J. M Bishop
24/3/91

South African Computer Journal
Number 4
March 1991

Suid-Afrikaanse Rekenaar-tydskrif
Nommer 4
Maart 1991

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

Assistant Editor: Information Systems
Dr Peter Lay
P. O. Box 2142
Windmeul 7630

Subscriptions

	Annual	Single copy
Southern Africa:	R32-00	R8-00
Elsewhere:	$32-00	$8-00

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

Computer Science Department, University of Pretoria, Pretoria 0002
Phone: (012) 420-2843 Fax: (int) + (012) + 43-6454

Sigma Press

Guest Editorial

Does Today's Industry Need Qualified Computer Scientists?

This guest editorial consists of two contrasting views on the value to industry of a professional degree in computer science. Both authors, one local and one from Germany, are managing directors of well-respected software houses. (Editor)

Viewpoint I

Hans G Steiner
MBP Software and Systems GMBH
Semerteichestrasse 47-49
D4600 Dortmund 1

I would like to begin by recounting from my student days a story that I consider to be relevant. While attending a career forum for computer scientists, mathematicians and physicists, the personnel officer from IBM Germany was asked if he would consider taking on mathematicians. The gist of his answer was as follows: "Of course I must admit that I could just as well give the mathematician's job to a theologian. What is important is the ability to think logically. It is only there, on the job, that he learns how to become productive for us."

This episode occurred 14 years ago at a time when graduating mathematicians did not necessarily learn programming and when computer scientists were few and far between. The situation has improved immensely since then. Mechanical engineers, electrical engineers and physicists, all with programming knowledge, have for the most part taken over many programming jobs. This shows industry that, as time goes by, the answer to the opening question is becoming an ever-louder and more frequent "NO".

I support this opinion and in the remainder of this essay I will expand on my reasons, as well as highlight some exceptions.

An employee who is recruited directly from a university should possess the following four capabilities:
1. An ability to think logically: One of the basic requirements in our business is the ability to recognise, analyze, structure, break down and solve a problem as well as to fully synthesize the solution. The important thing is to break down the problem in such a way that the individual components can feasibly be solved. This is what distinguishes an engineer/scientist from an arts scholar. The latter usually concentrates on the complete problem and tends to settle for a contentious, complex and partially non-feasible solution. In our business, it is not enough to merely ask the "right" questions.

This ability to think logically may be accentuated in computer science; however engineers/scientists will generally possess the ability to an equal extent.

2. Programming skills: Our employees' prime tool of the trade is their ability to encode solutions to problems. Ideally, this ability ought to be held as abstract as possible. In other words, the further away from the "bit", the better. FORTRAN programmers who, for example, concentrate on the multiple use of memory space of all variables will never be successful programmers in an object-oriented programming language.

The difference can be seen, even in today's universities. For example, one only has to read a PROLOG program from a student who learned PASCAL in his first semester and PROLOG in his fifth. On average, this is always a "PASCAL program in PROLOG". The various possibilities offered by a predicate calculus language are only recognised and used by the best students. Again, we do not need the average computer science scholar who has spent between six and eight years writing complicated PASCAL programs, but rather the "thinker" with basic programming knowledge who is capable of abstracting the task. Once again, the ability is independent of faculty.

3. Teamwork skills: Working successfully in a team

This SACJ issue is sponsored by
Department of Computer Science
Rhodes University

requires assertiveness, tolerance, stability and one's own ideas. Very few problems have solutions that can be managed by one person successfully in the allocated time. Out of 700 employees, we can only afford approximately five "lone warriors" who are, in turn, the leading specialists in a wide field. They have a strategic vision which we follow. All remaining employees are evaluated, for better or worse, on their team performance. Some people have an in-built ability to work in teams. A few universities - unfortunately not enough - encourage this team-thinking. Again we see that the ability is independent of university faculty.

4. *Motivation:* The ability to enjoy one's particular job is a major driving force in every employee. Whereas in the sixties everything had to be "bigger, faster and better" and in the eighties "things had to be meaningful to society", the theme for the nineties is self-realization. Those companies who succeed in incorporating different employees (ie employees with different driving forces) into the company culture and who motivate each employee optimally will be successful in the nineties and beyond. There are huge productivity gains to be had from motivating employees. Compared with this, the possibilities offered by CASE tools pale into insignificance.

One basic requirement is thus the recruitment of a self-motivated employee who should at no stage become demotivated, whether it be by company culture, superiors or working conditions.

Again, this is not linked to a specific university faculty and is independent of know-how.

As none of these four capabilities are necessarily restricted to studies in computer science, the technical/-scientific background of new employees who are being recruited is largely irrelevant.

