South African
Computer
Journal
Number 6
March 1992

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 6
Maart 1992

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal

An official publication of the South African Computer Society and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif

'n Amptelike publikasie van die Suid-Afrikaanse Rekenaarvereniging en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof John Schochot
University of the Witwatersrand
Private Bag 3
WITS 2050
Email: 035ebrse@witsvma.wits.ac.za

Production Editor
Prof G de V Smit
Department of Computer Science
University of Cape Town
Rondebosch 7700
Email: gds@cs.uct.ac.za

Subscriptions

	Annual	Single copy
Southern Africa:	R45_00	R15_00
Elsewhere	$45_00	$15_00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Computer Science Department, University of Pretoria, Pretoria 0002
Phone: (int)+(012)+420-2504 Fax: (int)+(012)+43-6454 email: dkourie@rkw-risc.cs.up.ac.za

Sigma Press

Guest Contribution

This guest contribution is a slightly edited report to the Foundation for Research and Development (FRD) drawn up by Ed Coffman. Ed was an FRD-sponsored guest at the 6th South African Computer Research Symposium. The report was not originally intended for general distribution. Rather, it was specifically compiled for the FRD and its staff. I am therefore grateful to both the FRD and the author for agreeing to its publication in SACJ. I believe that it contains several incisive observations that merit further thought and discussion amongst South African computer scientists. (Editor)

Impressions of Computer Science Research in South Africa

E. G. Coffman, Jr.
AT&T Bell Laboratories, USA

In commenting on the cross section of computer science research in South Africa, I will use the classification in the table of contents of the "Summary of Awards: Fiscal Year 1989," a document published recently by the US National Science Foundation. Of the 5 categories, I will treat Numeric and Symbolic Computation as inappropriate for the discussion below. In this category I noted no research in the computer science setting in South Africa. It is also common in the US and elsewhere to place this effort in other departments, e.g., departments of mathematics or applied mathematics.

Of the remaining categories I found South Africa to be strongest in software systems and engineering, to have a substantial investment in computer systems and architecture, and to be weakest in computer and computation theory.

The coverage in software systems and engineering (SSE) was broad, topical, and similar in scope to that in US universities. Technology transfer and the corresponding relations with industry seemed to be in place or developing along promising lines. I comment in passing that this was rather surprising to me. In the US the development of SSE within university departments has lagged behind almost all other disciplines of computer science. A primary problem has been the insatiable appetite of industry for all Ph.D. graduates in the SSE field.

The investment in parallel processing, computer networks, and distributed computing appears sound, although I expected to see a greater emphasis on mathematical foundations (see my remarks below), particularly in the parallel algorithms area. Given current resources, South African institutions are doing remarkably well in computer science research. But computer science is a fundamentally important course of study, beginning at an early age and extending through graduate Ph.D. research; I take this as sufficiently obvious that I need not dwell on justifications. With this in mind, and with the necessary resources in hand, South Africa should, in my opinion, expand and consolidate its computer science research effort, increase its visibility in the international arena, and correct the rather thin distribution of graduate research among universities.

I can see much of this proceeding along present lines, but I would strongly recommend a concerted development in computer and computation theory (CCT), education and research; this is mainstream computer science and forms the basis for virtually all other fields of study within computer science. It is by no means absent in South Africa curricula, but it appears to be under--represented in advanced studies and Ph.D. level research.

At the graduate level CCT is heavily mathematical. I understand that mathematical foundations are supplied by mathematics departments in certain cases. This is not ideal, but workable and it is justified by limited resources. However, it is important that mathematics departments not regard this as a mere service; faculty will have to make a major commitment to theoretical computer science, publishing in its leading journals (e.g. SIAM Journal of Computing, Journal of the ACM, Journal of Algorithms, Algorithmica, Journal of Computer and Systems Sciences, Theoretical Computer Science, etc.), and providing the supervision of theses sponsored by computer science departments and leading to degrees in computer science. I would also encourage active participation in the international computer science "theory" societies and their meetings; two highly prestigious examples of the latter are the annual Symposium on the Theory of Computing and the Foundations of Computer Science conference.

Returning to the thin distribution of computer science research, I would make the following point. If the current situation is only a stage of development - i.e., if further resources (both human and financial) can be counted on to bring at least a few of the departments to a critical mass - then little needs to be said beyond the earlier remarks. Critical mass is hard to define, but calls for adequate, expert coverage of mainstream computer science re- search. In view of the breadth of this research, 8-10 Ph.D. full- time-equivalent faculty would seem to be barely adequate; with the usual clumping of faculty in specific research areas, more would be expected. South Africa has a talent base such that there is little doubt that such departments would achieve a much wider international recognition.

On the other hand, if resources remain fixed at current or even slightly retrenched levels, then I would recommend consolidation to achieve the same goals on a smaller scale. Within a university this can often be done by establishing interdisciplinary, degree-granting laboratories or institutes of computer science, which bring together the computer science efforts located in various departments other than computer science, such as electrical engineering, industrial engineering, business/-management science, mathematics, and operations research. The idea is to enjoy the advantages (opportunity, synergy, awareness, etc.) to both students and faculty of reasonably large computer science programs. There are many examples of such intramural laboratories in North America and Europe.

