SUID-AFRIKAANSE REKENAAR SIMPOSIUM
SOUTH AFRICAN COMPUTER SYMPOSIUM

HOLIDAY INN PRETORIA
JULIE 1 – 3 JULY 1987
Proceedings
of the
4th South African Computer Symposium

Holiday Inn, Pretoria
1 - 3 July 1987

edited by

Pieter Kritsinger
Computer Science Department
University of Cape Town
PREFACE

Computer science is an emerging discipline which is having difficulty in being recognised as a worthy member of the sciences. I will paraphrase John Hopcroft, co-winner of the 1986 Turing Award, when, during a recent interview, he said that the primary reason for the lack of recognition, is the age of our researchers. Probably not one of the researchers who presented their work at this symposium is older than 45. I know of no computer scientist in South Africa who is in a position where (s)he can affect funding priorities. As far as I know we have no representation on any of the committees of the Foundation for Research Development and for our Afrikaans speaking fraternity, none who is a member of the Akademie vir Wetenskap en Kuns. It will take time and conscious effort to establish our presence. The same is true of course for our universities. Again, with one exception, I know of no dean of a science faculty, vice-principal or principal who is a computer scientist. We consequently spend an enormous amount of time trying to explain the needs of computer science and its difficulties. I believe this symposium is a further step towards accreditation by our peers and superiors from the other sciences.

The total number of papers submitted to the Programme Committee for consideration was 34. Each paper was reviewed by three persons knowledgeable in the field it represents. Of those submitted, 23 were finally selected for inclusion in the symposium. As a result the overall quality of the papers is high and as a computer science community in Africa we can be justly proud of the final programme.

This is the fourth in the series of South African computer symposia. This year the symposium is sponsored by the Computer Society of South Africa (CSSA), the South African Institute for Computer Scientists and the local IFIP Committee. The executive director of the CSSA and his staff deserve warm thanks for handling the organisation as well as they have, while the Organising Committee provided Derrick and I with very valuable advice.

Finally I would like to express my sincere appreciation to the authors, to the members of the Programme Committee and particularly the reviewers. Without the kind cooperation of everyone, this symposium would not have taken place.

Pieter Kritsinger
SYMPOSIUM CHAIRMAN: PS Kritzinger, University of Cape Town.
SYMPOSIUM CO-CHAIRMAN: D Kourie, University of Pretoria.
MEMBERS OF THE PROGRAMME COMMITTEE

Judy Bishop, Witwatersrand University
Chris Bornman, UNISA
Hannes de Beer, Potchefstroom University
Gideon de Kock, Port Elizabeth University
Jaap Kies, Western Cape University
Derrick Kourie, Pretoria University
Pieter Kritzinger, Cape Town University
Tony Krzesinski, Stellenbosch University
Michael Laidlaw, Durban Westville University
Peter Lay, Cape Town University
Ken MacGregor, Cape Town University
Theo McDonald, Orange Free State University
Jan Oosthuizen, University of the North
Dennis Riordan, Rhodes University
Alan Sartori-Angus, Natal University
John Shochot, Witwatersrand University
Theuns Smith, Rand Afrikaans University
Trevor Turton, ISM (Pty) Ltd
Gerrit Wiechers, Infoplan.
LIST OF REVIEWERS

BERMAN Sonia
BISHOP Judy
BORNMAN Chris
CAREY Chris
CHERENACK Paul
DE BEER Hannes
DE VILLIERS Pieter
GORRINGE Pen
KIES Jaap
KOURIE Derrick
KRITZINGER Pieter
KRZESINSKI Tony
LAIDLAW Michael
LAY Peter

MacGREGOR Ken
MATTISON Keith
McDONALD Theo
RENNHACKKAMP Martin
RIORDAN Denis
SATORI-ANGUS Alan
SCHOCCHOT John
SMITH Theuns
TURTON Trevor
VAN DEN HEEVER Roelf
VAN ROOYEN Hester
VON SOLMS Basie
VOS Koos
# TABLE OF CONTENTS

## Keynote Address

"An Extensible System and Programming Tool for Workstation Computers." ............................................. 1  
Niklaus Wirth, ETH, Zurich

## Invited Lectures

"The Relationship of Natural and Artificial Intelligence." ................... not included in Proceedings.  
G Lasker, University of Windsor, Ontario.

"Software Engineering: What Can We Expect in the Future?" ................ not included in Proceedings.  
D Teichrow, University of Michigan, U.S.A.

## Computer Languages I

"SPS-Algo·l: Semantic Constructs for a Persistent Programming Language." ................................. 13  
S Berman, University of Cape Town.

"Petri Net Topologies for a Specification Language." .... 25  
R Watson, University of the Witwatersrand.

