# A Programming Environment for a Transputer-Based Multiprocessor System*

M. ASPNÄS and R. J. R. BACK

*Åbo Akademi University, Department of Computer Science*
*Lemminkäisenkatu 14, SF—20520 Turku, Finland*

## Abstract

This paper presents a transputer-based multiprocessor system, Hathi—2, and the programming environment being developed for this system. Hathi—2 is mainly programmed in the language Occam, and thus the programming environment is based on the Occam model of parallelism and communication. The programming environment gives the user an abstraction of the physical structure of the multiprocessor system. The user sees the multiprocessor system as a pool of resources (processors and communication links), which are allocated to the users program and connected to the topology described by the program structure. The environment is implemented on a Sun graphical workstation.

## 1. Introduction

This paper describes the design of a graphical programming environment for a transputerbased multiprocesor system. The programming environment consists of a number of program development tools integrated under a common graphical user interface.

The Hathi-2 multiprocessor system was designed and built in a joint project between the Department of Computer Science at Åbo Akademi and the Technical Research Center of Finland (VTT/TKO) in Oulu. As a part of the project, a number of application programs have been implemented on Hathi-2. The experiences gained from the applications show that more sophisticated program development tools are needed for multiprocessor systems of this kind. At present, programming multiprocessor systems is considered more difficult than programming sequential computer system. This is mainly due to the lack of programming tools available for use in the design and debugging of parallel programs.

* Lecture presented at the 1st Finnish-Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, August 8—11, 1989.

A parallel program for a MIMD-type multiprocessor system is normally designed as a number of independent sequential processes, which communicate with each other by sending and receiving messages through point-to-point commuication channels. When writing a parallel program, the logical process network is first designed. The logical process network describes the structure of the processes in the program and their interconnections through logical communication channels. The processes are written and tested separately, until the programmer is confident in their behaviour. After this, the programmer decides how these processes are placed on physical processors and executed in parallel. Two steps are required to do this: first, one must describe how the processes are placed on physical processors and what communication links connecting these are needed, and second, the multiprocessor system has to be connected (reconfigured) into this topology.

Both these steps involve a substantial amount of work for the programmer and introduce an additional source of errors. When the programmer has written a parallel program, he wants to experiment with different processor interconnection topologies and process placement schemes and make the program as well balanced and effective as possible. This is done by monitoring the execution of the parallel program and identifying the bottlenecks of the program. Information about the utilization of the physical resources used by the program during execution is gathered and presented to the user. The bottlenecks in a parallel program are usually caused by either overloaded physical communication channels or by processors which are allocated too much computation. In the ideal case, all physical resources have an evenly distributed utilization, and no bottlenecks exist in the program. To remove an identified bottleneck, the programmer has to change the logical process network, the placement of the logical processes on the physical processes or the interconnection structure of the physical processors. Often all these are changed simultaneously, and the programmer has to place the logical processes onto the physical structure again, and the design cycle is repeated.

To identify and remove logical errors in a parallel program, the programmer wants to observe the logical behaviour of the program during execution. In a parallel program, this can be done by using algorithm animation techniques, in which the program execution is presented to the user in a graphical way as an animation of the execution. Traditional methods for program debugging (traces, breakpoints etc.) can not generally be used, as there is no global control of the system.

Thus, the programming cycle for parallel programs consists of designing the logical process network and the processes, reconfiguring the pysical process network into a suitable topology, mapping the logical process structure onto the physical processor network, debuging and correcting logical programming errors and monitoring the execution of the program to identify bottlenecks, which often leads to changes in the logical program structure, and so the cycle is repeated.

At present, the programmer has to do all these steps manually. Clearly, some of these steps could be done automatically by a set of programming tools. The programming environment presented in this paper gives the user this type of utilities, by providing an integrated set of tools for mapping a process structure onto a physical processor network, monitoring the resource utilization of an executed program and animating the logical behaviour of a program. The presented programming environment provides the user with an abstract view of the multiprocessor system by hiding the physical interconnection structure of the system from the user.

The paper is organized as follows: the architecture of the Hathi—2 system is presented in Section 2. In Section 3 we give a short description of the programming language Occam. In Section 4 we describe the programming environment and finally, in Section 5 we describe the future developments of the presented programming environment.

