A Graphical Tool for the Visualization and Animation of Communicating Sequential Processes

Ali E. Abdallah

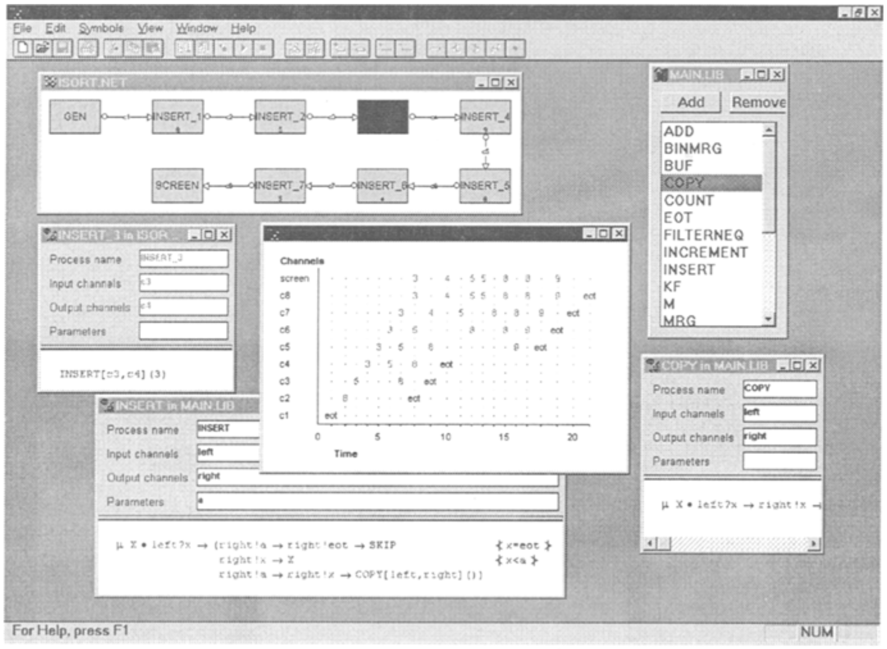
Department of Computer Science
The University of Reading
Reading, RG6 6AY, UK
email: A.Abdallah@reading.ac.uk
WWW:http://www.fmse.cs.reading.ac.uk/people/aea

Abstract. This paper describes some aspects of an interactive graphical tool designed to exhibit, through animation, the dynamic behaviour of parallel systems of communicating processes. The tool, called *VisualNets*, provides functionalities for visually creating graphical representations of processes, connecting them via channels, defining their behaviours in Hoare's CSP notation and animating the evolution of their visulization with time. The tool is very useful for understanding concurrency, analysing various aspects of distributed message-passing algorithms, detecting deadlocks, identifying computational bottlenecks, and estimating the performance of a class of parallel algorithms on a variety of MIMD parallel machines.

1 Introduction

The process of developing parallel programs is known to be much harder than that of developing sequential programs. It is also not as intuitive. This is especially the case when an explicit *message passing* model is used. The user usually takes full responsibility for identifying parallelism, decomposing the system into a collection of parallel tasks, arranging appropriate communications between the tasks, and mapping them onto physical processors. Essentially, a parallel system can be viewed as a collection of independent sequential subsystems (processes) which are interconnected through a number of common links (channels) and can communicate and interact by sending and receiving messages on those links. While understanding the behaviour of a sequential process in isolation is a relatively straightforward task, studying the effect of placing a number of such processes in parallel can be very complex indeed. The behaviour of all the processes becomes inter-dependent in ways which are not at first obvious making it very difficult to comprehend, reason about, and analyse the system as a whole. Athough part of this difficulty can be attributed to the introduction of new aspects such as synchronizations and communications, the main problem lies in the inherent complexity of *parallel control* as opposed to sequential control.

Visualizations are used to assist in absorbing large quantities of information very rapidly. Animations add to their worth by allowing visualizations to evolve with time and, hence, making apparent the dynamic behaviour of complex systems. *VisualNets* is an interactive graphical tool for facilitating the design and analysis of synchronous networks of communicating systems through visualization and animation.



INSERT_3 in ISOR...