"Towards a Programming Environment Standard in LISP." .......... 45  
R Mori, University of Cape Town

"ADA for Multiprocessors: Some Problems and Solutions." .... 63  
J Bishop, University of the Witwatersrand.

## Computer Graphics

"Polygon Shading on Vector Type Devices." ...................... 75  
C F Scheepers, CSIR.

"Hidden Surface Elimination in Raster Graphics Using Visigrams." .............................................. 97  
P Gorringe, CSIR.

## Database Systems I

"On Syntax and Semantics Related to Incomplete Information Databases." ................................. 109  
M E Orlowska, UNISA.

"Modelling Distributed Database Concurrency Control Overheads." ............................................. 131  
M H Rennhackkamp, University of Stellenbosch.

## Operating Systems

"The Development of a Fault Tolerant System for a Real-time Environment." ............................... 149  
M Morris, CSIR.

"A New General-purpose Operating System." .......................... 161  
B H Venter, CSIR.
Computer Languages II

"The Representation of Chemical Structures by Random Context Structure Grammars." .......................................................... 175
E M Ehlers and B von Solms, RAU.

"A Generalised Expression Structure." .............................. 189
W van Biljon, CSIR.

Computer Networks and Protocols I

"An Approximate Solution Method for Multiclass Queueing Networks with State Dependent Routing and Window Row Control." .......................................................... 203
A E Krzesinski, University of Stellenbosch.

"A Protocol Validation System." ........................................ 227
J Punt, University of Cape Town.

Computer Networks and Protocols II

"Protocol Performance Using Image Protocols." .................. 251
P S Kritzinger, University of Cape Town.

Artificial Intelligence

"A Data Structure for Exchanging Geographic Information." .......................................................... 267
A Cooper, CSIR.

"The Design and Use of a Prolog Trace Generator for CSP." .......................................................... 279
D G Kourie, University of Pretoria.

Database Systems II

"An Approach to Direct End-user Usage of Multiple Databases." .......................................................... 297
M J Phillips, CSIR.

"A Semantic Data Model Approach to Logical Data Independence." .......................................................... 329
S Berman, University of Cape Town.

Information Systems

"The ELSIM Language: an FSM-based Language for the ELSIM SEE." .......................................................... 343
L du Plessis and C Bornman, UNISA.

"Three Packaging Rules for Information System Design." .......................................................... 363
J Mende, University of the Witwatersrand.
Computer Languages III

"Experience with a Pattern-matching Code Generator." ... 371
M A Mulders, D A Sewry and W R van Biljon, CSIR.

"Set-oriented Functional Style of Programming." ... 385
C Mueller, University of the Witwatersrand.

Tutorial

The use of Modula-2 in Software Engineering." ... 399
N Wirth, ETH, Zurich.
### DAY 1