I would now like to point out a few exceptions which might give a computer scientist the upper hand in an interview. I refer exclusively to our own company and our specific company tasks.

1. Porting our COBOL Compiler onto the latest UNIX machine from the manufacturer XY. Knowledge of the UNIX operating systems could be very valuable and enable the new employee to rapidly become productive.
2. Programming the 37th interface (special customer request) for our ISDN card. Knowledge of interface protocols or experience with protocol conversions would be very useful and could be a decisive factor. Such specialized knowledge is usually very rare.
3. Adapting our integrated office automation system to the 17th foreign language. The employee must command the language perfectly. Simply outsourcing the translation would mean that this language version could not be maintained or supported. From this example one can see that specialized knowledge not only refers to knowledge gained from computer science studies.

In the product business, it sometimes happens that computer scientists with specialized knowledge are sought. (This is almost impossible in the project business, due to the variety of tasks to be performed.) However such a "knowledge" advantage over others usually only lasts about a year. After that, the achievements of two different employees (one with specialized knowledge and the other without) tends to even out.

Most applicants who start out do not know our products, as the flow of employees in this industry is almost always from manufacturer to user. Hardware and software manufacturers often lose their products specialist to the products' users. Seldom do employees change in the other direction.

In my opinion, universities can learn two things from this essay:

1. Studies in computer science give basic knowledge that can be used in various jobs. The student should however be careful not to place all his eggs in one basket.
2. Teamwork should be encouraged more. Time allows for very few geniuses, acting as 'lone warriors', to initiate progress in our society.

I have taken the liberty of basing my interpretation and answer to the opening question on my own judgement and experiences. I would be grateful for other opinions and experiences on this topic.

I would like to conclude by expressing my gratitude for having had this opportunity to express my views.

Viewpoint II

Pierre Visser
Grinaker Informatics, P.O.Box 29818, Sunnyside, 0132

The title question currently generates as many viewpoints as a counterpart question: "What is the correct curriculum for a computer science qualification?" Such questions stem from the many and diverse requirements expected to be fulfilled by the still developing applied science. A basic assumption of this editorial is that we need to have an explosion and consolidation in computer science theory. Only after this has occurred will a more general consensus of opinion exist - as is the case in other matured sciences.

An argument is presented here for the current approach of striving towards a balance between immediate industry needs and long term perceived theoretical requirements of industry, even though the balance, as viewed from either side, will always be imperfect.

Industry can, of course, do without qualified computer scientists - that is how it was established. Dedicated mathematicians, physicists, engineers and other scientists will, as in the past, continue to effect improvements. However, as one of those scientists from the

early days, it is difficult for me to understand why one would choose to continue this way.

A computer science qualification is viewed here as a university education (4 years) into theory that is not obtainable otherwise. By definition, therefore, a qualified computer scientist is not trained to conform to specific job requirements. Rather, the computer scientist will possess knowledge that will serve him long past the present day's computing technology.

Whether industry needs qualified computer scientists depends on two issues. Firstly, can an education be provided for computing technology that will serve as a foundation for the student's next 45 years in industry; and secondly, can industry build upon this foundation to created wealth more effectively than without qualified computer scientists.

It is widely accepted that, in broad terms, the teaching of fundamental theory will serve the first purpose. However, what subject matter to include from the wealth of mathematics, physics, OR, and from computing fields such as networking, operating systems and others, remains the illusive issue. Universities can merely strive to select the right mix for the perceived future needs of industry. This requires insight into the evolution of computing technology. I will later discuss such insight as a basic requirement for a qualified computer scientist.

What is important in teaching is to focus on fundamental theory. Just as the natural science student needs to breed fruit flies in order to gain insight into the dynamics of inheritance, so too the computer science student needs to develop software. The purpose should be to create understanding and insight into fundamental theory, and, just as in the case of the breeder of fruit flies, the software developed should never be measured against efficiency requirements from industry.

The second issue is whether industry can build on this theoretical foundation to create wealth.

A depth of insight into computing technology, more so than with other training, can be identified as the focus of the potential value of a qualified computer scientist to industry. Three areas which require such insight are discussed below, namely organisation, product definition and the application of new computing technology in industry.

Computing products form an integral part of an organisation, and represent a significant capital investment aimed at increasing efficiency. These products are incorporated in an evolutionary way to match changing organisational requirements with improving product capabilities. Decisions to use products determine the long term efficiency and cost-effective replacement. Such decisions require insight into computing technology and its evolution. A qualified computer scientist can improve such decisions only if he gains enough insight into computing technology as well as its interaction with business through years of practice.