Process name: INSERT_3
Input channels: c3
Output channels: c4
Parameters:

```
INSERT[c3,c4](3)
```

INSERT in MAIN.LIB

Process name: INSERT
Input channels: left
Output channels: right
Parameters: a

```
μ X • left?x → (right!a → right!eot → SKIP          { x=eot }
                right!x → X                          { x<a }
                right!a → right!x → COPY[left,right]())
```

COPY in MAIN.LIB

Process name: COPY
Input channels: left
Output channels: right
Parameters:

```
μ X • left?x → right!x →
```

Fig. 1. A screenshot showing the animation of an insertion sort algorithm

The tool, depicted in Fig. 1, allows the user to interactively create graphical representations of processes, link them via channels to form specific network configurations, describe the behaviour of each process in the network in Hoare's CSP notation [11], and study an animated simulation of the network in execution. The CSP syntax is checked before the animation starts. The visualization of the network animates communications as they take place by flashing the values being communicated over channel links. The tool builds a timing diagram to accompany the animation and allows a more in-depth analysis to be carried out. Networks can be built either as part of an orchestrated design process, by systematic refinement of functional specifications using techniques described in [1,2], or "on the fly" for rapid prototyping.

The benefits of using visualization as an aid for analysing certain aspects of parallel and distributed systems has long been recognised. The main focus in this area has been on tools for monitoring and visualizing the performance of parallel programs. These tools usually rely on trace data stored during the actual execution of a parallel program in order to create various performance displays. A good overview of relevant work in this area can be found in [10, 13]. A different approach aimed at understanding not only the performance of parallel and distributed systems but also their logical behaviours is based on simulation and animation for specific architectures [6, 5, 12].

The remainder of the paper is organised as follows. Section 2 introduces various features of the tool through a case study and Section 3 concludes the paper and indicates future directions.

2 A Case Study

We will give a brief description of the functionality of the tool through the development of a distributed sorting algorithm based on insertion sort. Given a non-empty list of values, the insertion sort algorithm, *isort*, starts from the empty list and constructs the final result by successively inserting each element of the input list at the correct position in an accumulated sorted list. Therefore, sorting a list, say $[a_1, a_2, ..a_n]$, can be visualized as going through n successive stages. The i^{th} intermediate stage, say *insert_i*, holds the value a_i, takes the sorted list *isort* $[a_1, a_2, ..a_{i-1}]$ as input from the preceeding stage and returns the the longer list *isort* $[a_1, a_2, ..a_i]$ to the following stage.

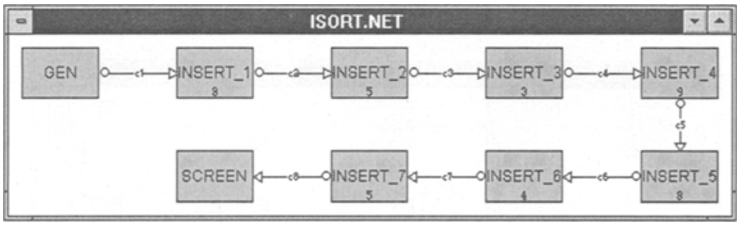
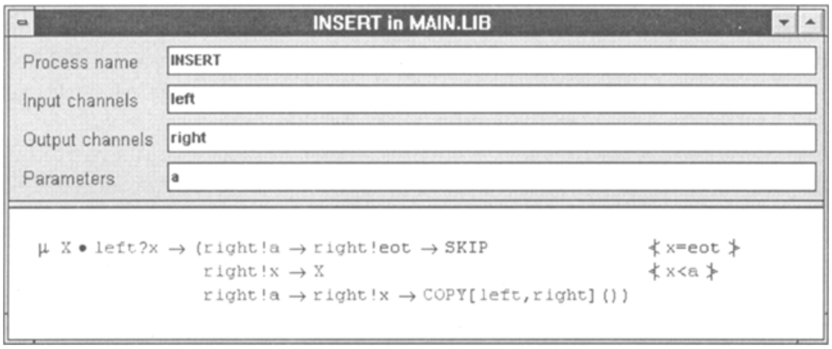


Fig. 2. Network configuration for parallel insertion sort

2.1 Creating a Graphical Network

The network configuration for sorting the list $[8, 5, 3, 9, 8, 4, 5]$ is depicted in Fig. 2. Each process in the network is represented graphically as a box with some useful information, such as values of local parameters and process name, inside it. Processes in the network can be linked via channels to form any topographical configuration. The construction of a network involves operations for adding, removing, and editing both processes and channels. The interface allows these operations to be carried out visually and interactively.