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>07h30</td>
<td>Registration and Coffee.</td>
</tr>
<tr>
<td>08h45</td>
<td>Welcoming address, President of the South African Institute of Computer Scientists, Dr. G. Wiechers.</td>
</tr>
<tr>
<td>09h00</td>
<td>Invited Lecture. Professor D. Teichrow, University of Michigan.</td>
</tr>
<tr>
<td></td>
<td><em>Software Engineering, ... What Can We Expect in the Future.</em></td>
</tr>
<tr>
<td>10h00</td>
<td>Coffee</td>
</tr>
<tr>
<td>10h15</td>
<td>Computer Languages I. Chairman: G. Wiechers.</td>
</tr>
<tr>
<td>10h50</td>
<td>S. Berman, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><em>SPS-Algol: Semantic Constructs for a Persistent Programming Language.</em></td>
</tr>
<tr>
<td>10h50</td>
<td>A. Watson, University of the Witwatersrand.</td>
</tr>
<tr>
<td></td>
<td><em>Petri Net Topologies for a Specification Language.</em></td>
</tr>
<tr>
<td>11h25</td>
<td>R. Mori, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><em>Towards a Programming Environment Standard in USP.</em></td>
</tr>
<tr>
<td>11h50</td>
<td>J. Bishop, University of the Witwatersrand.</td>
</tr>
<tr>
<td></td>
<td><em>ADA for Multiprocessors: Some Problems and Solutions.</em></td>
</tr>
<tr>
<td>12h30</td>
<td>Lunch</td>
</tr>
<tr>
<td>14h00</td>
<td>Computer Graphics. Chairman: D. Kourie</td>
</tr>
<tr>
<td></td>
<td>C. F. Scheepers, CSIR.</td>
</tr>
<tr>
<td></td>
<td><em>Polygon Shading on Vector Type Devices.</em></td>
</tr>
<tr>
<td>14h35</td>
<td>P. Gorringe, CSIR.</td>
</tr>
<tr>
<td></td>
<td><em>Hidden Surface Elimination in Raster Graphics Using Visigraphics.</em></td>
</tr>
<tr>
<td>15h15</td>
<td>Coffee</td>
</tr>
<tr>
<td>15h30</td>
<td>Database Systems I. Chairman: B. von Solms.</td>
</tr>
<tr>
<td></td>
<td>M.E. Orlowska, UNISA.</td>
</tr>
<tr>
<td></td>
<td><em>On Syntax and Semantics Related to Incomplete Information Databases.</em></td>
</tr>
<tr>
<td>16h05</td>
<td>M.H. Rennhakkkamp, Stellenbosch University.</td>
</tr>
<tr>
<td></td>
<td><em>Modelling Distributed Database Concurrency Control Overheads</em></td>
</tr>
<tr>
<td>18h00</td>
<td>Cocktail Party in Cullinan Room A.</td>
</tr>
<tr>
<td></td>
<td>M. Morris, UNISA.</td>
</tr>
<tr>
<td></td>
<td><em>The Development of a Fault Tolerant System for a Real-time Environment.</em></td>
</tr>
<tr>
<td></td>
<td>B. H. Venter, CSIR.</td>
</tr>
<tr>
<td></td>
<td><em>A New General-purpose Operating System.</em></td>
</tr>
<tr>
<td></td>
<td>Coffee</td>
</tr>
<tr>
<td></td>
<td>Computer Languages II. Chairman: J. Bishop.</td>
</tr>
<tr>
<td></td>
<td><em>The Representation of Chemical Structures by Random Context Structure Grammars.</em></td>
</tr>
<tr>
<td></td>
<td>W. van Biljon, CSIR.</td>
</tr>
<tr>
<td></td>
<td><em>A Generalised Expression Structure.</em></td>
</tr>
</tbody>
</table>
**DAY 2**

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>08h30</td>
<td>Keynote Address by Professor Niklaus Wirth, Swiss Federal Institute</td>
</tr>
<tr>
<td></td>
<td>for Technology, Zurich.</td>
</tr>
<tr>
<td></td>
<td><strong>An Extensible System and a Programming Tool for</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Workstation Computers.</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Computer Networks and Protocols I.</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Chairman: P.S. Kritzinger.</strong></td>
</tr>
<tr>
<td>09h30</td>
<td>A.E. Krzesinski, University of Stellenbosch.</td>
</tr>
<tr>
<td></td>
<td><strong>An Approximate Solution Method for Multiclass Queueing Networks</strong></td>
</tr>
<tr>
<td></td>
<td>with State Dependent Routing and Window Flow Control.</td>
</tr>
<tr>
<td>10h05</td>
<td>J. Punt, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><strong>A Protocol Validation System.</strong></td>
</tr>
<tr>
<td>10h30</td>
<td>COFFEE</td>
</tr>
<tr>
<td>11h00</td>
<td>P.S. Kritzinger, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><strong>Protocol Performance using Image Protocols.</strong></td>
</tr>
<tr>
<td>11h35</td>
<td>Invited Lecture by Professor G. Lasker, University of Windsor, Ontario.</td>
</tr>
<tr>
<td></td>
<td><strong>The Relationship of Natural and Artificial Intelligence.</strong></td>
</tr>
<tr>
<td>12h30</td>
<td>LUNCH</td>
</tr>
</tbody>
</table>

**Artificial Intelligence.**
Chairman: G. Lasker.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>14h00</td>
<td>A. Cooper, CSIR</td>
</tr>
<tr>
<td></td>
<td><strong>A Data Structure for Exchanging Geographic Information.</strong></td>
</tr>
<tr>
<td>14h35</td>
<td>A. I. Newcombe, University of Cape Town and R. Rado, National Library</td>
</tr>
<tr>
<td></td>
<td>of Medicine, Maryland.</td>
</tr>
<tr>
<td></td>
<td><strong>Strategies for Automatic Indexing and Thesaurus Building.</strong></td>
</tr>
<tr>
<td>15h15</td>
<td>COFFEE</td>
</tr>
</tbody>
</table>

**Database Systems II.**
Chairman: C. Bornman.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>15h30</td>
<td>M.J. Philips, CSIR</td>
</tr>
<tr>
<td></td>
<td><strong>An Approach to Direct End-user Usage of Multiple Databases.</strong></td>
</tr>
<tr>
<td>16h05</td>
<td>S. Berman, University of Cape Town.</td>
</tr>
<tr>
<td></td>
<td><strong>A Semantic Data Model Approach to Logical Data Independence.</strong></td>
</tr>
</tbody>
</table>