This approach could also be considered among universities within a confined geographical area, admittedly with greater difficulty perhaps. The Institute of Discrete Mathematics and Computer Science connecting Princeton University, Rutgers University, AT&T Bell Laboratories, and Bell Communications Research is a possible model. Examples in South Africa might consist of universities and research institutions on the Reef or those in the Western Cape (just to mention those with which I'm a little familiar).

As a final comment, I should note that my impressions have been based on limited information which may not give a representative picture. I am sure that my reactions will be appropriately discounted where I have been off target.

Editor's Notes

Prof John Schochot has graciously accepted to be SACJ's subeditor for papers relating to Information Systems. Authors wishing to submit papers in this general area should please contact him directly. I look forward working with John, and to a significant increase in IS contributions in future.

The hand of the new production editor, Riël Smit, will be clearly evident in this issue. Those papers not prepared in camera-ready format by the authors themselves were prepared by him in TEX. He will be announcing revised guidelines for camera-ready format in a future issue. If you use TEX or one of its variations, Riël would be happy to provide you with a styles document to SACJ format.

At last some Department of National Education committee has decided that SACJ should now be on the list of approved journals. This places it amongst the ranks of some 6800 other journals. These include not merely a number of ACM and IEEE Transactions but also such journals as *Ostrich*, *Trivium*, *Crane Bag*, *Koers*, *Mosquito News*, *Police Chief*, *Connoisseur*, *Lion and the Unicorn*, *About the House* and *Ohio Agricultural Research and Development Center Department Series ESS*. You will recall that in 1990 this same committee decided that, if judged on its own merits, SACJ did not deserve to be on the illustrious list. In the absence of other evidence, we must assume that the sole reason for its revised decision is that SACJ's predecessor, Quæstiones Informaticæ, was there. (I have a secret suspicion that the committee liked that name.)

It is my understanding that for official purposes, all journals on this list are regarded as *equally* meritorious, and all of them are more meritorious than *any* conference proceedings. What does all of this mean?

The momentous implication of the committee's deliberations is that the State will not give your institution a single cent for anything that you publish in SACJ. Instead, the State and your institution will scrupulously keep a score of the annual number of publications that count - but actually don't - because someday they might! And to encourage your enthusiastic participation in this Alice in Wonderland exercise, your institution might actually give you some of the standard subsidy funding that the State should have provided according to its own formulae, but didn't.

You will not be allowed to use this money to buy yourself a car - not even a casual meal. You may only use it to finance activities that are provably directed towards producing more papers in approved journals. The great consolation, of course, is that you will not be required to pay income tax on this money. The only tax involved will be the VAT component when you spend it in an approved manner. As a good computer scientist who enjoys recursion, my vote would be that all such revenue collected by the State should be earmarked to be placed in the pay packets of committee members who decided that SACJ should be approved.

If you publish in these approved journals with sufficient regularity and enthusiasm you will almost deserve to be regarded as a researcher. What you additionally need to do, is to ensure that you befriend and impress at least three overseas referees. You then apply to the FRD for official recognition as a researcher, and if they are sufficiently impressed, they will give you more of the non-taxable kind of money that you need to spend on research to publish in approved journals.

Derrick Kourie
Editor

The CSP Notation and its Application to Parallel Processing

P.G. Clayton
Department of Computer Science, Rhodes University, Grahamstown, 6140 RSA
Internet: cspc@alpha.ru.ac.za

Abstract

The principles put forward by Hoare's CSP proposals have been a foundational influence in the development of distributed and parallel processing software and hardware. This paper provides a brief introduction to the notation, and reviews the influence of CSP on concurrent programming languages and formal methods. Samples of both the language principles and the mathematical theory of CSP are presented, the primary applications of the notation are discussed, and some introductory level references to the topic are suggested.
Keywords: *CSP, communicating sequential processes*
Computing review categories: *D.3.3, F.3.1*

1. Introduction

In 1978, Professor Tony Hoare published a paper on Communicating Sequential Processes [12] that was regarded almost immediately as classic. CSP, as the notation described in Hoare's paper quickly became known, has since become a model for the development of a generation of concurrent programming languages and communicating microprocessors, and has formed the foundation of a branch of mathematical theory for describing the behaviour of concurrent systems.

Hoare's original proposal is particularly attractive in its handling of communication and synchronization, and it is largely this aspect that has made CSP a landmark in the field of parallel and distributed processing. The notation combines the needs of data communication and synchronization in a single mechanism, the *rendezvous*, which is viewed at the same primitive level as assignment. This mechanism maps efficiently onto a large class of distributed multicomputer architectures, making use of point to point interconnections.