INSERT in MAIN.LIB

Process name	INSERT
Input channels	left
Output channels	right
Parameters	a

```
μ X • left?x → (right!a → right!eot → SKIP              ⊀ x=eot ⊁
                right!x → X                              ⊀ x<a ⊁
                right!a → right!x → COPY[left,right] ())
```

Fig. 3. A CSP definition of $INSERT(a)$

2.2 Defining Process Behaviour

Having defined the network configuration, the next stage is to define the behaviour of each process in the network. This is done using a subset of CSP. The screen shot captured in Fig. 3 shows the CSP definition for the process $NSERT(a)$ which is stored in a library of useful processes. The processes $SERT\_i$, $1 \leq i \leq 7$, depicted in the above sorting network are all defined as pecific instances of the library process $INSERT(a)$ by appropriately instantiating the value parameter a and renaming the channels *left* and *right*.

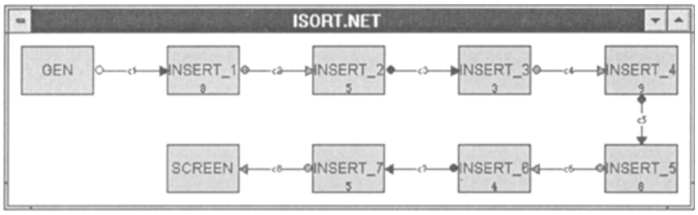


Fig. 4. Animation of the network

In CSP, outputing a specific value v on a channel c is denoted by the event $c!v$, inputing any value v on channel c and storing it in a local variable x is denoted by the event $c?x$. The arrow $\rightarrow$ denotes prefixing an event to a process. The notation $P \ \not< b \not> \ Q$, where b is a boolean expression and P and Q are processes, is just an infix form for the traditional selection construct **if** b **then** P **else** Q. Note that the prefix and conditional operators associate to the right. The special message *eot* is used to indicate the end of a stream of messages transmitted on a channel. The process $COPY$ denotes a one place buffer. For any lists s, the process $Prd(s)$ outputs the values of s in the same order on channel *right* and followed by the message *eot*. The process GEN is $Prd([])$.

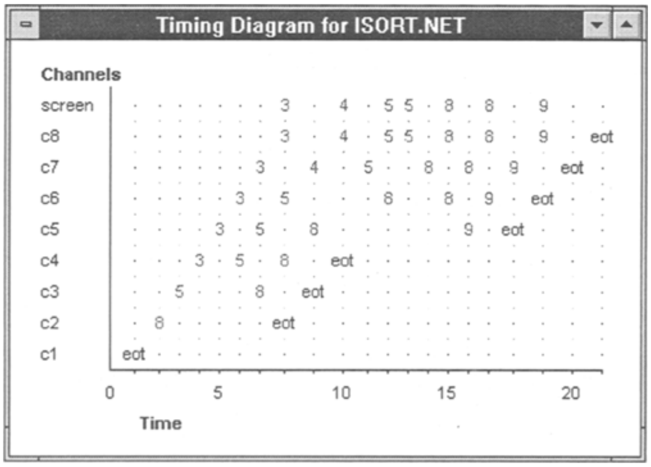


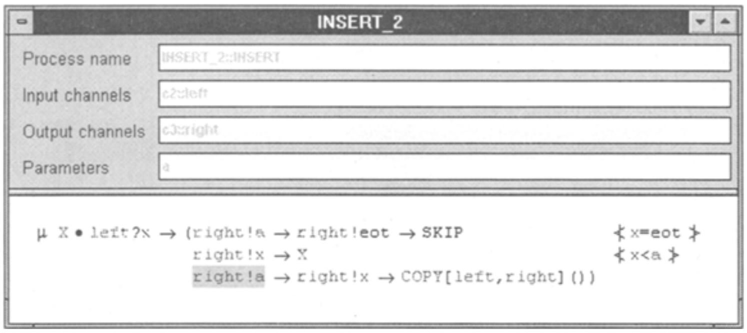
Fig. 5. Timing diagram for the network

2.3 Animation of the Network

Having created the network and defined each process, we can now graphically animate the execution of the network by selecting the **run** button for continuous animation, or the **step** button to show the new state of each process in the network after one time step. Channels in each process are colour coded to reflect the state of the process as follows: **red** for unable to communicate, **green** for ready to communicate, and **white** when the process has successfully terminated. When the indicators of a channel between two processes are both green, the communication takes place at the next time step.