**Information Systems.**
Chairman: D. Teichrow.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>14h00</td>
<td>L. du Plessis and C. Bornman, UNISA.</td>
</tr>
<tr>
<td></td>
<td><strong>The ELSIM Language: an FSM-based Language for the ELSIM SEE.</strong></td>
</tr>
<tr>
<td>14h35</td>
<td>J. Mende, University of the Witwatersrand.</td>
</tr>
<tr>
<td></td>
<td><strong>Three Packaging Rules for Information System Design.</strong></td>
</tr>
<tr>
<td>15h15</td>
<td>COFFEE</td>
</tr>
</tbody>
</table>

**Computer Languages III.**
Chairman: N. Wirth.

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>15h30</td>
<td>W. van Biljon, CSIR</td>
</tr>
<tr>
<td></td>
<td><strong>Experience with a Pattern-matching Code Generator.</strong></td>
</tr>
<tr>
<td>16h05</td>
<td>C. Mueller, University of the Witwatersrand.</td>
</tr>
<tr>
<td></td>
<td><strong>Set-oriented Functional Style of Programming.</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Time</th>
<th>Event</th>
</tr>
</thead>
<tbody>
<tr>
<td>16h45</td>
<td>Open Forum with professors G. Lasker, D. Teichrow and N. Wirth.</td>
</tr>
<tr>
<td></td>
<td><strong>Moderator: Dr. D. Jacobson.</strong></td>
</tr>
<tr>
<td>19h30</td>
<td>Symposium Banquet in Cullinan Room.</td>
</tr>
<tr>
<td></td>
<td><strong>Guest speaker, Dr. D. Jacobson., Group Executive: Technology, Allied Technologies Limited.</strong></td>
</tr>
</tbody>
</table>
08h00  Registration (Tutorial only).

08h30  Tutorial.
The Tutorial will be given by professor Niklaus Wirth, Division of Computer Science, Swiss Federal Institute of Technology, Zurich.

The use of Modula-2 in Software Engineering.
Topics to be covered include:

What is Software Engineering?
Data types and structures.
Modularization and information hiding.
Definition and implementation parts.
Separate compilation with type checking.
Facilities to express concurrency.
Pompous programming style.
What could be excluded?

12h15  Close of Symposium.

12h30  LUNCH
Ada for multi-microprocessors: some problems and solutions

Judy M Bishop
Computer Science Department
University of the Witwatersrand
Johannesburg 2050

Abstract

This paper looks at ways of obtaining a distributed version of an Ada program. Since current approaches require a large investment in system software, we propose instead the use of an Adapter, which can be a manual methodology or an automatic translator. The Adapter accepts source of a concurrent Ada program, adds communication and control tasks, and produces source for a single distributed Ada program, which can then be compiled and run on a multi-processor computer. The process of adaption does not alter the contents of any package in the original program, so that the method is directly applicable to systems that make use of library and generic packages. The method has been applied to programs of the client-server mode, and could be adapted for other rendezvous-based languages such as occam.

Keywords: distributed programming, multi-microprocessors, virtual nodes, concurrency, Ada tasking, transputers.
1. Introduction

With the growing availability of multi-microprocessor systems, ordinary programmers wishing to take advantage of the parallel execution these systems offer are faced with the task of distributing their programs. The first step is to rewrite a sequential program as a concurrent one. The second step is to prepare the program so that parts of it can be distributed onto various processors.

We have found the second step to be quite difficult for computers comprised of independent processors with no shared memory and with limited communication links between them. Even if every processor can communicate with every other, the algorithms for synchronisation on a centralised system have to be amended to work on a distributed system, in order to take account of delays in the reception of information, and the limited perception of the system available in each processor [Andre et al 1985].

There are two approaches that a programmer wishing to distribute an existing program can take

- wait until a clever compiler cum support environment is available, which will understand both the program and the architecture and be able to perform the mapping.
- completely rewrite the program as several separate programs and make them communicate via messages which are handled by additional communication software, perhaps at the operating system level.

These are illustrated in Figures 1 and 2, respectively.

![Figure 1. The "clever compiler" approach to distribution](image)

There are projects working on the first approach, notably the Honeywell Distributed Ada Project [Cornhill 1983, Eisenhauer et al 1986] and the Ada Software Partitioner and Distributer [Armitage and Chelini 1985], and it is possible that such systems will become available in the future. Advantages of this approach are:

- the software development process can proceed without reference to the hardware configuration
- there are no restrictions placed on the way in which in the language is used.

Disadvantages of the approach are:

- the overheads in communication introduced by the compiler into the resulting program cannot easily be controlled by the programmer;
- the resulting system is likely to be machine specific;
- the distributed version of the program is not available to the programmer, which will make debugging difficult.