Although its influence has been widespread, the programming notation suggested by CSP lacks much of the syntactic structure necessary for a production quality programming language; it is in fact a programming language fragment, designed with the minimum syntax necessary to demonstrate the novel features which Hoare was proposing. As a consequence, programming languages based on CSP principles are of necessity supersets or adaptations of the proposal.

To date, the most notable commercial application of CSP principles has been in the Occam [19] programming language, developed by Inmos Ltd. as a programming notation for its family of transputer VSLI processing chips. Initially, Occam was designed to be little more than an implementation of Hoare's CSP. The language was soon superseded by a larger language, Occam 2 [20], to satisfy a programmer demand for additional convenience and security features. A greatly extended version of the language is now in the design stage [2], to be known as Occam 3. Of considerable significance to the CSP influence was the close design correspondence between Occam and the transputer family. Transputers were designed to execute Occam efficiently, and special attention was given to the handling of such features as rendezvous, process creation, and context switching at the hardware level. In essence, a network of communicating transputers can be viewed as a CSP machine.

In his original CSP proposal [12], Hoare carefully highlighted, as one of the deficiencies of his notation, that he had not suggested any proof method to assist in the development and verification of programs. In the ensuing years he set about creating such a formal method. With two colleagues, he developed a mathematical theory to underpin the programming language principles of CSP [6], and he later published an introductory textbook based upon this theory [13]. This work was influenced by the ideas of another pioneer of formal methods for concurrent systems, Professor Robin Milner. In return, Hoare's work has strongly influenced the design of Milner's calculus of communicating systems [22], Brinksma's LOTOS language for the formal description of communication protocols [5], the Z notation for specifying and designing software [31], and other leading work on the formal specification and analysis of concurrent systems. The theory has been used as a direct notational underpinning for developing tools for reasoning about commercial programming languages. For example, the transformational semantics [27] for Occam developed at the Oxford University Computing Laboratory support reasoning about programs written in that language.

This paper introduces CSP and outlines its

foundational influence in the development of distributed and parallel computing models. From a notational point of view, the paper is intended to be a brief, informal introduction to CSP, not a definitive description. The discussion of the CSP notation is therefore superficial, and a brief glossary of symbols is situated at the end of the paper to aid understanding of the examples. The set of references points readers to both introductory and more advanced literature describing the notation, and its application and implementation on computers.

2. CSP - A Programming Notation

The CSP programming notation views processes as sequential execution units, which communicate using a message passing model. The concept of processes is fundamental to the structure of CSP. A CSP program should be viewed as a hierarchy of processes; a complete program is considered to be a single process which is composed of sub-processes, each of which is decomposed further into component processes. Each process itself can be large and complex or, at the other extreme, a process may be a single primitive operation.

The notation is introduced in this section by example. Program constructors have a terse mathematical form, and declarations and expressions are presented in a Pascal-like notation.

There are three primitive processes in CSP for the operations of input (represented by *?*), output (represented by *!*), and assignment (represented by the familiar *:=* symbol). Processes may be named, and the names of processes are used to denote the source or destination of an input or an output operation. For example (curly brackets are used to contain comments):

A ! e {output the value e to process A}

B ? x {from process B input to x}

Message passing provides the appropriate means for sharing information between communicating processes of a program, and references to shared data items between communicating processes have to be mapped by the programmer onto sets of message exchanges. The effect of the rendezvous comprising the above communication pair is to have executed the assignment

x := e

across the boundary of two parallel processes, where *x* is a variable in process *A* and *e* is an expression in process *B*.

As the communication is synchronous, the first process to execute one of the operations in the communication pair will be suspended until the other process reaches the matching communication primitive. Only when both processes are ready to communicate will data pass from *e* to *x*. Both processes will then proceed, independently and concurrently.

Square brackets are used to denote the scope of complex process definitions, and the symbol || is used to indicate the simultaneous execution of processes.

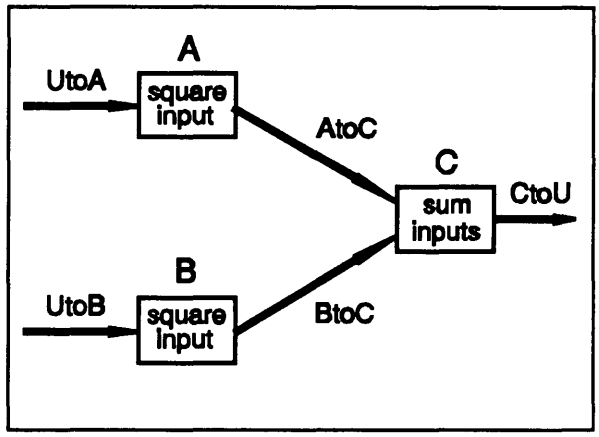

Figure 1 - Three concurrent processes.

Figure 1 illustrates the interaction of three concurrent processes. Processes *A* and *B* each square their input values and pass them on to process *C*, which computes the sum of its input values. Process *A* can be described by two CSP statements, executed sequentially.

*USER ? x; C ! x * x {a semicolon in CSP denotes sequential execution}*

Process *B* can be similarly described.