The success of products in some areas is dependent on market requirements which depend on computing technology and its evolution. The correct definition of characteristics of products that interface to computers is such an example. Insight into computing technology is able to create the versatility, simplicity or other improved selling features which can open new market segments.

The third area where insight into computing technology plays an important role is in the application of new computing technology (or a new trend) in an organisation. Examples include the introductory period for networking, DBMS-technology, distributed processing and document image processing. In areas such as these, the newly qualified computer scientist can be applied effectively and at the same time build up insight through experience which he will require for the other areas of organisational and product decisions mentioned above.

A major dilemma in the continuous development of insight into computing technology by qualified computer scientists is their correct application in industry. The identification of the opportunities within the three areas discussed above, requires insight into computing technology itself. Winning companies that depend on computing technology have this ability. In such companies the insight of the qualified computer scientist into computing technology as well as its contribution to the business is constantly stimulated, turning the qualified computer scientist into a valuable company resource.

What has been neglected in this whole discussion is the role of the "technician" and of the casual user of computing technology. Such personnel are required to implement selected computing technology of the day efficiently, whether in accounting, chemical engineering or other specialised disciplines. Their role and place is unquestioned. However, it cannot be expected of them to evaluate the potential of new computing technology, formulate algorithms from fundamental theory or any such decisions which require insight built upon a sound theoretical knowledge of the field.

The final aspect in answering the opening question is whether the qualified computer scientist can outperform other professionals who build up their own experience in computing technology. Many examples could be cited of improvement brought about by non-computer scientists in the past. However, these individuals formed part of the bootstrapping for computer science theory and education. We should have faith in this bootstrapping of computer science qualifications, because computing technology will increasingly diversify into many directions of specialisation in years to come, each requiring a body of fundamental theory.

This complexity cannot be left to a casual development of insight - industry requires qualified computer scientists to experience interaction with business objectives in order to cope successfully with future computing technology.

An Interrupt Driven Paradigm of Concurrent Programming

Peter G Clayton
Department of Computer Science, Rhodes University,
P O Box 94, Grahamstown, 6140 RSA

Abstract

This paper introduces a class of concurrent programming construct based on an interrupt driven paradigm. This programming model is designed to simplify the application of conventional imperative programming techniques to distributed parallel processing environments in which processors have no physical memory in common. The paper uses the notation of CSP to demonstrate a theoretical basis and identify safety properties for this approach, and describes the semantics and application of an actual programming construct which falls into this class.
Keywords: *Concurrent notations, parallel processing, distributed processing.*
Computing Review Category: *D.3.3*

Presented at Vth S.A. Computer Symposium

1. Introduction

Since the appearance of VLSI computing devices which are affordable in large numbers, cost effective parallel processing hardware designs have tended towards systems constructed from processors which have local memory only, and which communicate with each other via high speed interconnection networks[I]. Instead of reading and writing shared variables, the concurrent processes of programs which execute on such systems communicate and synchronize with each other using message passing. The notations put forward in CSP [8] and DP [4] reflect this type of multiprocessor architecture, as do languages influenced by them such as occam [13] and *MOD[II] [6][III].

Effective software development tools which are able to cope with distributed parallel processing networks (in which processors have no memory in common) have lagged behind innovative hardware developments. In particular, inadequate progress has been made towards improving the extent to which the program suits the solution it represents and the ease with which the programmer is able to express the solution using the facilities of the programming language. In their endeavour to narrow the semantic gap between the language and its host computer, and thereby increase the efficiency of the implementation, the designers of languages like occam and *MOD have increased the semantic gap between the programming language and its user by supplying only low level primitives for communication. This often results in references to passive data items in a solution having to be mapped by the programmer onto sets of message exchanges. If not done extremely carefully, this transformation can be the source of abstruse run-time errors. Whereas message-oriented languages are well suited for programming pipelined computations, shared variable languages are far better suited for programming the large class of client/server systems [2].

The construct described in this paper is aimed at simplifying the use of imperative programming techniques for distributed processing. Because of its correspondence with human behaviour, an interrupt driven paradigm has been chosen as the principal feature of the notation. A high level interrupt is intended to provide an atomic language structure for communication and condition synchronization between concurrent processes, and an alternative mechanism to polling.

2. An interrupt-generating variable

The use of interrupts is a well established principle of hardware interfacing and low level computing. Allowing an interface to inform the computer when it is ready to transfer data is an efficient and intuitive alternative to the CPU constantly monitoring a status register. In contrast, the regular monitoring of a status value is a common characteristic of the control structures of high level imperative programming languages. The designers of structured imperative languages have been unable to incorporate support for interrupts at the application program level because the conventional set of control structures for these languages has been designed uncompromisingly around environments for sequential execution[IV]. Interrupts are only possible if two entities are active simultaneously, one to perform the task which might be interrupted, and the other to produce the interrupt signal. Concurrent programming languages afford program segments the

potential for executing in parallel; consequently, the support for interrupts between high level program segments is feasible.