Because of the high cost of developing such a compilation system, it is not likely that it will be a viable option for ordinary programmers for several years to come.

The second approach relies on less built-in software, but it involves the programmer in a good deal of effort in writing or making use of the necessary message passing protocols. As such, it is more suited to large projects, as for example, the PULSE operating system [Keefe et al 1985] and the Robot Research Laboratory effort described in Volz et al [1985]. The disadvantage of this approach is that communication is not checked across the individual programs, as it would be within a program, and the software development process is considerably complicated by the introduction of message passing protocols. Moreover, adapting these methods for limited number of communication lines may require considerable extra communication software.
Within these two approaches there are variations (see the surveys in Armitage and Chelini [1985] and Burns et al [1986]), but for the ordinary programmer who wishes to obtain immediate benefit from a multiprocessor system, a compromise between the two extremes is needed.

2. An interim approach to distribution

Rationale
This investigation into distribution for the ordinary programmer was initiated by the needs of an ESPRIT Project to build a low-cost, high-performance computer from transputers [Hey et al 1986]. Initially, programming of the computer will be in occam [May and Shepherd 1985], but both Fortran and Ada compilers are in progress. At present, the distributed control and communication software has to be written anew in occam for each case, and although paradigms such as multiplexers are emerging, this duplication of effort is clearly a waste of time and a source of errors. This proposal for an interim approach to distribution is designed to meet the immediate needs of programmers interested in the machine, as well as to provide experience and insight into distributed control for future compiler projects.

Approach
The interim approach presented here is to study the kinds of control software that are needed for a distributed program (and that a clever compiler would need to generate), and to implement these as a collection of routines together with skeleton declarative statements. These are then combined with the original program in a semi-automatic way using an Adapter. The output of the Adapter is an augmented source program which can then be fed into an almost ordinary compiler to produce distributed code. This process is described in Figure 3.

The compiler being used is "almost ordinary" in the sense that it must be able to accept the process to processor mapping, supplied in very simple form, and generate code for multiple address spaces, together with a runtime system that can handle Ada's ordinary inter-task communication across processors. It is not required to generate any additional message passing or message routing code. This is not the same as the action required in the very clever compiler approach, where a description of the processors and their links is given, and the compiler works out which processes should go where, itself generating all the communicating code.

Implementation
The Adapter can be implemented with varying degrees of automation. In its initial form, it consists of a handbook of guidelines which the programmer consults and, using an ordinary editor, makes the necessary additions from the control and communication routines to produce a distributed version of the program. At the other end of the scale, the Adapter can be implemented as a programming tool which performs the merging itself, perhaps with occasional guidance from the programmer. This paper outlines the initial contents of the handbook (or manual version) of the Adapter, and makes reference at the end to the progress that is being made with the programming tool (or automated version).
Advantages

The advantages of this approach are:

- **availability** — the manual version can be used immediately and it requires the minimum of new compiler support
- **accessibility** — having the distributed program in source form enables it to be fine-tuned for efficiency, and augmented for debugging purposes
- **security** — since the distributed version is a single program, it is subject to all the checking provided by the Ada compiler, particularly for messages across processors (which are implemented as Ada variant records, rather than byte strings) and for intertask communication (in the sense of detecting obvious deadlock situations).

3. Tailoring Ada or distributed targets

Tedd *et al* (1984) identified the failings of Ada with respect to distributed programming. These fall into three groups.

- communication between tasks via shared data or nested task types, and the conditional and timed entry call
- the existence of a centralised heap and the use of access objects during a rendezvous
- the packages STANDARD and SYSTEM, and representation specifications.

The last of these refers to different processors, where different representations might be applicable. Tedd outlines a possible solution, but we shall ignore this problem in the current study, since the emphasis is on identical processors. The other two problems have to be solved by a two-level approach.

We start off by recognising the concept of a virtual node. A virtual node consists of Ada program units which together perform some coherent function and may only communicate with other virtual nodes via entry or procedure calls. By partitioning a concurrent Ada program into virtual nodes, we are restricting use of the language at certain boundaries, in order to obtain clean interfaces. The key issue is that Ada as a language is not altered, only that there is a convention as to its use. Typically, one would imagine a one-to-one mapping between virtual nodes and physical processors, but having several virtual nodes on a processor is not excluded. In a testing phase, for example, all the virtual nodes could reside on single processor.
Having partitioned the program, the other problems of the heap and the entry calls can be addressed, and workable solutions are to be found in Tedd et al [1984], Stammers [1985] and Volz et al [1985].