*USER ? y; C ! y * y*

In the case of process *C*, a parallel construct is required, since the two input values may arrive in either order.

A ? x || B ? y

Once both input values have been received, their sum must be output. So, the complete specification for process *C* will be:

[A ? x || B ? y] ; USER ! x + y

By adding names (using a double quote notation) and variable declarations to processes, the whole of figure 1 can be described in CSP by the composite process:

```
[ A::x:integer; USER ? x; C ! x * x
|| B::y:integer; USER ? y; C ! y * y
|| C::x,y:integer; [A ? x || B ? y] ; USER ! x+y ]
```

In any concurrent programming language basing its interprocess communication facilities on synchronous message passing, there is a need to allow a process to wait for one of a number of possible communications.

CSP handles this form of non-determinism using the symbol [] to represent an alternate structure in an adaptation of Dijkstra's guarded command [8]. To illustrate a guarded expression, consider the CSP form of process *A* from figure 1, modified to repeat indefinitely.

*[*x:integer; USER ? x → C ! x * x*]

The leading asterisk denotes a repetitive process which terminates when the guard *USER ? x* fails. Each time the guard succeeds, the process must perform the guarded command, *C ! x * x*, before re-examining the guard.

To illustrate the alternate structure, process *A* might be further modified to accept input values from one of two alternative sources.

*[*x:integer; USER1 ? x → C ! x * x*
[] *x:integer; USER2 ? x → C ! x * x*]

The semantics of this process are that one of the alternatives whose guard does not fail will be executed. If all guards fail, the alternative command (and consequently the repetitive command as well in this example) fails; if more than one guard succeeds, one is arbitrarily selected and executed.

Boolean guards may be used in place of - or together with - input guards in CSP's alternative clause to provide for conditional execution.

3. Implementations of CSP programming features

3.1 Occam

Occam[1] is a high level language in the imperative mould, which is intended for programming many interconnected or distributed computers. It adopts the CSP concept of programs comprising a hierarchy of processes, and has the same three primitive processes as CSP[2]. In Occam, processes communicate by naming predeclared channels rather than processes, as in CSP. Scope and block structure is denoted by strict indentation rather than with square brackets, and infix constructor symbols such as || and ; are replaced by prefix constructors with names such as *PAR* and *SEQ* (comprising more readily realizable characters in case of unusual symbols such as ||).

An Occam equivalent for the system described in figure 1 is (program comments are introduce by the character pair --):

```
CHAN OF INT UtoA, UtoB, AtoC, BtoC, CtoU :
PAR

  -- process A
  INT x :
  SEQ
    UtoA ? x
    AtoC ! x * x

  -- process B
  INT y :
  SEQ
    UtoA ? y
    AtoC ! y * y

  -- process C
  INT x, y :
  SEQ
    PAR
      AtoC ? x
      BtoC ? y
    CtoU ! x + y
```

This example has been written to correspond closely to the CSP version presented earlier. However, it would be more clearly and concisely formulated using Occam's syntax for declaring complex processes as procedures with parameters.

The alternate structure of CSP is introduced in Occam by the prefix constructor *ALT*. A modified form of process *A* of figure 1, which is able to select repetitively from one of two possible inputs, might be coded in Occam as:

```
CHAN OF INT UtoA.1, UtoA.2, AtoC :

  -- process A
  WHILE TRUE
    INT x :
    ALT
      UtoA.1 ? x      -- guard
        AtoC ! x * x    -- guarded command
      UtoA.2 ? x      -- guard
        AtoC ! x * x    -- guarded command
```

Although the semantic interpretation of guarded commands is the same in both CSP and Occam, the semantics for abandoning an infinite loop such as the one above differs in the two notations. In addition, Occam has a separate *IF* construct to handle conditional execution, for which the semantics have been further altered from the CSP alternate structure.

3.2 CSP extensions to conventional sequential programming languages

As an alternative to using genuine parallel processing languages, programmers often choose to make use of a popular sequential language, to which concurrent control constructs have been added. In the case of transputer based programming environments, the extensions to such languages most often take the form of CSP-like calls, adapted to conform to the syntax structure of the host language. As an example of the way in which CSP features are adapted in this type of implementation, some interprocess communication calls are listed for the Logical System C (LSC) compiler[3] [23].

LSC provides an interface to the concurrency features

of the transputer by means of a library of C functions. To perform interprocess communication, the programmer is required to set up pointers to physical channel addresses using a standard channel pointer type. Examples of input and output calls are:

```
#include <con.h>
ChanOut (c, cp, cnt)
      Channel *c;
      char    *cp;
      int     cnt;

ChanIn (c, cp, cnt)
      Channel *c;
      char    *cp;
      int     cnt;
```

The CSP alternate structure is implemented by a set of calls which determine the status of channels and indicate which (if any) of a list of channels is ready for input. For example, the *ProcAlt()* function causes the calling process to block until one of the channels in its (zero terminated) argument list is ready for input. On completion, the function returns an index into the parameter list for the channel which is ready for input.

```
int ProcAlt (c1, c2, ..., cn, 0);
      Channel *c1;
      Channel *c2;
      ...
      Channel *cn;
```

3.3 Concurrent programming language implementations based on CSP

Apart from Occam, a large number of concurrent languages have been designed around the principles of CSP. Some profess to be close implementations of CSP, while others merely acknowledge the CSP influence. All of them differ to CSP in some respect, and all contain enhancements in those areas of their notations which are not addressed by the CSP proposals.