## 2. The Hathi-2 Multiprocessor System

Hathi-2 is a reconfigurable general purpose multiprocessor system consisting of 100 32-bit IMS T800 transputers [Inm1], 25 16-bit IMS T212 transputers and 25 IMS C004 crossbar switches. The system can be characterized as a loosely coupled MIMD multiprocessor, with a reconfigurable distributed interconnection network and a modular design. A more detailed description of the Hathi-2 architecture and its use can be found in [AsBaMa], [AsMa] and [Peh]. The distributed switching network is described in [Äij].

Hathi-2 consists of 25 identical boards, each containing four T800 transputers, one T212 transputer and one 32 link crossbar switch. The T800 transputers are connected pairwise to each other via one of the four communication links. The three remaining links are connected to the crossbar switch (see Fig. 1). Three links from each switch are used as I/O links, i.e., to connect users host computers and peripheral units to the system. The remaining 16 links from the crossbar switch are used to form a statical torus connection between the boards in the Hathi-2 system, thus forming the distributed switching network.

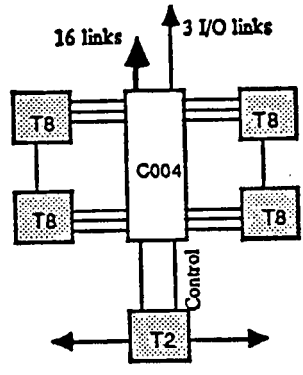


*Figure 1.* Hathi-2 board architecture

The C004 crossbar switch is controlled by the T212 transputer via a control link. One link on the T212 is connected to the crossbar switch and can be connected via the switch to any other transputer link. The two remaining links on the T212 (links 0 and 1) are used to connect the T212 transputers into a ring, thus forming the distributed control system.

The crossbar switches on the Hathi-2 boards are connected to each other in a static torus connection by connecting each pair of neighbouring boards to each

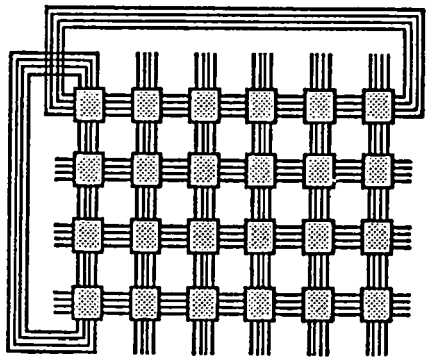

*Figure 2.* Hathi-2 board connections

other with four links (see Fig. 2). The crossbar switches form a distributed switching network connecting the communication links of the T800 transputers, which enables the system to be reconfigured by software.

Hathi-2 is used as a back-end computing resource. The user edits, compiles and links his programs on a host computer, i.e., a Sun workstation. The program can then be loaded on to the multiprocessor system and executed.

The Hathi-2 system can be shared between a number of simultaneous users by paritioning it into several smaller independent multiprocessor systems (see Fig. 3). All users are allocated a separate partition which is independent of all other partitions. A user has full control over his own partition, but can not interfere with other users.

The T212 transputers are connected to each other in a ring, thus forming a separate control system which controls the switching network (see Fig. 4). The control system is totally independent from the rest of the system. The only connection between the user and the control system is via a link connecting one T212 transputer to the

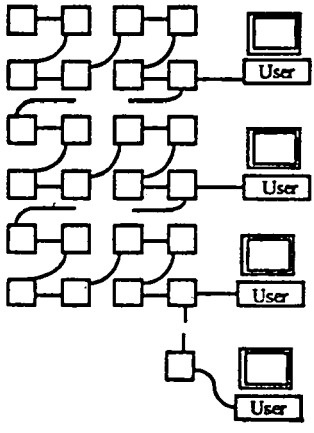


*Figure 3.* Partitioning the system

users host computer. The user can request system services by sending commands to the control system via this link.

The control system has two main tasks: to control the distributed switching network and to monitor the activities in the system. The Hathi-2 architecture contains hardware dedicated to monitoring the resource utilization in the system. The monitoring hardware consist of a CPU load meter which measures the CPU utilization by observing the bus activity and a FIFO buffer connecting all T800 transputers on a board to the controlling T212 transputer. The FIFO buffer can be used for sending reports about resource utilization from the T800 to the T212 without affecting the communication links.