A second feature, needed to reduce the programming effort for concurrent solutions which are naturally expressed using shared variables, is a mechanism for sharing items of data between concurrent processes of the application program. The shared data mechanism is combined with the interrupt driven paradigm to produce the concept of an *interrupt-generating variable*, a feature for combining a shared data facility with condition synchronization between concurrent processes. This concept is deliberately intended to resemble a simple form of human *pop-up* memory; this special type of variable is expected to cause an interrupt to be generated when its stored value corresponds to a value of interest to a process of the application program. The proposal includes the migration of the interrupt mechanism to the system level, so that the interrupt-generating variable can be regarded as a conventional variable for use in a control structure which adopts the interrupt driven paradigm. System processes monitor particular run-time values, and send interrupt messages to processes declared in the application program when conditions of interest have been satisfied.

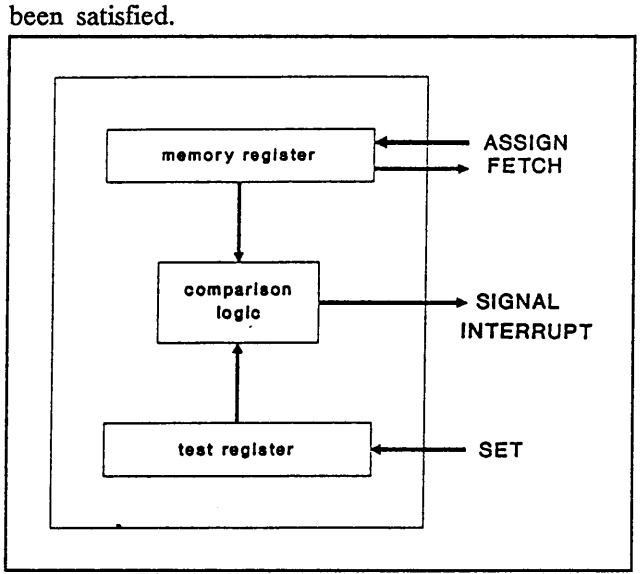

Figure 1. Block diagram of a system process to emulate an interrupt-generating variable.

Figure 1 illustrates the functional components of an interrupt-generating process which emulates a shared variable. As well as the standard *memory* register, this model incorporates a *test* register and a comparison mechanism. The comparison mechanism is able to perform a bit-wise comparison of the *memory* and *test* registers and generate an interrupt signal if they are equal. The external signals depicted in figure 1 are events which represent both control signals and the movement of data, and are best viewed as synchronous messages. *ASSIGN* and *FETCH* represent the standard operations of assigning values to - and fetching values from *memory*. The *SET* message enables a value to be recorded in the *test* register of the emulated variable. The comparison mechanism is triggered automatically after every *ASSIGN* event (which alters the *memory* register), to establish whether the variable has assumed the particular *test* value of interest, and generate an interrupt signal if it has.

The interrupt-generating process exists at the same atomic level of language construct as the Ada exception [1], but is intended to be a more general facility. In contrast, the object-oriented languages (particularly the Actor based languages [11]), which provide even more widely applicable objects, require explicit programming of the object's methods and safety properties.

An interrupt-generating variable's operation includes the prevention of interference between references made to it by several concurrent application processes. In the model presented later in the paper, this is achieved using synchronized communication within a CSP [9] form of alternative structure. This facility eliminates the need for the application programmer to write code to ensure exclusive access of the shared item. Such code would normally be executed for every reference to the item even if attempts for simultaneous access were infrequent.

As an additional precaution against interference, interrupt-generating variables are limited to performing condition evaluation for only one process. This safety property is formulated in terms of a restriction preventing several processes from assuming the same state simultaneously, a frequently used principle in concurrent programming [15]. In the text which follows, this restriction is referred to as the *ownership rule* of interrupt-generating variables.

3. A formal model of an interrupt-generating variable

The approach of CSP [9] is used as a formal notation for defining a process which may be used as a substitute for shared memory, and which makes use of interrupts to synchronize with other processes on conditional events. The formal notation is used to illustrate the need to introduce a safety requirement which restricts each memory emulation process to servicing only one interrupt condition.