Fig. 4 clearly illustrates that on the next time step, communications can only happend on channels with green (light grey) colour on both ends of the links. These are, channels c_2, c_4, c_6, and c_8. Fig. 5 illustrates the timing diagram for animating the network. It contains a record of all communications on each channel in the network coupled with the appropriate time stamp.

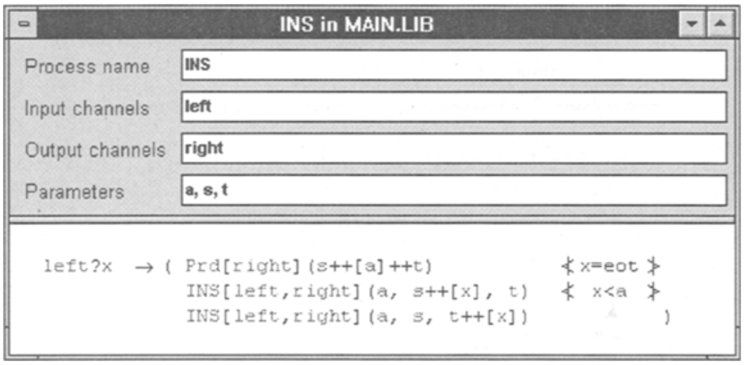
The evolution of each individual process in the network can be dynamically monitored during the animation of the network. An indicator highlights the exact place in the code of the process which will be executed at the next time step. For example, the cursor in Fig. 6 indicates that the process $INSERT\_2$ is willing to output the value 8 at the next time step.


INSERT_2

Process name	INSERT_2::INSERT
Input channels	c2::left
Output channels	c3::right
Parameters	a

```
μ X • left?x → (right!a → right!eot → SKIP         ⦃ x=eot ⦄
                right!x → X                          ⦃ x<a ⦄
                right!a → right!x → COPY[left,right]())
```

Fig. 6. Monitoring control within an individual process during network animation


INS in MAIN.LIB

Process name	INS
Input channels	left
Output channels	right
Parameters	a, s, t

```
left?x  → ( Prd[right](s++[a]++t)              ⦃ x=eot ⦄
            INS[left,right](a, s++[x], t)      ⦃  x<a  ⦄
            INS[left,right](a, s, t++[x])               )
```

Fig. 7. The process $INS(a, s, t)$.

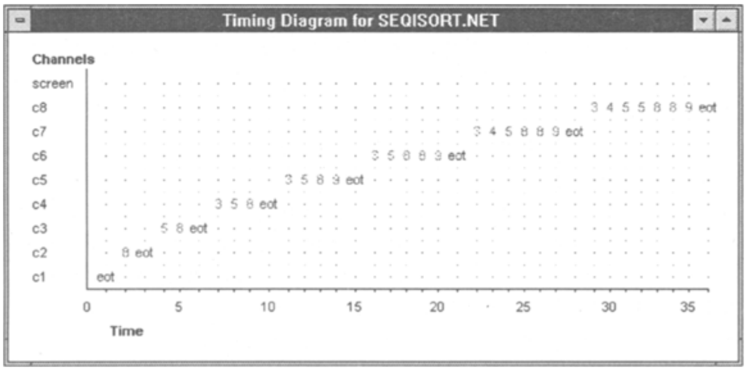


Fig. 8. Timing diagram for the execution of the new network.

2.4 Alternative Designs

The process $INSERT(a)$ is just a valid implementation of the function *insert*(a) which takes a sorted list of values and inserts the value a at the appropiate position so that the extended list is also sorted. Another process $INSERT'(a)$ which also correctly implements *insert*(a) can be defined as depicted in Fig. 7. In this definition, the additional variable s (and t resp.) is used to accumulate the list of input values which are stricly less than a (and greater or equal to a respectively).

$$INSERT'(a) = INS(a, [], [])$$

The timed diagram of the new network is shown in Fig. 8. The behaviour of the network is completely sequential. Parallelism is only syntactic; each stage in the pipeline needs to completely terminates before the following stage starts. One of the strengths of *VisualNets* is that it allows the user to alter the network being investigated very quickly, so that variations can be tested and compared interactively. After each change the user can immediately run the network again, view the animation and the timing diagram as in Fig. 8. until a good design is reached.