Those programming languages which claim to be close implementations of CSP include[4] CSP-i[5] [35], COSPOL [26], CSP-80 [15], PLITS[6] [9], CSP-S [24] and ECSP [1].

Numerous programming notations have been influenced implicitly or explicitly by CSP, particularly in the area of communication. A non-exhaustive list includes Ada [34], CHILL [10], Concurrent-C [11], Input Tools [33], Joyce [4], Nil [32], Planet [7], and the many expressly port-based languages, such as CP (Communication Ports) [18], Directed Ports [30], Ordered Ports [3], and Port Language [17].

4. CSP - A Mathematical Theory

The formal notation of the CSP mathematical theory is illustrated briefly below by the example of two conventional high level language data types which are defined in the form of processes.

By identifying the set of events (termed the *alphabet*) in which each component process of a system may engage, writing out the behaviour at each level of parallel sub-processes as a composite process, and examining the resultant event traces, a lucid view of the system's behaviour can be presented. It is logically impossible for a process to engage in an event which is not part of its alphabet. On the other hand, processes which share a common event must all be ready to engage in the common event for that event to actually occur (common events between processes therefore represent synchronization).

A Boolean variable can be described using two processes, *TRUE* and *FALSE*, to represent its behaviour in each of its two states. These processes may engage in any of the events specified by their event alphabets:

$$\alpha\, TRUE = \{read, fetch_1, assign_0, assign_1\}$$
$$\alpha\, FALSE = \{read, fetch_0, assign_0, assign_1\}$$
$$\alpha\, BOOLEAN = \alpha\, TRUE \cup \alpha\, FALSE$$

The behaviour of the Boolean variable can be specified as a choice of events from the alphabets of processes *TRUE* and *FALSE*. The events which assign a value to the variable determine which state the variable will assume, and consequently which of the processes will describe the ensuing behaviour of the variable.

$$\begin{aligned} BOOLEAN = {} & (assign_0 \rightarrow FALSE \\ & \Box\ assign_1 \rightarrow TRUE) \\ FALSE = {} & (read \rightarrow fetch_0 \rightarrow FALSE \\ & \Box\ assign_0 \rightarrow FALSE \\ & \Box\ assign_1 \rightarrow TRUE) \\ TRUE = {} & (read \rightarrow fetch_1 \rightarrow TRUE \\ & \Box\ assign_1 \rightarrow TRUE \\ & \Box\ assign_0 \rightarrow FALSE) \end{aligned}$$

A legal event trace of process *BOOLEAN* might be:

$$assign_0 \rightarrow read \rightarrow fetch_0 \rightarrow assign_1 \rightarrow read \rightarrow fetch_1 \rightarrow \ldots$$

Another legal trace might be:

$$assign_1 \rightarrow read \rightarrow fetch_1 \rightarrow assign_0 \rightarrow read \rightarrow fetch_0 \rightarrow \ldots$$

But the definition of *BOOLEAN* precludes the trace:

$$assign_1 \rightarrow read \rightarrow fetch_0 \rightarrow \ldots$$

Note that this Boolean variable refuses to allow its value to be fetched until after a value has first been assigned to it.

As a second example, consider how this definition can be extended to cater for a range of values, such as are needed to describe an integer variable capable of storing values in the range $-n$ to $+n$. The behaviour of the

variable in each of its k states can be defined by a process VAL_k. As in the case of the Boolean variable, the integer variable's behaviour can be specified as a choice of events from the alphabets of the VAL_k processes.

$$\alpha INTEGER = \bigcup \alpha VAL_k \quad for\ -n \le k \le n$$
$$\alpha VAL_k = \{read, fetch_k, assign_i\} \quad for\ -n \le i \le n$$

$$INTEGER = ((i: -n..n)assign_i \rightarrow VAL_i)$$
$$VAL_k = ((i: -n..n)assign_i \rightarrow VAL_i \ \square\ read \rightarrow fetch_k \rightarrow VAL_k) \quad for\ -n \le k \le n$$

If the current value stored in the integer variable is not of interest, the alphabet of events can be simplified to ignore the distinction between states which the variable may assume, and to consider only the actions of making references to the variable. A simplified process, which is only concerned with the actions of assigning and fetching values without regard to the magnitude of values, may be depicted as:

$$\alpha INTEGER = \{assign, fetch\}$$
$$INTEGER = (assign \rightarrow X.(assign \rightarrow X \ \square\ fetch \rightarrow X))$$

This definition consists of a single process which, once initiated with the appropriate event prefix, is able to reference an elemental nested process definition recursively (the process X) to describe all possible traces of events that the variable may engage in.