**Virtual nodes and Ada entities**

There is still debate as to which entities in Ada can correspond to virtual nodes. Packages and tasks are both candidates, and several writers seem to feel that one or the other should be chosen [Tedd et al 1984, Schonberg and Schonberg 1985 (for tasks), Burns et al 1986, Welch 1985 (for packages)]. Packages are popular because they embody the data abstraction mechanism and are classed as compilation units, which allows separate compilation. Tasks, on the other hand, have the advantage of being types, which enables them to be replicated, and of providing the full power of the rendezvous for communication between nodes.

This proposal allows both packages and tasks. Tasks are important for replication. The client-server models under consideration rely heavily on multiple instances of the same code. Arrays of tasks, or even dynamically created tasks, will be an essential component of such programs. Packages are essential for encapsulating data types, providing state machines and generally as the building blocks of large Ada systems. The Adapter methodology copes with packages in the full, including generic and library packages. With both Ada entities permitted, the development of the program can proceed with the responsibility of reconciling the differences between packages and tasks being left to the Adapter.

An important advantage of our Adapter approach is that library and generic routines can be incorporated into the distributed system without alteration. This is vital if essentially system packages such as text io or mathematical software are to be available, not to mention the advantages that accrue from re-using tried and tested packages. This point has not been addressed by most other writers.

**4. The Adapter methodology**

The Adapter approach is capable of being applied to any concurrent program in which partitions can be identified and a distributed control algorithm devised to replace unrestricted inter-task communication. Such algorithms are applicable to a definable class of programs. So far, we have considered the class of concurrent programs which have the following properties:

- there is no use of shared global variables
- mutually exclusive access to packages is already programmed in, where required
- tasks in the program are either actors or servers, but not transducers [Welch 1986].

Example of this class are client-server programs, such as the powerful and flexible system suggested by Roubine [1985] (extended by Bishop [1986]) and Monte Carlo simulations. By distributing such programs, speed-up almost proportional to the number of available processors can be obtained. Nevertheless, given the restriction on the number of communication lines, the control of the communication with server tasks and the accessing of packages has to be implemented differently.

To obtain a distributed version of such a program, the tasks are assigned to virtual nodes, the packages are encapsulated inside new tasks, forming additional virtual nodes, and communication is set up between the virtual nodes so as to keep the number of communication lines to a minimum, say two or three. Thus the communication algorithms that have been developed will be applicable to a wide range of multiprocessor system configurations.

The basic elements of the control and communication code are:

- a **control task** in each virtual node through which all communication with other virtual nodes is routed
- a **crescent** of communication lines linking the controllers associated with tasks accessing a common package (known as a resource)
- a **ring** of communication lines linking the controllers associated with tasks that are competing for sole access to a resource.

Two sample configurations from the class of programs under discussion are given below. Each configuration is shown as a diagram using the usual notation for tasks, packages and data flow [Bishop 1987a, Buhr 1985]. Thus a task is represented by a parallelogram, a package by a round cornered rectangle, and data flow by stubby arrows. Grouping of units into virtual nodes is shown by shading.
Figure 4. Original and distributed versions of multiple processes, single resource systems

Configuration 1. Multiple processes communicating to a single resource

Here we consider several identical processes accessing the same package (Figure 4a), with parameters being passed in one direction only. For the distributed version, the package is prefaced by a task, but accesses to the resulting resource have to be decoupled and sent around a crescent, as illustrated in Figure 4b. This decoupling is necessary because there may not be communication lines from every task to the processor on which the package will reside. Calls by the original task on subprograms in the package are redirected as entry calls to the task’s controller. The controller prefaces the parameters with a code for the appropriate subprogram, and sends this as a message around the crescent. Thus the stubby arrow indicating the data flow is shown with a solid stub, indicating that the data includes some control information. The task that prefaced the package looks at the code and calls the relevant subprogram with the parameters given. It may well be desirable in the original algorithm that several processes be executing the same code in a package at any one time. This possibility is preserved in this configuration, since the resource’s task does not wait for the completion of a subprogram before processing another message. A simple example of configuration 1 would be processes wishing to print data via text io. Another example would be Monte-Carlo simulations, where worker tasks communicate their results to a clerk.

Configuration 2. Multiple processes, multiple resources, some exclusive access

With data coming into the processes (as, for example, in response to a text_io.get), this is a more general system than the previous one, and requires messages to be labelled with source and destination identifiers. Figure 5a shows an example of a system with several processes and two resources. Exclusive access to one of the resources is required, and this is handled by a guard task which synchronises requests and releases issued by the processes. In the distributed program (Figure 5b) the guard task cannot necessarily be accessed by every process, and so its functions have to be distributed among the controllers. This is done by setting up a separate ring of communication lines circulating around the controllers, on which they send and accept an interrupt known as privilege. When the privilege arrives at a controller, it is able to accept a request from its process, after which it will transmit all messages down the crescent to the resource until a release is encountered. At this point, it passes the privilege on. If the controller receives the privilege and no request is forthcoming from its process within a short period, it passes the privilege on.

