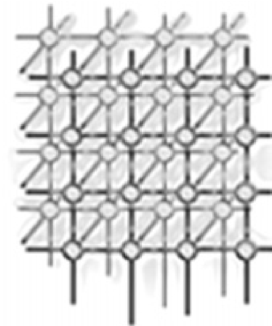
CONCURRENCY: PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2000; **12**:281–288

Concurrent programming in VISO



Muhammed S. Al-Mulhem*

Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia

SUMMARY

Concurrent programming is more difficult to use and understand than sequential programming. In order to simplify this type of programming a number of approaches have been developed such as visual programming. Visual Occam (VISO) is a visual programming language for concurrent programming. It has a graphical syntax based on the language Occam and its semantics is represented both in petri net and process calculus. This paper presents a modular visual approach to write concurrent programs using the VISO language. Concurrent programs in VISO are specified graphically at different levels of abstraction. This paper describes this modular visual approach by constructing two examples in VISO. The first example is a simple concurrent program and it is mainly used to show the details of constructing a concurrent program in VISO. The second example is a larger concurrent program with more levels of abstraction. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: concurrent programming; Occam; visual programming languages

1. INTRODUCTION

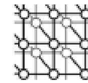
Concurrent programs are far more difficult to write, understand and debug than sequential programs. Visual programming research in this direction has been successful but it still needs a lot more work. There have been various attempts to develop visual languages for concurrent programming.

I-Pigs [1] is an interactive graphical environment for concurrent programming. The main advantage of I-Pigs is that it helps the users to visualize and understand their concurrent program. Its design is based on CSP and Pascal. I-Pigs uses a mixture of graphics and text to represent a program and it provides structured charts for writing the textual part. It uses a process icon but it has no icons for parallel and programming constructs. Ports connect one process to another if they communicate, but this results in cluttering of the screen representing the system structure. Recently system structuring aspects have been explored in [2–4]. Despite being a small system with its disadvantages I-Pigs

*Correspondence to: Muhammed Al-Mulhem, Information and Computer Science Department, King Fahd University of Petroleum and Minerals, Dhahran, Saudi Arabia.

Received 5 March 2000

Copyright © 2000 John Wiley & Sons, Ltd.



supports the idea that interactive graphical support for concurrent programming is feasible and effective.

PFG (parallel flow graphs) [5–8] is a language for expression of concurrent and time-dependent computations. The syntax is graphical and hierarchical but loses visual aid for realistically sized programs. The operational semantics of PFG are provided by the HG (hierarchical graph) model of concurrent time-dependent systems. PFG semantics are basically only for the shared memory model. PFG papers did not define a complete concurrent language and did not mention any complete examples. Nevertheless the underlying petri net basis of PFG allows the solution of several common problems in concurrent systems, such as deadlock detection and automatic mutual exclusion on shared data. It provides icons for some parallel constructs.

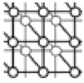
CODE and HeNCE [9] are visual programming languages that use annotated graphs to represent programs using textual annotations. Computations are defined by calls to sequential subprograms in textual languages such as C, and represented by the graph nodes. The meaning of the graph nodes in both languages is similar but the graph representation of programs is different. A program in CODE is a dataflow graph which shows that a data item is created by one sequential computation and used by another. A node may fire when all of its firing rules are satisfied. These rules are defined by the user as part of the textual annotations. A program in HeNCE is a graph that shows the control flow and not the dataflow. The HeNCE language uses a shared name model. In this model two or more nodes can access the same variable, and access to variables is controlled by the textual annotations. A node may fire when all of its predecessor nodes are fired.

Khoros [10] is a visual language that also uses graphs to represent programs. A program in Khoros is a dataflow graph where the nodes are represented by icons. The user selects these icons and connects them forming a dataflow graph that is displayed within a workspace.

AVS [11] is a data visualization tool that transforms massive and complex quantities of data into visual representations. It consists of an integrated set of data visualization and analysis tools. These tools include both traditional visualization tools such as 2D plots and advanced image processing tools such as 3D interactive rendering and volume visualization.

VISO [12–14] is a visual programming language for concurrent programming. It uses *message passing* for interprocess communication and processes in VISO are *disjoint*; meaning that they have no shared data. VISO syntax is graphical and based on the language Occam, and its semantics is represented in both petri net [12,13] and process calculus [14]. It uses the modular visual approach of visual programming to construct concurrent programs. Writing a program in VISO is done at different levels of abstractions. There are three levels of abstractions: system, processes, and statements levels. Each level is specified in a separate window, namely **SYSTEM** window, **PROCESS** window, and **statement** window. The **SYSTEM** window is used to specify the program to be written by specifying graphically the processes of the program and the channels connecting these processes. The **PROCESS** window is used to graphically specify the statements defining the body of the process and each process is defined in a separate window by itself. Finally, the details of the statements that appear in the body of each process are graphically specified in a **statement** window. There is one type of **statement** window with different labels corresponding to different statements in the program, such as **WHILE** window, **IF** window and **PAR** window. For example, the statements defining the body of a *while* statement are defined graphically in a **WHILE** window. Section 2 shows the details of constructing two concurrent programs in VISO.