To illustrate concurrency, we turn again to the example from figure 1. Using the notation $c.v$ to represent the event on channel c which communicates the message with value v, the three processes might be described as:

$$\alpha A = \{UtoA.x, square.x, AtoC.x\}$$
$$A = (UtoA.x \rightarrow square.x \rightarrow AtoC.x)$$

$$\alpha B = \{UtoB.y, square.y, BtoC.y\}$$
$$B = (UtoB.y \rightarrow square.y \rightarrow BtoC.y)$$

$$\alpha C = \{AtoC.x, BtoC.y, sum, CtoU.sum\}$$
$$C = (AtoC.x \rightarrow BtoC.y \rightarrow sum \rightarrow CtoU.sum \ \square\ BtoC.y \rightarrow AtoC.x \rightarrow sum \rightarrow CtoU.sum)$$

And the whole system displayed in figure 1 might be described as:

$$FIG.1 = (A \parallel B \parallel C)$$

Events appearing in the alphabets of two or more processes (such as event $AtoC.x$ in processes A and C) represent a synchronization of those processes during the occurrence of that event. Thus process C has to wait for either of processes A or B to reach their final event before it can commence. Hence, a legal trace of process *FIG.1* might be:

$$UtoA.x \rightarrow UtoB.y \rightarrow square.x \rightarrow square.y \rightarrow AtoC.x \rightarrow BtoC.y \rightarrow sum \rightarrow CtoU.sum$$

While another legal trace might be:

$$UtoB.y \rightarrow UtoA.x \rightarrow square.y \rightarrow square.x \rightarrow BtoC.y \rightarrow AtoC.x \rightarrow sum \rightarrow CtoU.sum$$

Once expressed in a precise, formal notation, systems (or sub-systems) become possible to reason about. For example, undesirable event traces can be identified (the system does something that it shouldn't), or the reason for desirable event traces not occurring might be identified (the system could, for example, be in deadlock because the event order is antithetically constrained in communicating processes - the processes are prepared to engage in some further action but cannot agree on which action this should be). It is even possible to detail formal proofs about the properties of the system (such as the absence of deadlock)[7]. Because events which involve only one process cannot affect the interaction between processes, such events (for example *sum* in process C above) can be ignored for the purpose of a proof concerning process interaction. Formal analysis of a system will normally take the route of a sequence of formal specifications, each containing more details than the last, and homing in on areas of complexity or uncertainty.

5. Applying formal methods

It is essential that a software or hardware system should have a precise mathematical meaning. Bringing the precise mathematical meaning of the system to the fore with the aid of a mathematical notation yields two avenues for profit. Firstly, a formal specification acts as a single, reliable reference for all involved with system development, maintenance, and use. Because it is independent of program code or silicon, a formal version of a system can be completed early in the development phase, and can be used as the definitive, rigorous reference, even though it might need to be modified during the course of the project. Secondly, it provides a secure mathematical foundation for reasoning about properties of the program, for detecting and avoiding abstruse errors, and for verifying the correctness of the design. Regrettably, the degree of rigour required for a detailed formal analysis is typically extremely tedious for all but the most modest of systems, and implementors are frequently unfamiliar with the mathematical methods

required.

All the same, formal verification methods are no longer the exclusive property of academic experimentalists, and are being used increasingly for commercial purposes. While it is often not practical to apply formal verification methods to complete hardware or software systems, they are being employed in the development of sub-assemblies in which the complexity, novelty, or subtlety of the design justifies the effort required for formal analysis.

The use of formal methods in software development, particularly in systems which perform critical tasks, is reasonably common place. The increasing complexity of VSLI devices, and the high cost of correcting design errors either before or after a product is released, have made VSLI design a popular application area for formal work in recent years. There now exist commercial companies which specialize in formal verification[8].

The transputer family of microprocessors provides a good example of commercial accomplishment with the aid of formal analysis. Formal methods were used to uncover a number of abstruse design errors in the proposed implementation of the Floating Point Unit (FPU) of Inmos's T800 microprocessor[9] [21]. It has been claimed[10] that the use of formal methods in the design of the T800 FPU not only increased the quality of the finished product, but reduced the design cost and allowed the design to be completed an estimated year ahead of what would otherwise have been achievable. Formal methods have since been used extensively in the design of Inmos's new generation T9000 microprocessor [28].

Further reading

This paper makes reference to a range of publications covering the CSP notation, programming languages, and formal methods. For readers wishing make a further study of the formal handling of concurrent systems, the following topics are recommended (along with references to general introductory texts to each topic): Hoare's mathematical theory of CSP [13], the Z notation for specifying and designing software [31], the LOTOS language for the formal description of communications protocols [5], Milner's calculus of communicating systems [22], and the Petri Net Theory for specifying and modelling systems [25]. A layman's description of the formal work involved in the development of the IMS T800 FPU has been published in New Scientist [29].

