicating sequential processes. *Communications of ACM* **21** (8), 666–677 (1978).
10. K. Jensen and N. Wirth, *Pascal – user manual and report.* Springer-Verlag, Berlin (1974).
11. I. Nassi and B. Shneiderman, Flowchart techniques for structured programming. *ACM SIGPLAN NOTICES* **8** (8), 12–26 (1973).
12. E. W. Dijkstra, Guarded commands, nondeterminancy, and formal derivation of programs. *Communications of ACM* **18** (8), 453–457 (1975).
13. Inmos Limited, *Occam Programming Manual.* Prentice Hall International, London (1984).
14. T. W. Mao and R. T. Yeh, Communication port: a language concept for concurrent programming. *IEEE Tran. Software Engineering* **SE-6** (2), 194–204 (1980).
15. J. Foley and A. van Dam, *Fundamentals of Interactive Computer Graphics.* Addison-Wesley, Reading, MA, U.S.A (1982).
16. N. Wirth, What can we do about the unnecessary diversity of notation for syntactic definitions? *Communications of ACM* **20** (11), 822–823 (1977).

APPENDIX: SYNTAX OF GRAPHICAL PART OF PIGSTY

Note:

(1) The syntax is based on Extended Backus-Normal Form[16] where
(*symbols*) means grouping of *symbols*;
[*symbol*] means *symbol* is optional;
{*symbol*} means *symbol* is repeated for 0 or more times;
'*literal*' means the string *literal* is a terminal symbol; and a grammar rule is terminated by a period (.).
Extra meta-symbols are used to represent graphical relationships.

(2) A comment is preceded by ! and runs until the end of the line.

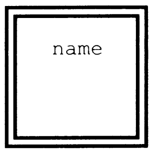
(3) The meta-symbol + means the placement of a port icon on the boundary of a process icon.

(4) The meta-symbol & in a rule '*Symbol1 & Symbol2*' means that *Symbol2* is associated with *Symbol1*. The association is not textual juxtaposition, but is supported by the programming environment.

The grammar rules for system structure:

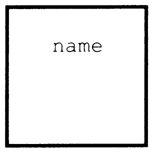
```
PigstyProgram=System.
System=CommunicatingProcess {{Link}
       CommunicatingProcess}.
CommunicatingProcess=
   (HighLevelProcess {+Port}) & System
 | (BasicProcess {+Port}) & ChartProgram.
 ! '& System' stands for the refinement
 ! subsystem of the HighLevelProcess.
 ! '& ChartProgram' stands for the
 ! coding of the BasicProcess in the
 ! form of a ChartProgram.
```

HighLevelProcess =

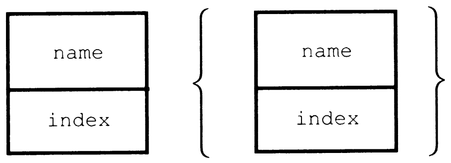


```
BasicProcess=SingleProcess
  | OneDimensionalArrayOfProcesses.

SingleProcess=
```



```
OneDimensionalArrayOfProcesses=
```



```
    ! A row or a column of element-process
    ! icons with index ranging from the lower
    ! bound to the upper bound of the array.
Port=OutPort & portName
  | InPort & portName.
    ! '& portName' stands for portName is
    ! the name of the port icon.

OutPort=SingleOutPort
  | OutPortOnElementOfProcessArray
  | ArrayOfOutPorts.
```

! The icon of all kinds of Outports is
! represented as a triangle with one
! of its sides on the side of a box.

```
InPort=SingleInPort
|  InPortOnElementOfProcessArray
|  ArrayOfInPorts.
```

! The icons of all kinds of InPorts is
! represented as a triangle with one
! of its vertex on the side of a box.

```
Link=OneToOneLink | OneToManyLinks |
      ManyToManyLinks.
```

```
OneToOneLink=(SingleOutPort|
           OutPortOnElementOfProcessArray)
    >-->(SingleInPort |
           InPortOnElementOfProcessArray).
```