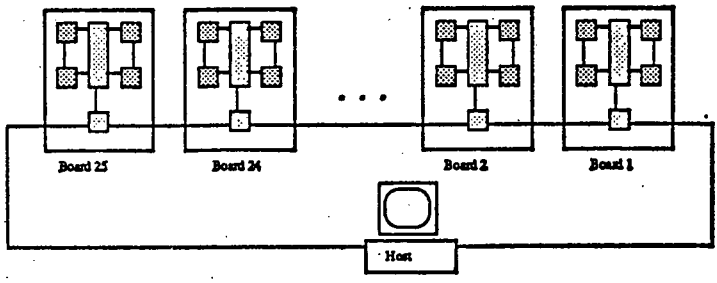


*Figure 4.* The control system

The control system also contains an interrupt subsystem implemented using the transputers EVENT interrupt. A processor in the control system can send an interrupt signal to all processors in the same partition. This interrupt is used in the monitoring system to generate a synchronizing signal which divides the time into short time intervals. The CPU and link utilization are measured for each interval and reported to the user.

## 3. The Occam programming language

Occam [Inm2], [JoGo] is a high-level programming language based on the CSP language [Hoa]. An Occam program consists of a number of sequential processes, which communicate with each other via unidirectional channels using synchronous message passing.

A channel connects two processes, of which one acts as a sender and the other as a receiver. A process sends a message $M$ via a channel $c$ with an output statement $c!M$, and the receiving process inputs a message from the channel to a local variable with an input statement $c?M$. A process can wait for input from a number of channels at the same time, using an **ALT** construct. The sending process can not choose between different communication alternatives, but commits itself to a communication when it executes an output statement. Communication is synchronous, i.e., the process which first executes a communication statement remains waiting until its communication partner executes a corresponding communication statement.

Parallelism is expressed in occam by the **PAR** construct, which specifies that

```
PAR
  SEQ
    X := 5
    c ! X
  SEQ
    c ? Y
    Y := Y*2
```

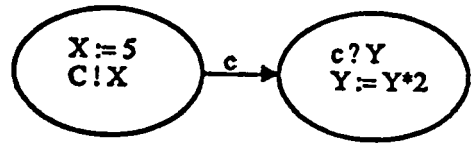


*Figure 5.* Communicating processes in Occam

two or more processes are executed in parallel. Sequential execution is specified with the **SEQ** construct. Scope is expressed in Occam by indentation. In the example in Figure 5, two processes communicate with each other via a channel *c*.

More than one process can be executed simultaneously on one transputer. The transputer divides its time between processes using a simple round-robin scheduler, which is built into the transputer hardware. Communication between processes executed on the same transputer is implemented through memory locations.

To execute a program with real parallelism on more than one transputers the programmer has to describe on which transputers the processes are to be executed and which communication links are used for communication between the processes. This is done by an Occam-like configuration language. The example in Figure 6 describes a ring of three processors, each executing a process Calculate. The processes communicate with each other by inputting from link 3 and sending on link 2. The user thus has to explicitly describe on which processor each process is executed and which communication links are used for communication between the processes. This means that the user has to have detailed knowledge about the hardware structure of the multiprocessor system.

```
CHAN OF INT C0, C1, C2:

-- SC PROC Calculate (CHAN OF INT From.previous, To.next)

PLACED PAR

  PROCESSOR 0 T8
  PLACE C0 AT 2 :-- link2out
  PLACE C2 AT 7 :-- link3in
  Calculate (C2, C0)

  PROCESSOR 1 T8
  PLACE C0 AT 7 :-- link3in
  PLACE C1 AT 2 :-- link2out
  Calculate (C0, C1)

  PROCESSOR 2 T8
  PLACE C1 AT 7 :-- link3in
  PLACE C2 AT 2 :-- link2out
  Calculate (C1, C2)
```

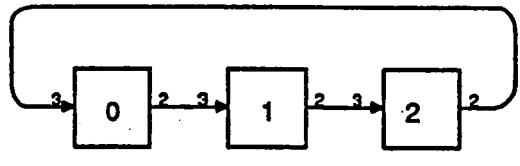


*Figure 6.* Placing processes on processors

## 4. The programming environment

The programming environment developed for the Hathi-2 multiprocessor system is designed by integrating a number of tools and utilities under a graphical user interface. The approach taken has been to use as much as possible of already existing software, i.e., editors, compilers, configurers, network loaders and debuggers. This is possible, because the Hathi-2 architecture is fully software compatible with Inmos transputer products.