Copyright © 2000 John Wiley & Sons, Ltd.



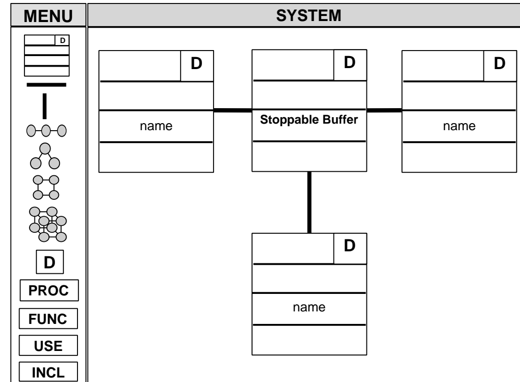


Figure 1. SYSTEM window: Stoppable buffer process with other processes.

2. CONSTRUCTION OF PROGRAMS IN VISO

This section presents two concurrent programs and shows how to construct them using the VISO system. The first program is a simple implementation of a *stoppable buffer* process. The main purpose of this example is to show in detail the steps needed to construct a concurrent program in the VISO system. The second example is a complete program that consists of two main processes. The first process is an input/output process and the second one is an array of identical processes to sort a set of characters. The next two sections show the details of these two examples.

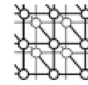
2.1. Example 1: Stoppable buffer

This section presents a simple concurrent program and shows how to construct it in the VISO system. The program consists of one process that communicates with three other processes through three external channels. It simply reads from a channel called **in** and writes whatever read to another channel called **out**, until anything is received on a channel called **stop**. This program implements a *stoppable buffer* process.

Assume that the *stoppable buffer* process is part of a larger system with four processes. Also, assume that the user has already defined the three other processes in the **SYSTEM** window. The user now selects a process icon (box with three horizontal lines inside it) from the **MENU** of the **SYSTEM** window, places it in the working area and names it *Stoppable Buffer*. Then the user selects a channel icon (vertical or horizontal line) from the **MENU** and connects the new process icon with the other three processes as shown in Figure 1.

A channel icon is used to illustrate two interacting processes; the details of communication should be specified explicitly. The user can do this by double clicking on the channel icon and specifying whether the channel is an **in** channel or **out** channel and specifying how many channels are there (in VISO a channel provides a one-way communication).

Copyright © 2000 John Wiley & Sons, Ltd.



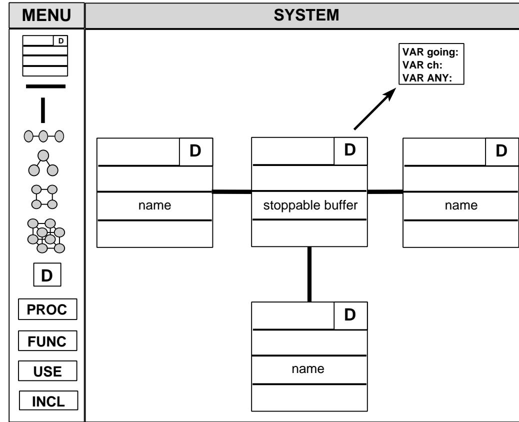


Figure 2. SYSTEM window: making declarations.

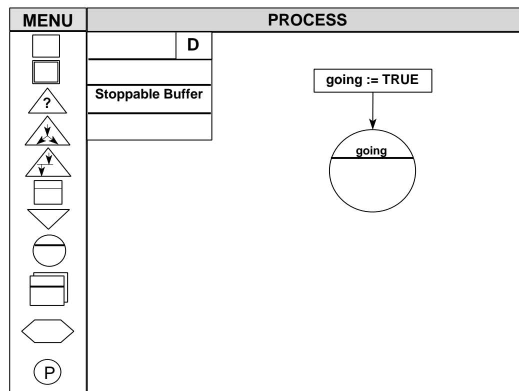


Figure 3. PROCESS window: defining the statements of the Stoppable Buffer process.