To introduce the formal notation, consider the definition of a conventional high level language variable. If the current value stored in an integer variable is not of interest, the simple alphabet of events can ignore the distinction between states which

the variable may assume, and consider only the actions of making references to the variable.

$$\alpha\ INTEGER = \{assign, fetch\}$$
$$INTEGER = (assign \rightarrow \mu\ X.(assign \rightarrow X \ [] \ fetch \rightarrow X))$$

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 to describe all possible traces of events that the variable may engage in.

Using a local variable m to represent the variable's *memory* register, this specification can be written in the form of a process which communicates with the rest of the system by passing messages[v]. The events which cause a value to be transferred to or from the variable are equivalent to synchronous communication primitives.

$$\alpha\ INTEGER = \{assign.k\ ,\ fetch.k\}$$
$$INTEGER = (assign\ ?\ m \rightarrow \mu\ X.(assign\ ?\ m \rightarrow X \ [] \ fetch\ !\ m \rightarrow X))$$

An interrupt-generating process may be superimposed upon this simple variable definition to describe the behaviour of the enhanced interrupt-generating variable of figure 1. The process which emulates such a variable and which is referenced by one application process can be expressed using the alphabet of events:

$$\alpha\ VARIABLE = \{assign.k, fetch.k, set.k, interrupt.k\}$$

Using the local variables t and m to represent the *test* and *memory* registers of the variable, and the constant *any* to represent an arbitrary value, the behaviour of the process to emulate an interrupt-generating variable can be specified as:

$$VARIABLE = (set\ ?\ t \rightarrow VAR)$$

where $VAR = (assign\ ?\ m \rightarrow TEST \ [] \ fetch\ !\ m \rightarrow VAR \ [] \ set\ ?\ t \rightarrow VAR)$

and $TEST =$ **if** $m = t$ **then** $\mu\ X.(interrupt\ !\ any \rightarrow VAR \ [] \ fetch\ !\ m \rightarrow X)$ **else** VAR

(S-1)

This model ensures that the variable's *test* register is set before the *memory* register can be used. Once the *memory* register of the variable has been assigned a value which equals the *test* register, the process referencing the variable may not assign another value to the variable without first accepting an interrupt signal. The set of legal traces for specification S-1 excludes this possibility. On the other hand, a process may fetch the current value of the variable any number of times between assigning a value and accepting an interrupt signal, or between two successive assignments.

Specification S-1 provides the interrupt-generating mechanism but not the shared variable facility. Before introducing an example, S-1 will be enhanced to provide for shared access. The interaction between a process which emulates an interrupt-generating variable, and a number of concurrent application processes which communicate with it simultaneously can be specified as follows.

$$\alpha\ VARIABLE = \cup\ \alpha\ VAR_i \qquad for\ 1 \le i \le n$$
$$\alpha\ VAR_i = \{assign_j.k, fetch_j.k, set_i.k, interrupt_i.k\} \qquad for\ 1 \le j \le n$$

$$VARIABLE = ((i: 1..n)set_i\ ?\ t \rightarrow VAR_i)$$

where

$$VAR_i = ((j: 1..n)assign_j\ ?\ m \rightarrow TEST_i \ [] \ (j: 1..n)fetch_j\ !\ m \rightarrow VAR_i \ [] \ set_i\ ?\ t \rightarrow VAR_i) \qquad for\ 1 \le i \le n$$

and

$$TEST_i = \textbf{if}\ m = t\ \textbf{then}\ \mu\ X.(interrupt_i\ !\ any \rightarrow VAR_i \ [] \ (j: 1..n)fetch_j\ !\ m \rightarrow X)\ \textbf{else}\ VAR_i \qquad for\ 1 \le i \le n$$

(S-2)

This model allows only one application process to set the *test* register, thereby claiming ownership of the variable, whereafter only that process may be interrupted. The other processes may alter or fetch the *memory* register value, but may not set the *test* register or respond to an interrupt signal. The other processes are also prevented from altering the *memory* register between the signalling of an interrupt and the acceptance of this signal by the owner process.

Since specification S-2 is at the heart of the concept introduced by this paper, a simple example of a shared interrupt-generating variable is presented at this point to illustrate its operation. The following pseudo-code shows a process to control a stack for buffering items produced by another piece of code which executes concurrently with the stack process. It is possible (though not good programming practice as far as object-oriented programming is concerned)

to make the stack pointer both visible to the stack process and useful in synchronizing the producer's activities, by declaring a shared variable for use in an interrupt driven structure.

```
Variable integer top :  {declare a shared variable}

PROCESS WAREHOUSE    {control the stack}
  {Declare the stack to have stack_max elements}
  Loop forever
    Either
      synchronize with the producer
        Top <- top + 1
        Receive an item into stack[top]
    Or
      synchronize with a consumer
        Supply an item from stack [top]
        Top <- top - 1
  end of loop

PROCESS PRODUCER
  Loop forever
    Manufacture an item
    Synchronize with the warehouse and
       supply an item
    when top = stack_max
      wait
  end of loop
```

In this example, both processes have access to variable *top*, even if they are running on separate processors. The producer process is given exclusive ownership (described more fully later) enabling the interrupt-generating mechanism of *top* to delay the production of items while the condition *top* = *stack_max* is true. Because of the presence of the interrupt-generating mechanism, there is no need for the producer to test the value of *stack_max* after each item has been manufactured. A more complete version of this example is presented in section 5.1.