! '>-->' stands for a straight line
! between an OutPort icon and
! an InPort icon.

```
OneToManyLinks=(ArrayOfOutPorts >-=>
   InPortOnElementOfProcessArray)
 | (OutPortOnElementOfProcessArray >=->
   ArrayOfInPorts).
```

! '>-=>' stands for fan-out straight
! lines from an `ArrayOfOutPorts` icon to
! `InPortOnElementOfProcessArray` icons.
! '>=->' stands for fan-in straight lines
! from `OutPortOnElementOfProcessArray`
! icons to an `ArrayOfInPorts` icon.

```
ManyToManyLinks =
     OutPortOnElementOfProcessArray>==>
     InPortOnElementOfProcessArray.
```

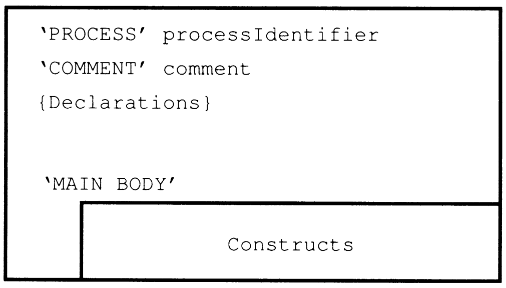
! '>==>' stands for straight lines between
! individual pairs of OutPort and InPort
! icons of two arrays of processes.

The grammar rules for chart program:

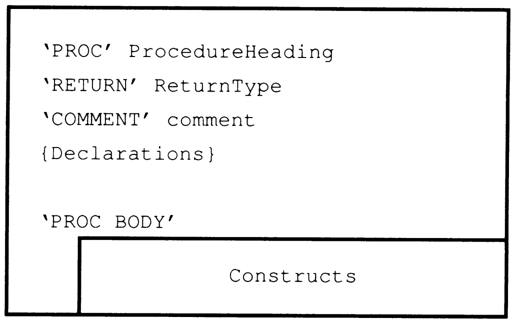
! The meta-symbol // in a rule '*Symbol1 // Symbol2*'
! stands for the vertical concatenation of *Symbol2*
! below *Symbol1*.

```
ChartProgram=ProcessBlock
             [//Procedures].

Procedures=ProcedureBlock
           {//ProcedureBlock}.
ProcessBlock=
```

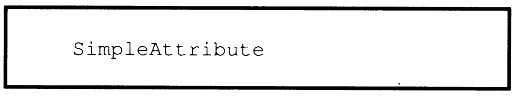


```
ProcedureBlock=
```



```
Constructs=ControlConstruct
          {//ControlConstruct}.
ControlConstruct=DoNothing
          | SimpleBlock
          | CompoundBlock.
DoNothing='(* DO NOTHING *)'.

SimpleBlock=
```



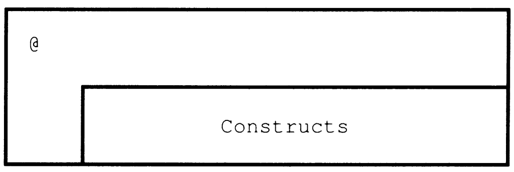
```
SimpleAttribute=TextualStatementList
          | 'EXIT_LOOP' comment
          | 'RETURN' comment.
```

! In the following context-sensitive grammar rules,
! the meta-symbol @ is used as position indicator
! in the structured chart; and
! the meta-symbol ˜ is used to force two rules to
! hold simultaneously.

! The part of the rule '*Chart@Attribute*'
! means *Attribute* is placed inside *Chart*
! at the position indicated by @.
! The rule '*X˜Y=(x)˜(y).*' means the two rules
! '*X=x.*' and '*Y=y.*' hold simultaneously.

```
CompoundBlock  =
   NestedBlock@Single Attribute
 | NestedBlock@HeadAttribute
   {// NestedBlock@NextAttribute}.

NestedBlock@=
```



```
SingleAttribute= 'WHILE'
  TextualCondition.
HeadAttribute˜NextAttribute=
  ('IF' TextualCondition)˜('/IF'
   TextualCondition)
 | ('ALT' TextualGuard)˜('/ALT'
   TextualGuard)
 | ('*ALT' TextualGuard)˜('/*ALT'
   TextualGuard).
```