Glossary of symbols

Notation	*Meaning*
a, b, ...	words in lower case denote distinct events (*a, b*) or variables denoting events (*x, y*)
P, Q, ...	words in upper case denote specific processes (*P, Q*) or variables denoting processes (*X, Y*)
αP	the alphabet (set of legal events) of process *P*
c.v	the event on channel *c* which communicates the message with value *v*
U	union of sets
$a \rightarrow P$	*a* then *P*
$(a \rightarrow P \,[]\, b \rightarrow Q)$	*a* then *P* choice *b* then *Q* (for deterministic and non-deterministic traces)
X.F(X)	the process *X* such that $X = F(X)$
$P \parallel Q$	*P* in parallel with *Q*
$P \,[]\, Q$	*P* choice *Q*
$x := e$	*x* takes on the value *e*
b ! e	on channel *b* output the value of *e*
b ? x	from channel *b* input to *x*
P ; Q	*P* successfully followed by *Q*

For the complete set of symbols used in CSP, the reader is referred to the text by Hoare [13].

Notes

1. The generic term *Occam* is used to refer to the family of Occam definitions. The Occam examples in this paper conform to Occam 2, the most recent commercially available version of the language.
2. Occam has two additional primitive processes, *STOP* and *SKIP*.
3. Logical System C is a popular C implementation for transputer based parallel processing machines. It is available as a stand-alone system [23], and has been adopted as the key program development tool for the Trollius, Genesis, and Express operating systems.
4. [14] compares a number of CSP-like implementations.
5. CSP-i [35] is an implementation of CSP which does not stray too far from the original [12] language proposals. It was developed at the author's university, and is available in the form of an interpreter suitable for tutorial purposes.
6. As a variation on the CSP rendezvous concept, programming languages exist which make use of asynchronous message passing (the messages from output operations are buffered, and those from input operations time-out if no incoming messages are available). PLITS (Programming Language in the Sky) is an example of a programming language which makes use of asynchronous message passing [9].
7. No examples of formal proofs are shown in this paper due to space limitation. Readers are referred in the first instance to the introductory text by Hoare [13].
8. The formal analysis for the IMS T800 and IMS T9000 processors was undertaken by Formal Systems (Europe) Ltd., which works from Oxford University premises at Unit 7, The S.T.E.P. Centre, Osney Mead, Oxford OX2 0ES, U.K. Formal Systems also has a branch in Auburn, Alabama, U.S.A.
9. This work lead to the conferring of a Queen's Award for Technical Achievement on Inmos Ltd. and Oxford

University Computing Laboratory.

10. *ref.* occam user group newsletter, No. 13, July 1990, page 74-76.

References

[1] F Baiardi, L Ricci, and M Vanneschi, [1984], Static Checking of Interprocess Communication in ECSP, Proc. SIGPLAN84 Symposium: Compiler Construction, *ACM SIGPLAN Notices*, **19**(6), 290-299.

[2] G Barrett, [1990], *The Development of Occam: Types, Sharing and Modules*; in: Zedan, H. *(ed.)*, *Real-time Systems with Transputers*, Proc. OUG TM-13, York, Sept., IOS Press, Amsterdam.

[3] J Basu, L M Patnaik, and A K Goswam, [1987], Ordered Ports - A Language Concept for High-Level Distributed Programming, *Computer Journal*, **30**(6), 487-497.

[4] P Brinch Hansen, [1987], Joyce - A Programming Language for Distributed Systems, *Software - Practice and Experience*, **17**(1), 29-50.

[5] E Brinksma, [1988], *Information Processing Systems - Open Systems Interconnection - LOTOS - A Formal Description Technique based upon the Temporal Ordering of Observational Behaviour*, Draft International Standard IOS8807.

[6] S D Brookes, C A R Hoare, and A W Roscoe, [1984], A Theory of Communicating Sequential Processes, *J. ACM*, **31**, 560-599.

[7] D Crookes and J W G Elder, [1984], An Experiment in Language Design for Distributed Systems, *Software - Practice and Experience*, **14**(10), 957-971.

[8] E W Dijkstra, [1975], Guarded Commands, Nondeterminacy, and Formal Derivation of Programs, *Comm. ACM*, **18**(8), 453-457.

[9] J A Feldman, [1979], High Level Programming for Distributed Computing, *Comm. ACM*, **22**(6), 353-359.

[10] C J Fidge, and R S V Pascoe, [1983], A Comparison of the Concurrency Constructs and Module Facilities of CHILL and Ada, *Australian Computer J.*, **15**(1), 17-27.

[11] N H Gehani, and W D Roome, [1986], Concurrent C, *Software - Practice and Experience*, **16**(9), 821-844.

[12] C A R Hoare, [1978], Communicating Sequential Processes, *Comm. ACM*, **21**(8), 666-677.

[13] C A R Hoare, [1985], *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, NJ.

[14] M E C Hull, [1986], Implementations of the CSP notation for concurrent systems, *Computer J.*, **29**(6), 500-505.