The crescent is now closed at the top, to enable data to be returned from the resources. The messages are passed around the crescent as usual, but when a process wants to activate a subprogram in a resource, its controller first sends a special message around the crescent and expects a reply. The resource sends the data into the top of the crescent and it drops through until it reaches a controller which will accept it. Throughout this time, the process itself is blocked, but the controller is still able to pass messages through. The controllers for both process and resources may receive messages that are for another process and have to be forwarded. This would occur at the point where the controller is actually waiting for a message of substance; the controller can still operate in a passive mode and forward messages untouched while it is not doing anything else.
5. The Adapter methodology in Ada

The original program consists of tasks and packages. Each of these is assigned to a virtual node and the intention is that each virtual node will reside on a separate processor. If this is not to be the case, then the programmer can do one of three things:

- rewrite the program so as to group tasks and packages more coarsely
- allow the adaptation to proceed and tolerate the additional control that will now exist within a processor
- inform the Adapter of the grouping that is required.

The last of these is possible in both the manual and automatic Adapters, and is the most general solution. For the sake of simplicity, we shall assume that a processor per virtual node is satisfactory.

The adaptation now proceeds by considering each process (task) and each resource (task or package) and applying the transformations and making the additions necessary to produce the distributed program. The Adapter methodology for this class of programs has been fully described in Bishop et al [1987b] and has been tested on several programs. The important points about the Ada implementation can be summarised here.

Messages

A basic message consists of the name of the remote subprogram that is being called, and the parameters, if any. The parameters are not converted to bytes, as would happen on a system where several Ada programs are communicating, but are formed into a variant of a record, which is discriminated by the name of the subprogram in question. Thus full type checking of parameters is maintained across virtual nodes. For example, if some text_io routines and some random number routines are being called, then the types for the variant and the message record itself might be:
TYPE subprograms is (put1_prog, put2_prog, get_prog, new_line_prog, seed_prog, random_prog, prog);

TYPE parameters (subprogram : subprograms := prog) is
RECORD
  when put1_prog => s : fixed_string;
  when put2_prog => n : integer;
  when get_prog => m : integer;
  when new_line_prog => null;
  when seed_it_prog => null;
  when random_prog => limit : positive; r : integer;
END RECORD;

Notice the use of a type fixed_string where string would have been expected. This is caused by the need in Ada for actual parameters to be constrained. Thus the process will pass a constrained string to the controller, which can accept it into an unconstrained formal string parameter, but must copy it into a constrained local string before passing it on. It is important to preserve the original length of the string, and array slices are used for this purpose.

The processes

Each actor task or task type $T$ is replaced by a record which has the original task as one field and a controller task as the other.

TYPE $T$_sites is RECORD
  process : $T$;
  control : $T$_controllers;
END RECORD;

This type forms the basis for the replication of tasks that are connected on a crescent. The specification for the controller includes entry points for every subprogram in all the resources that the process accesses, i.e.

TASK TYPE $T$_controllers is
-- communicating with the process
-- CALLS initialise (name : $T$_site_range);
ENTRY ...
ENTRY Ci(pars) -- for every subprogram call in the process body.
ENTRY ...
-- communicating with other $T$_sites and the resource
-- CALLS initialise (name : $T$_site_range);
ENTRY initialise (name : $T$_site_range);
-- CALLS send (par : parameters);
ENTRY send (par : parameters);
-- further entries depend on the case: see point 6.
END $T$_controllers;

Then, by including a renaming declaration of the form

DECLARE $P$ : $T$_controllers renames $T$_site(id).control;

subprogram calls in the process are redirected to the controller, from where they can be routed around the system. However, function calls, which have a different syntax to procedure calls and are not equivalent to entry calls, must be replaced by corresponding calls in the procedural form, using out parameters and temporary variables.

The body of the controller task is straightforward, consisting of a loop around a select statement, accepting calls from the process for access to resource subprograms, and calls from the previous site for the routing of messages around the crescent. For programs that require mutually exclusive access to resources, the privilege interrupts are also handled. The few lines that handle the routing are:

accept send (par : parameters) do hold := par; end;
if last then resource.send(S);
  else site(next).control.send(s);
end if;
It is interesting that one Ada compiler got quite upset at this sequence, since it thought it constituted a recursive entry call!