To illustrate the need for exclusive ownership of an interrupt-generating process, a two-process example is introduced. It is possible to allow an application process *P2* to be interrupted once another application process *P1* has secured ownership of the interrupt-generating process as long as the two processes agree on the value to be placed in the *test* register. In the specification which follows, *p:(1..2)* is a cardinal value which represents the number of application processes which can be interrupted.

$$VARIABLE = p := 1 \rightarrow (set_1\ ?\ t \rightarrow VAR_1 \ \square\ set_2\ ?\ t \rightarrow VAR_2\ )$$

where

$$\begin{aligned} VAR_1 = (&assign_1\ ?\ m \rightarrow TEST_1 \\ \square\ &assign_2\ ?\ m \rightarrow TEST_1 \\ \square\ &fetch_1\ !\ m \rightarrow VAR_1 \\ \square\ &fetch_2\ !\ m \rightarrow VAR_1 \\ \square\ &set_1\ ?\ t \rightarrow UNIQUE_1 \\ \square\ &set_2\ ?\ y \rightarrow IDENTICAL_1\ ) \end{aligned}$$

$$\begin{aligned} UNIQUE_1 = \ &\textbf{if } p = 1 \\ &\textbf{then } VAR_1 \\ &\textbf{else } STOP \text{ \{cannot reassign interrupt value\}} \end{aligned}$$

$$\begin{aligned} IDENTICAL_1 = \ &\textbf{if } y = t \\ &\textbf{then } p := 2 \rightarrow VAR_1 \\ &\textbf{else } STOP \text{ \{P1 and P2 disagree on the value of t\}} \end{aligned}$$

$$\begin{aligned} TEST_1 = \ &\textbf{if } m = t \\ &\textbf{then if } p = 2 \\ &\quad \textbf{then } (SENDP_1 \parallel SENDP_2) \rightarrow VAR_1 \\ &\quad \textbf{else } SENDP_1 \rightarrow VAR_1 \\ &\textbf{else } VAR_1 \end{aligned}$$

$$SENDP_i = (interrupt_i\ !\ any \rightarrow SKIP \ \square\ fetch_1\ !\ m \rightarrow SENDP_i \ \square\ fetch_2\ !\ m \rightarrow SENDP_i\ ) \quad for\ 1 \le i \le 2$$

and VAR_2, $TEST_2$, $UNIQUE_2$ and $IDENTICAL_2$ are similar with all 1's replaced by 2's and vice versa.

If *P1* re-assigns the *test* register after *P2* has requested an interrupt by setting the *test* register, the process will *STOP* (this might take the form of reporting a run-time error and aborting) to avoid the possibility of an incorrect interrupt signal. A similar situation arises if *P2* attempts to set the *test* register to a value which is distinct from the one assigned to it by *P1*. These conditions represent conflicts which result in deadlock. Situations such as this cannot always be predicted by the application programmer, and it is not possible to exclude them from the design of a programming notation. However, the elimination of these sources of deadlock at the system level eradicates them from the higher levels as well. The price of facilitating the correct execution of high level languages is the imposition of the exclusive ownership property of interrupt-generating processes.

To release an interrupt-generating process from the clutches of an application process which has claimed ownership of it, it is necessary to include a facility for resetting the interrupt-generating process once it is no longer needed. Specification S-2 should be modified (not shown in this paper due to the space limitation) to take account of this.

4. Sequential processes and the interrupt-generating model

For a sequential process to interact with the interrupt-generating model of the previous section, it

is necessary to allow an interrupt to occur as an alternative event to every statement process at the most primitive level. It is convenient to design the control structure in the form of an infinite loop. Successful termination of the loop is specified as an action which results from the receipt of an interrupt signal.