[15] M Jazayeri, [1980], *CSP/80: A Language for Communicating Sequential Processes*, Proc. Fall IEEE COMPCON80, IEEE, NY, 736-740.

[16] G Jones, [1987], *Programming in Occam*, Prentice-Hall, Englewood Cliffs, NJ.

[17] J Kerridge, and D Simpson, [1986], Communicating Parallel Processes, *Software - Practice and Experience*, **16**(1), 63-86.

[18] T W Mao, and R T Yeh, [1980], Communication Ports: A Language Concept for Concurrent Programming, *IEEE Trans. Software Eng.*, **SE-6**(2), 194-204.

[19] D May, [1983], Occam, *Sigplan Notices*, **18**(4), 69-79.

[20] D May, [1987], *Occam 2 Language Definition*, Inmos Ltd., Bristol.

[21] D May, and D E Shepherd, [1987], *Formal Verification of the IMS T800 Microprocessor*, Proc. Electronic Design Automation Conference, Wembley, July.

[22] R Milner, [1989], *Communication and Concurrency*, Prentice-Hall, Englewood Cliffs, NJ.

[23] J Mock, [1990], *Processes, Channels, and Semaphores*, Pixar; in: Logical Systems C for the Transputer, Computer System Architects.

[24] L M Patnaik and B R Badrinath, [1984], Implementation CSP-S for description of distributed algorithms, *Computer Languages*, **9**(3/4) 193-202.

[25] J L Peterson, [1981], *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliffs, N.J.

[26] T J Roper C J and Barter, [1981], A Communicating Sequential Process Language and Implementation, *Software - Practice and Experience*, **11**(11), 1215-1234.

[27] A W Roscoe and C A R Hoare, [1986], *The Laws of Occam Programming*, Oxford University Programming Research Group, **PRG-53**.

[28] A W Roscoe, M H Goldsmith, A D B Cox, and J B Scattergood, [1991], *Formal Methods in the Development of the H1 Transputer*; in: P Welch, D Stiles, T L Kunii, and A Bakkers, *(eds.)*, *Transputing '91*, Proc. World Transputer User Group Conference, IOS Press, Amsterdam, 629-642.

[29] D Shepherd, and G Wilson, [1989], Making Chips that Work, *New Scientist*, **1664**, May, 61-64.

[30] A Silberschatz, [1981], Port Directed Communication, *Computer Journal*, **24**(1), 78-82.

[31] J M Spivey, [1989], *The Z Notation - A Reference Manual*, Prentice-Hall, Englewood Cliffs, NJ.

[32] R Strom, and S Yemini, [1983], NIL: An Integrated Language and System for Distributed Programming, Proc. Symposium on Language Issues in Software Systems, *ACM Sigplan Notices*, **18**(6), 73-82.

[33] J Van den Bos, R Plasmeijer, and J Stroet, [1981], Process Communication Based on Input Specifications, *ACM Trans. Programming Languages and Systems*, **3**(3), 224-250.

[34] J Welsh, and A Lister, [1981], A Comparative Study of Task Communication in Ada, *Software - Practice and Experience*, **11**(3), 257-290.

[35] K L Wrench, [1988], CSP-i: An Implementation of Communicating Sequential Processes, *Software - Practice and Experience*, **18**(6), 545-560.

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as a Communications or Viewpoints. While English is the preferred language of the journal papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted **in triplicate** to the editor.

Form of Manuscript

Manuscripts for review should be prepared according to the following guidelines.

- Use double-space typing on one side only of A4 paper, and provide wide margins.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be on separate sheets of A4 paper, and should be numbered and titled. Figures should be submitted as original line drawings, and not photocopies.
- Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.
- References should be listed at the end of the text **in alphabetic order** of the (first) author's surname, and should be cited in the text in square brackets. References should thus take the following form:

  [1] E Ashcroft and Z Manna, [1972], The translation of 'GOTO' programs to 'WHILE' programs, *Proceedings of IFIP Congress 71*, North-Holland, Amsterdam, 250-255.

  [2] C Bohm and G Jacopini, [1966], Flow diagrams, Turing machines and languages with only two formation rules, *Comm. ACM*, **9**, 366-371.

  [3] S Ginsburg, [1966], *Mathematical theory of context free languages*, McGraw Hill, New York.

Manuscripts *accepted* for publication should comply with the above guidelines, and may be provided in one of the following formats:

- in a **typed form** (i.e. suitable for scanning);
- as an **ASCII file** on diskette; or
- as a **WordPerfect**, **TEX** or **LATEX** or file; or
- in **camera-ready** format.

A page specification is available on request from the editor, for authors wishing to provide camera-ready copies. A styles file is available from the editor for Wordperfect, **TEX** or **LATEX** documents.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R20-00 per final page for camera-ready contributions and the maximum is R100-00 per page for contributions in typed format.

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

South African Computer Journal

Number 6, March 1992
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 6, Maart 1992
ISSN 1015-7999

Contents