To make variables declaration, the user double clicks on the **D** icon associated with the process icon and types the declarations as shown in Figure 2. Note that in Figure 2 a thin dark arrow is used to indicate the contents of the **D** icon as a response to double clicking that icon. In order to define the actions of the *Stoppable Buffer* process, the user double clicks on that particular process icon to get the **PROCESS** window where he can graphically specify the statements describing the actions of the process. The *Stoppable Buffer* process consists of an *assignment* statement and a *while* statement. The user selects an *assignment* statement icon (box) from the **MENU** and places it in the working area of the **PROCESS** window as shown in Figure 3. Similarly, he selects a *while* statement icon (circle with a line

Copyright © 2000 John Wiley & Sons, Ltd.

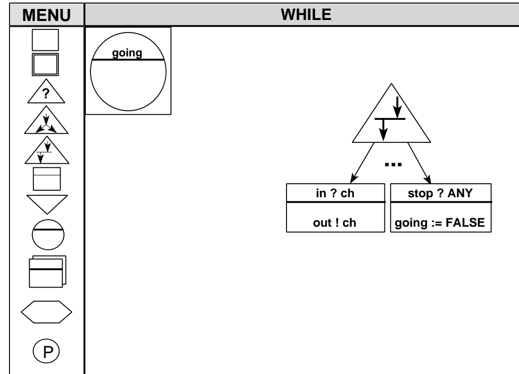


Figure 4. WHILE window: defining the body of the while statement.

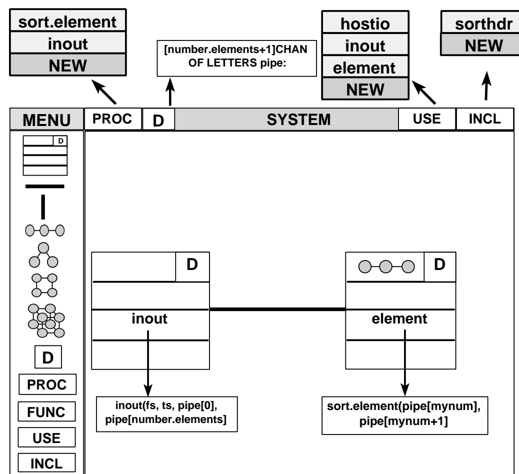
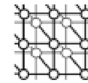


Figure 5. Pipeline sorter example.

inside) and defines its condition as shown in Figure 3. In order to define the body of the *while* statement the user double clicks on the body part (lower part) of the *while* icon to get the **WHILE** window. Then the user defines the statements of the body of the *while* statement by selecting the appropriate icons from the **MENU** of the **WHILE** window and places them in the working area as shown in Figure 4. Note that the defined icon is always displayed in the upper left corner of the corresponding window.

This completes our example of program construction. As we have seen the program is constructed in three levels of abstraction. The first level defines the processes in the **SYSTEM** window, the second

Copyright © 2000 John Wiley & Sons, Ltd.



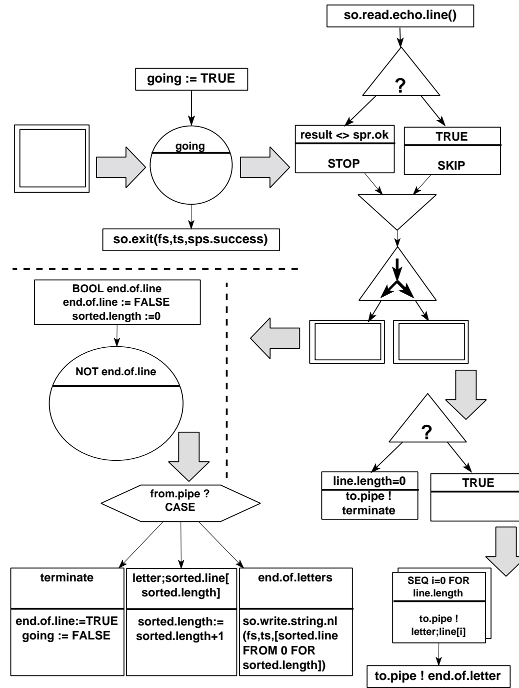


Figure 6. The definition of the *inout* process.

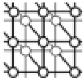
defines the details of the *Stoppable Buffer* process in the **PROCESS** window, and the third defines the body of the *while* statement in the **WHILE** window. We feel that this type of modular visual approach is essential as a visual aid for any visual programming, and in particular concurrent visual programming. Attempts to make a complete visual program on one screen results in a cluttered screen and thus destroys the usefulness and effectiveness of the visual programming approach.

2.2. Example 2: Pipeline sorter

Now, a program called pipeline sorter [15] is presented to show how a complete program might be constructed in terms of separate compilation units, libraries, and shared protocols. In this example thin dark arrows are used to indicate the contents of an icon as a response to double clicking that icon. Also, instead of showing all windows corresponding to the different programming constructs with their contents we show only the contents separated by thick arrows. It should be emphasized that both of these symbols (thin and thick arrows) are not part of the VISO language.