$\alpha\ LOOP = \cup\, \alpha\ STATEMENT_i \qquad for\ 1 \le i \le n$
$\alpha\ STATEMENT_i = \{statement_i,\ interrupt,\ action\}$

$LOOP = STATEMENT_1$
$STATEMENT_i = (statement_i \rightarrow STATEMENT_{i+1}$
$\quad [] \ interrupt \rightarrow$
$\quad\quad (action \rightarrow STATEMENT_i$
$\quad\quad [] \ action \rightarrow SKIP))$
$\quad\quad\quad for\ 1 \le i < n$
$STATEMENT_n = (statement_n \rightarrow STATEMENT_1$
$\quad [] \ interrupt \rightarrow$
$\quad\quad (action \rightarrow STATEMENT_n$
$\quad\quad [] \ action \rightarrow SKIP))$

(S-3)

In this model, the action occurring as a result of an interrupt event may take the form of a subroutine, or a sequence of statements which result in the successful termination of the loop (the *SKIP* processes in the specification represent successful termination of $STATEMENT_i$ or $STATEMENT_n$, and consequently of the loop). These two possibilities have been represented as distinct alternatives. Specification S-3 excludes the evaluation of the test condition for determining the point of termination. Typically, this would be done by a second process executing in parallel, which would cause process *LOOP* to be interrupted. However, there is no reason why one of the statements in S-3 should not perform this evaluation. In this case, a language implementation should be able to avoid an unnecessary message passing overhead by transforming the message passing primitives required to implement the interrupt, into simple memory references equivalent to those required for a conventional sequential condition test.

An application process comprising n statements may interact with m interrupt-generating processes by expanding the choice of interrupt events in S-3. In the specification below, the interrupt events have been represented as input processes to accentuate the nature of the input signals.

$\alpha\ LOOP = \cup\, \alpha\ STATEMENT_i \qquad for\ 1 \le i \le n$
$\alpha\ STATEMENT_i = \{statement_i,\ interrupt_j k,$
$\quad\quad action_j\} \qquad for\ 1 \le j \le m$

$LOOP = STATEMENT_1$

$STATEMENT_i = (statement_i \rightarrow STATEMENT_{i+1})$
$\quad [] \ (j: 1..m) interrupt_j\ ?\ any \rightarrow$
$\quad\quad (action_j \rightarrow STATEMENT_i$
$\quad\quad [] \ action_j \rightarrow SKIP))$
$\quad\quad\quad for\ 1 \le i < n$
$STATEMENT_n = (statement_n \rightarrow STATEMENT_1$
$\quad [] \ (j: 1..m) interrupt_j\ ?\ any \rightarrow$
$\quad\quad (action_j \rightarrow STATEMENT_n$
$\quad\quad [] \ action_j \rightarrow SKIP))$

(S-4)

Interrupt events are associated with nested processes according to the rule:

$((e_1 \rightarrow P \ [] \ i_1 \rightarrow P) \ [] \ i_2 \rightarrow P)$
$= \ (e_1 \rightarrow P \ [] \ i_1 \rightarrow P \ [] \ i_2 \rightarrow P)$

where e_1 is an event with i_1 as an alternative, and i_2 is a higher order event.

Finally, it is frequently desirable to regard a sequence of events as being an atomic operation. A critical region and the *P* operation on a semaphore are examples. To simulate loops in which an inseparable sequence of actions must be completed before an interrupt is serviced, it is necessary to specify a means of delaying the interrupt until the desired action is completed.

$\alpha\ LOOP = \cup\, \alpha\ STATEMENT_i \qquad for\ 1 \le i \le n$
$\alpha\ STATEMENT_i = \{statement_i,\ interrupt,\ action\}$

$LOOP = STATEMENT_1$
$STATEMENT_i = (statement_i \rightarrow STATEMENT_{i+1})$
$\quad\quad\quad for\ 1 \le i < n$
$STATEMENT_n = (statement_n \rightarrow STATEMENT_1$
$\quad [] \ interrupt \rightarrow statement_n \rightarrow$
$\quad\quad (action \rightarrow STATEMENT_1$
$\quad\quad [] \ action \rightarrow SKIP))$

(S-5)

The delaying action of S-5 should be implemented in such a way as to enable an interrupt condition to be noted sufficiently promptly to avoid further program execution altering the circumstances which gave rise to the condition. The interrupt-generating model of specification S-2 avoids the potential loss of information by insisting that an interrupt it has signalled should be serviced before its register contents may be altered. This works reasonably for other parallel processes, but may lead to deadlock with the process which has claimed ownership of the interrupt-generating process. For example, when a process P_1

$P_1 = (set_1\ !\ k \rightarrow LOOP)$

$LOOP = (x \rightarrow assign_1 \,!\, k \rightarrow$
$(x \rightarrow LOOP$
$[]\ interrupt_1 \,?\, any \rightarrow SKIP))$

$\exists\ assign_1 k \in \alpha\ LOOP$ and $x \in \alpha\ LOOP$

is executing in parallel with the process from specification S-2

$$(P_1 \quad \| \quad VARIABLE)$$

the possibility exists for the *VARIABLE* to signal an interrupt before P_1 has reached the $assign_1\ !\ k$ process. P_1 then refuses to acknowledge the interrupt until the assignment has been received; while *VARIABLE* refuses to service the assignment until the interrupt has been acknowledged. To prevent this situation arising, the implementation must make provision for the interrupt to be noted, but only acted on later. For example, a dummy service routine may be used to note the interrupt, although the execution of the action code is delayed as in specification S-5.