The utilities that have been developed for Hathi-2 in the project are based on the specific hardware characteristics of the system and are not directly portable to other architectures. These tools include a utility wich allows the user to reconfigure the topology of the system, a monitoring utility which is used for monitoring the utilization of the resources of the system, and an animation tool which is used to visualize the execution of a parallel program.

The goal of this work is to make the multiprocessor system easier to use for the programmers by building a user-friendly graphical interface to the tools, and to hide the physical structure of the multiprocessor system from the programmer. The user should be able to construct a parallel program for the Hathi-2 system entirely within the programming environment. The whole cycle of editing a program, compiling, loading the program onto a number of processors and executing it, debugging the program and monitoring the performance of the program can be carried out within the programming environment.

### *4.1. The user interface*

The user interface of the programming environment is based on a hierarchical graph editor. The user describes the *process structure* of a distributed program by drawing a graphical representation of the processes and their interconnections. The graph representing a parallel program consists of a number of nodes and arcs, the nodes representing processes and the arcs representing communication channels between the processes. A node in the process graph is associated either with a subgraph or directly with the code of the process. The source code describing a process can be edited by selecting the node representing the process by clicking on it with the mouse. This will bring up the Occam folding editor, and the code of the process can be edited in the normal way.

The processes in the process graph are grouped together to form *tasks*. A task is a separately compiled unit of code (in Occam called a *SC*), which is executed on one processor and usually consists of a number of parallel communicating processes. The processes constituting a task are executed on one transputer using the transputers timeslicing scheduler. Thus, the process graph is condensed into a *task graph*, which determines the physical structure of the processor network on which the program is to be executed. In Figure 7 is an example of a process graph, which is condensed into a task graph using four processors connected into a pipeline. The physical communication links connecting processors are drawn with fat lines, and are always bidirectional (consisting of two unidirectional links).

The utilities in the programming environment use the information about the distributed program contained in the process graph, the source code of the processes and the grouping of the processes into tasks. The editor used is a stand-alone

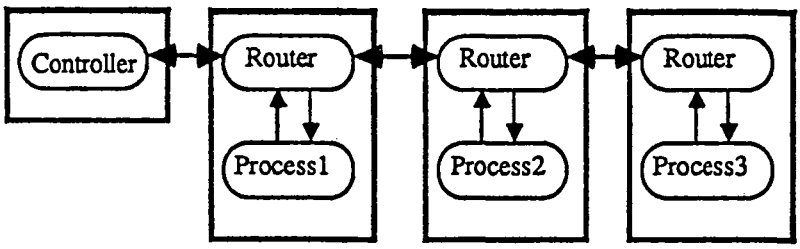


*Figure 7.* A process graph partitioned into a task graph

Occam folding editor. Similarly, the compiler, the configurer, the network loader and the debugger are the stand-alone Occam program development tools from Inmos. When the user invokes one of these tools by selecting an apropriate entry from a menu in the user interface or by clicking on a node in the process graph, this is translated to a corresponding Unix call which activates one of these utilities.

### *4.2. The mapping utility*

The mapping utility developed for Hathi-2 automatically maps a task graph onto the transputers in Hathi-2 and establishes the needed link connections between the transputers. The input from the user to the mapping utility consists of the task graph of the distributed program. As output, it generates the configuration statements needed by the Occam configurer to place this program structure onto a physical topology. The mapping utility also generates the commands needed by the reconfiguration software to connect the transputers into the topology described by the task graph.

The mapping utility makes it possible to hide the physical structure of Hathi-2 from the user. The user does not have to explicitly specify which of the four links on a transputer should be used for communication with other processors. This is a very useful feature when writing parallel programs, since the design of the configuration statements is considered to be difficult and very error-prone. The mapping of the processor graph onto the physical structure of Hathi-2 is done by a simulated annealing algorithm [Bok].