The resources

Once again, a task is created to sit in front of each package or task resource. If the unit is called \( P \), then a simple version of the task is

```ada
TASK P_resource is
   ENTRY send (par : parameters);
   -- CALLS send (par : parameters); to site(o)
END P_resource;
```

```ada
TASK BODY P_resource is
USE P;
hold : parameters;
BEGIN
   LOOP
      accept send (par : parameters) do
         hold := par;
      end;
      case hold.subprogram is
         when C1 => .. .
         when C2 => ... call appropriate subprogram in P
         when Cn => .. .
         when others => null;
      end case;
   END LOOP;
END P_resource;
```

Where the subprogram has out parameters, then the call to the appropriate subprogram is followed by a reply of the form:

\[
T_{\text{site}}(\text{T.siterange'=first}).\text{control.send(hold);}\]

If more than one resource is present, then each must be assigned a name, as a constant in its declarations, or via the initialisation entry. The accepting of messages is then replaced by the loop which checks the destination and passes the message on to the next resource if necessary.

On Ada paradigms

A by-product of this initial study was the additional insight it gave into the power of Ada in supporting data abstraction and concurrency. The more unusual Ada features of variant records, task types, renaming tasks and records of tasks were all used to good effect. Making better use of Ada in this way contributed to the freedom of having packages or tasks as virtual nodes. With Ada being a relatively new language, there is still room to explore paradigms and to investigate how the language should best be used. Two similar studies have adopted the program adaptation approach for other purposes and have described the Ada code involved in sufficient detail for the method to be followed by hand. In Lamb and Hilfinger [1980], the simulation of procedure variables and procedures as parameters was discussed, and in Roubine [1985] there is the creation of a dynamic Ada system with multiple clients and servers whose names and types are unknown to each other. The work of Fantechi \textit{et al} [1986] and of Welch [1986] also falls into this category.

6. Case study

As an example of the method being applied to an ordinary program, consider a simple Monte Carlo simulation of collisions of neutrons in a concrete wall. The concurrent program has ten worker tasks performing the same calculations and passing results on to a collector task at the end, which sums them, as shown in the profile diagram of Figure 6a. This is a simple program of the sort covered by configuration 1 but serves to illustrate the Adapter process well. With the workers executing on separate processors, the speed-up over a single processor would be almost proportional to the number of worker tasks in action. It is therefore exactly these
simple types of programs for which an easy route to distribution is required.

As evident from Figure 6a, the program called *Neutron_2* has one array of actor tasks – these qualify as processes in the Adapter terminology – and one server task which qualifies as resource. The four library packages that are accessed could also be classed as resources and reside on separate processors. This would be very inefficient, though, and instead we group them with the workers and collectors to form composite virtual nodes. It is important to recognise this trade-off between replication and efficiency. In some cases, it may not be feasible to have copies of library packages on every processor, in which case they must become resources and be accessed via the crescent or ring communication lines.

Having made the decision about the composition of the virtual nodes, the Adapter methodology comes into play. Each of the worker tasks acquires a controller task, and these are grouped in a record. The field name for the worker tasks is *process*. A process wishing to send results to the collector is diverted to the corresponding control field of its site, which takes care of passing the data around the crescent formed with the other control fields. Eventually the data arrives at the collector resource, which in tum passes it on to the actual collector. A diagram of the units in the adapted program is given in Figure 6b. The full Ada program is given in Bishop *et al* [1987b].
7. Conclusions

The Adapter approach provides an immediate means whereby a program can be prepared for distribution. By distributing the control of the communication between virtual nodes to special controller tasks within each virtual node, the demands made on the compiler are kept to a minimum. These demands are the ability to generate code in a multiple address space, and to handle a rendezvous between two directly connected nodes. The Ada runtime system that will be resident at each site will not have to take care of a rendezvous with remote sites further down a line: this is all incorporated at the source level.

In preparing the Adapter algorithms for Ada client-server programs, good use was made of the more unusual aspects of Ada typing and tasking. The virtual node concept was used and expanded, and the result is that very few restrictions have to be placed on the nodes and their connections prior to the application of the Adapter. It was encouraging that library and generic packages could be incorporated as virtual nodes without change.

Work on expanding the scope of the algorithms, of implementing them in occam, and of designing the automatic version of the Adapter is continuing.

Acknowledgements

This work was inspired by the needs of the ESPRIT transputer array project at Southampton University, and the help of the members of that team is gratefully acknowledged. I would like to thank the the University of the Witwatersrand and the CSIR for financial assistance during this work.

References

Bishop J M, Data abstraction in programming languages, Addison-Wesley, Wokingham 1986.
Cornhill D, A survivable distributed computing system for embedded application programs written in Ada, Ada Letters III (3) 79-87, November 1983.