The complete example consists of two processes: an input/output process called *inout* and an array of processes called *element*. The *inout* process reads characters from the keyboard, passes the read characters to the *element* process to sort them, and writes the sorted characters to the screen. The

Copyright © 2000 John Wiley & Sons, Ltd.



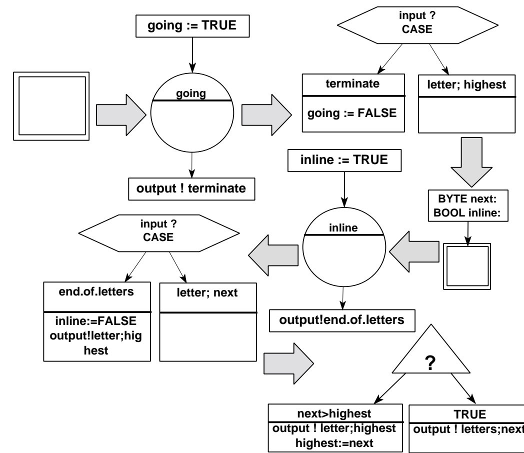


Figure 7. The definition of the *element* process.

system structure along with other details are shown in Figure 5. The definition of the *inout* process is shown in Figure 6. Note that in Figure 6 thick arrows are used to separate the contents of seven different windows defining the program. This is done mainly to minimize the number of pages of this paper. Similarly, the definition of the *element* process is shown in Figure 7.

This example shows that a fully fledged application can be constructed using the modular visual approach of the VISO language. This application is specified using a number of abstraction levels and each level is defined visually in a separate window by itself. Specifying a program over a number of windows representing different levels of abstraction is much easier and comprehensible than specifying the whole program in a single window with many cluttered graphical symbols.

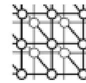
3. CONCLUSION

Constructing concurrent programs is a difficult task. This paper discusses the construction of concurrent programs in the visual language VISO. This language uses a **modular visual approach** to concurrent programming where programs are constructed at different levels of abstraction. The first level is always used to specify the processes in the program. This is followed by a number of levels to specify more details of the program. This approach should simplify the process of constructing concurrent programs and avoid specifying programs in one cluttered screen with many graphical symbols.

ACKNOWLEDGEMENT

I would like to thank KFUPM for providing the computing facilities to implement the VISO system.

Copyright © 2000 John Wiley & Sons, Ltd.



REFERENCES

1. Pong M. I-pigs: an interactive graphical environment for concurrent programming. *Computer Journal* 1991; **34**(4):320–330.
2. Kramer J, Magee J, Ng K. Graphical configuration programming. *Computer* 1989; **22**(10):53–65.
3. Magee J, Dulay N, Kramer J. Structuring parallel and distributed programs. *Software Engineering Journal* 1993; **8**(2):73–82.
4. Barbacci M, Weinstock C, Doubleday D, Gardner M, Lichota R. Durra: a structure description language for developing distributed applications. *Software Engineering Journal* 1993; **8**(2):83–94.
5. Stotts PD. Expressing high-level visual concurrency structures in the pfg kernel language. *IEEE 1988 Workshop on Visual Languages*, 1988; 168–174.
6. Stotts PD. The PFG environment: Parallel programming with petri net semantics. *Twenty-First Annual Hawaii International Conference on System Sciences*, 1988; 630–638.
7. Stotts PD. The PFG language: Visual programming for concurrent computation. *International Conference on Parallel Processing*, 1988; 72–79.
8. Stotts PD. Graphical operational semantics for visual parallel programming. *Visual Languages and Visual Programming*, Chang S-K (ed.). Plenum: New York, USA, 1990; 119–142.
9. Newton P. Visual programming and parallel computing. *Workshop on Enviroments and Tools for Parallel Scientific Computing*, May 26–27 1994.
10. http://www.khoros.com/khoros/khoros2/visprog.html.
11. http://www.avs.com/products/index.htm.
12. Al-Mulhem M, Ali S. Visual occam with petri net semantics. *Seventh International Conference on Parallel and Distibuted Computing Systems*, 1994; 746–754.
13. Al-Mulhem M, Ali S. Visual occam: syntax and semantics. *Computer Languages* 1997; **23**(1):1–24.
14. Al-Mulhem M, Ali S. Formal semantics of visual occam. *Computer Languages* 1998; **24**(2):99–113.
15. *A Tutorial Introduction to Occam Programming: Occam 2 Toolset User Manual*. BSP Professional Books: Oxford, England, 1989.

Copyright © 2000 John Wiley & Sons, Ltd.