It is possible to find processor structures that cannot be mapped to the hardware structure of the multiprocessor system. First, not all graphs can be established on a transputer network, because a transputer has only four links. One example of this is a 5-dimensional hypercube, which requires a node degree of five. Second, the architecture of the distributed switching network in Hathi-2 imposes some limitations on which graphs can be established. The main limiting factor here is that there are only four links available between every pair of neighboring boards in the static torus interboard connection. Finally, the algorithm used in the mapping utility does not guarantee that a mapping of a graph to the structure of Hathi-2 is found. However, the mapping algorithm has proved to work well in practice for a large class of problems.

### *4.3. The monitoring utility*

The monitoring utility is used for monitoring the utilization of the resources in the multiprocessor system during program execution. It is used for finding bottlenecks in parallel programs executing on the system, and to provide information about the load balance of the programs. Monitoring is done by observing the CPU and link activity in the transputer network. The monitoring software is based on the monitoring hardware built into the Hathi-2 architecture, which makes it possible to monitor the system without introducing any substantial overhead on the main computation.

Monitoring data, i.e., data about CPU and link utilization on the transputers executing the monitored program, is gathered by the transputers in the control system. This data is sent through the control system to the users host computer, where it is stored in a file and presented to the user.

The time during which monitoring is done is divided into short time intervals (typically 100 to 500 milliseconds), and for each interval the monitoring system records the percentual utilization of the CPU, the number of bytes transmitted over a link and the time a process has spent waiting for a communication to take place.

The monitoring data is presented to the user as the average percentual utilization of the CPUs and links during an interval. The presentation is based on the task graph of the distributed program. For each transputer, its percentual CPU utilization is presented as a number written inside the node representing the transputer in the graph. Similarly, the percentual utilization of each link is written above the arc representing the link in the graph. In Figure 8 there is an example of how the results from a monitored program is presented to the user.

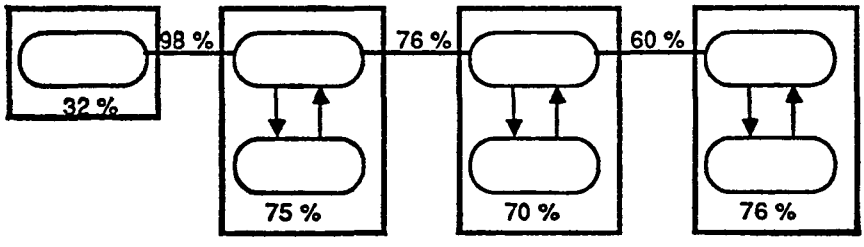


*Figure 8.* Presentation of monitoring data

The user can control the length of the time interval. Monitoring data is sampled with a fixed interval during program execution, but the data can be presented with any interval longer than this. If the user wants to see the result of the monitoring in an interval longer than the sampling interval, the mean values from a sufficient number of sampling intervals are calculated. The user can browse through the monitoring data both backwards and forwards in time. Normally, the first time a program is monitored, the user wants to view the result using a rather large timestep, to get an overall picture of the behaviour of the program. The user can later examine the execution of the program more closely, using a smaller time interval.

### *4.4. The animation utility*

A program animation is a graphical visualization of the execution of a program [Sol]. Program animation is used as a high-level debugging tool, which gives the programmer an understanding of how his program behaves during execution. This is especially important for parallel programs, as it is very difficult to get a picture of the overall behaviour of a program executing on a large number of processors.

Animation of parallel programs on the Hathi-2 system is implemented using the same hardware features as the monitoring system, i.e. the FIFO buffers. In the animation system, the data sent from the transputers to the control system contain information that controls the graphical animation of the executed program. This data is interpreted as graphical commands, which are executed by an animation process and which result in a graphical illustration of the program execution.

The animation is done by inserting commands into the animated processes, which send messages about their present state of execution to the animation process. The animation process receives these messages and translates them into graphical commands that update the screen. The user has to specify on which points in the execution the state of the program should be reported. The user also has to describe how each state should be represented in the animated picture. This is done by a graphical tool, which allows the user to draw the pictures of which the animation consists.

The execution of the animated program must also be slowed down, so that the user has time to register the updates on the screen. The program is slowed down uniformly, without affecting the logical behaviour of the program. This is implemented using the synchronization mechanism in Hathi-2. All processes are forced to wait for a synchronization signal, which is sent from the control system.

## 5. Conclusions and future work

This report describes work in progress in the Millipede project at Åbo Akademi. The tools that have been described have already been implemented and are now beeing separately tested and evaluated. The design of the graphical user interface which integrates the tools into a programming environment has recently started and is scheduled to be ready in the last quarter this year. After that, the environment will be evaluated and any possible further improvements and features will be considered.