3 Conclusions and Future Works

In this paper we have presented a graphical tool for the visualization, simulation, and animation of systems of communicating sequential processes. A brief overview of the functionality of the tool is described through the process of developing a distributed solution to a specific problem. Such a tool is of a great educational value in assisting the understanding of concurrency and in illustrating many distributed computing problems and the techniques underlying their solutions. Perhaps the most important aspect of the tool is the ability to visually alter a design, experiment with ideas for overcoming specific problems, and investigate, through animations, the potential consequences of certain design decisions. The tool proved very useful in detecting, through animation, undesirable behaviours such as bottlenecks deadlock [3]. However, currently *VisualNets* cannot deal with non-deterministic processes our underlying visualisation techniques are not easily scalable.

Work is presently in progress on a new version of the tool, rewritten in Sun Microsystems' Java language. The tool is platform-independant and will operate on UNIX or Windows-based systems. The new tool implements a considerably larger set of CSP operators, and can deal with networks that synchronise on events as well as on channel input or output. Internal parallelism within processes is supported, permitting a smaller network to be visualised as a single process and later zoomed to display the detail of the internal communications. The emphasis of the new project is on developing an advanced visualisation and animation tool and integrating it within an environment which allows higher level of abstractions and capabilities for systematic specification refinement and program transformation.

Acknowledgements

The work reported in this paper has evolved over several years and is the result of working very closely with several final year and postgraduate Computer Science students at the University of Reading, UK. In particular, I would like to thank Aiden Devine, Robin Singleton (for the implementation of the current tool) and Mark Green for developing the new Java implementation.

References

1. A. E. Abdallah, Derivation of Parallel Algorithms from Functional Specifications to *CSP* Processes, in: Bernhard Möller, ed., *Mathematics of Program Construction*, LNCS **947**, (Springer Verlag, 1995) 67-96.
2. A. E. Abdallah, Synthesis of Massively Pipelined Algorithms for List Manipulation, in L. Boug, P. Fraigniaud, A. Mignotte, and Y. Robert, eds, *Proceedings of the European Conference on Parallel Processing, EuroPar'96*, LNCS **1024**, (Springer Verlag, 1996), pp 911-920.
3. A. E. Abdallah, Visualization and Animation of Communicating Processes, in N. Namazi and K. Matthews, eds, *Proc. of IASTED Int. Conference on Signal and Image Processing, SIP-96*, Orlando, Florida, USA. (IASTED/ACTA Press, 1996), 357-362.
4. R. S. Bird, and P. Wadler, *Introduction to Functional Programming*, (Prentice-Hall, 1988).
5. Rajive L. Bagrodia and Chien-Chung Shen, MIDAS: Integrated Design and Simulation of Distributed Systems, *IEEE Transactions On Software Engineering*, **17** (10), 1993, pp. 1042–1058.
6. Daniel Y. Chao and David T. Wang, An Interactive Tool for Design, Simulation, Verification and Synthesis of Protocols, *Software and Experience*, **24** (8), 1994, pp. 747-783.
7. Jim Davies, *Specification and Proof in Real-Time* CSP, (Cambridge University Press, 1993).
8. H. Diab and H. Tabbara, Performance Factors in Parallel Programs, Submitted for publication, 1996.
9. Michael T. Heath and Jennifer A. Etheridge, Visualizing the Performance of Parallel Programs, *IEEE Software*, **8** (5), 1991, pp. 29–39.
10. Michael T. Heath, Visualization of Parallel and Distributed Systems, in A. Zomaya (ed), *Parallel and Distributed Computing Handbook*, (McGraw-Hill, 1996)
11. C. A. R. Hoare, *Communicating Sequential Processes*. (Prentice-Hall, 1985).
12. T. Ludwig, M. Oberhuber, and R. Wismller, An Open Monitoring System for Parallel and Distributed Programs, in L. Boug, P. Fraigniaud, A. Mignotte, and Y. Robert, eds, *Proceedings of the European Conference on Parallel Processing, EuroPar'96*, LNCS **1123** (Springer, 1996) 78-83.
13. Guido Wirtz, A Visual Approach for Developing, Understanding and Analyzing Parallel Programs, in E.P. Glinert, editor, Proc. *Int. Symp. on Visual Programming*, (IEEE CS Press, 1993) 261-266.