Several components of the environment will be developed further. The routing algorithm used by the reconfiguration software for establishing link connections between processors through the distributed switching network will be developed [Shen1]. Also the mapping algorithm which is used for mapping a task graph to a physical configuration of Hath-2 will be improved by investigating different types of heuristic algorithms [Shen2], [Shen3]. Finally, the graphical user interface will be developed, based on the experiences of the users. The goal is to make the environment as simple as possible to use for the programmers.

## Acknowledgements

The Hathi-2 multiprocessor system was designed and built in the Hathi project, which was financed by TEKES, The Academy of Finland, Åbo Akademi and VTT. Part of the work has been done in the FINSOFT III research program, financed by TEKES. The authors also wish to thank Kaisa Sere for comments on the paper.

## References

[AsBaMa] M. ASPNÄS, R. J. R. BACK, T-E. MALÉN, The Hathi-2 Multiprocessor System, Reports on Computer Science, Ser. A, No. 80, Åbo Akademi, 1989.

[AsMa] M. ASPNÄS, T-E. MALÉN, Hathi-2 Users Guide, version 1.0, Reports on Computer Science, Ser. B, No. 6, Åbo Akademi, 1989.

[Ber] F. BERMAN, Experience with an Automatic Solution to the Mapping Problem, in The Characteristics of Parallel Algorithms, Jamisson, Gannon and Douglas (ed.), MIT Press, 1987.

[Bok] S. BOKHARI, On the Mapping Problem, IEEE Transactions on Computers, C-30, no. 3 (March, 1981), pp. 207—214.

[CrMa] P. CROL and G. MANSON, Configuration Tools for a Transputer Workstation, in Applying Transputer Based Parallel Machines, A. Bakkers (ed.), Proceedings of the 10th Occam User Group Tecnical Meeting, Enschede, Netherlands, IOS, 1989.

[EkMa] P. EKLUND, T-E. MALÉN, Block Placement in Switching Networks, Proc. CONPAR-88, Manchester, Great Britain, Cambridge University Press, 1988, pp. 289—295.

[EMMM] J. EUDES, F. MENNETEAU, L. MUGWANEZA and T. MUNTEAN, PDS: Advanced Program Development System for Transputer Based Machines, in Applying Transputer Based Parallel Machines, A. Bakkers (ed.), Proceedings of the 10th Occam User Group Tecnical Meeting, Enschede, Netherlands, IOS, 1989.

[Hoa] C. A. R. HOARE, Communicating Sequential Processes, Communications of the ACM, 21, 8 (Aug. 1978), pp. 666—677.

[Inm1] Inmos Limited, *Transputer Reference Manual*, Prentice-Hall, 1988.

[Inm2] Inmos Limited, *Occam 2 Refernce Manual*, Prentice-Hall, 1988.

[JoGo] G. JONES and M. GOLDSMITH, *Programming in Occam 2*, Prentice-Hall, 1988.

[Peh] K. PEHKONEN, A Dynamically Reconfigurable Parallel Computer Hathi-2, Licentiate thesis, University of Oulu, Department of Electrical Engineering, 1989.

[Shen1] H. SHEN, Fast Path-disjoint Routing in Transputer Networks, to appear in Proc. First Finnish—Hungarian Workshop on Programming Languages and Software Tools, Szeged, Hungary, 1989.

[Shen2] H. SHEN, Mapping Parallel Programs onto Transputer Networks, to appear in Proc. Australian Transputer and OCCAM User Group Conference, Melbourne, Australia, 1989.

[Shen3] H. SHEN, Self-adjusting Mapping: A Heuristic Mapping Algorithm for Mapping Parallel Programs onto Transputer Networks, to appear in Proc. 11th Occam User Group Technical Meeting, Edinburgh, Great Britain, 1989.

[Sol] U. SOLIN, Animation Techniques for Parallel Agorithms, Proc. International Conf. on Parallel Processing and Applications, 23—25. 9. 1987, L'Aquila, Italy.

[Äij] T. ÄIJÄNEN, Distributed Interconnection of a Reconfigurable Multicomputer System, Microprocessing and Microprogramming, 3—1988, pp. 243—246.