5. Use of the interrupt driven paradigm in the programming language HUL

An experimental programming language called HUL[vi] [5] has been developed to investigate the interrupt driven paradigm, and is intended for execution in a highly concurrent environment. The notation has two additional objectives. One is the isolation of the concurrent source language from hardware considerations in distributed parallel processing environments. The other is ease of use.

Only the interrupt driven aspect of HUL is discussed in this paper. The other features of HUL have been based primarily on occam [12] and CSP [8]. HUL has been designed for the purpose of experimenting with control structures, and is not intended to be a rival for occam or Ada.

The primary control structure proposed in HUL is based on a common feature of human communication, the use of implied repetition in conjunction with interruption. *Move forward until you reach the target* implies a repeated *move forward* process. To emphasize the interrupt feature, the command can be rephrased as

Move forward.
When you reach the target
stop.

To encourage the use of concurrency, implied parallelism is a further feature adopted by HUL's human-like control structure. Unless a particular succession of component processes is explicitly stated, the construct assumes that the programmer wants all component processes executed simultaneously, or if insufficient processors are available for this to be possible, that the order in which they are executed is not important. HUL uses the reserved word *do* as a leading symbol for introducing this human-like construct as its primary control structure:

```
Do
  Move_forward
  Swing_your_arms
  When you_reach_the_target
    stop
```

The code to evaluate the condition for terminating this infinite loop is placed outside the bounds of the loop by implementing *you_reach_the_target* as an interrupt-generating Boolean variable. In this way, HUL eliminates condition evaluation on each pass through a loop.

5.1 The DO control structure

The *do* control structure is the primary structuring mechanism of HUL. By allowing a set of counters and qualifiers to be coupled to it, the *do* control structure may be extended to construct all loops and single iteration sequences, and to specify parallel and sequential execution. The essential syntactical alternatives of the *do* control structure are given by the following extended BNF description.

do = **do** [**intact**] [*count*] [*qualifier*]
process-body

count = **once** | **twice** | *simple-value* **times**

qualifier = **in parallel** | **in sequence** | **parallel** | **sequence** | **par** | **seq**

process-body = *statement*
{ *statement* }
{ *when-statement* }

when-statement = **when** (*variable* = *expression*
action
| [**not**] *boolean-variable*)

action = **wait**
| { *primitive* }
[**stop**]

An unqualified *do* statement presumes that the programmer wishes its component processes to be

executed in parallel, or if insufficient processors are available for this to be possible, that the order in which they are executed is not significant. The execution of each component process is repeated forever, with initial synchronization of the component processes occurring between each pass of the structure. Accordingly

```
do        implies        do forever in parallel
```

Counters and sequence qualifiers may be included to restrict the number of times that the process body is executed, or impose a sequential succession on the execution of the specified processes. Examples are:

```
Do once in sequence      Do twice          Do hour times in sequence
  Keyboard ? char          Write ["do"]      Chime
  Value <- char - 48                         Reset_the_chime
```

One or more *when* statements may be tagged onto the process body of the *do* control structure. Each *when* statement sets up an interrupt condition supported by an interrupt-generating variable. Implementation is such that the state of the interrupt-generating variable is not examined on each pass of the loop. When the interrupt-generating variable is assigned a value which agrees with the value of the interrupt condition which it supports, the loop body is automatically suspended and the appropriate programmer defined action is taken. Some examples are:

```
When store_empty                          When seconds = 60
  wait until item_count = 1                 minutes <- minutes + 1

When time_up
  stop {stop terminates the control structure, not the program}
```

A more accurate HUL version of the stack example, presented earlier in the paper (section 3), can now be given.

```
Variable integer top :  {shared interrupt-generating variable}

Procedure warehouse =
  Variable character stack [stack_max] :
  Do  {to repeat the if construct}
    If
      request ready
        Request receive message
        Consumer send stack [top]
        Top <- top - 1
      producer ready    {test for a synchronous message}
        Top <- top + 1
        Producer receive stack [top]  {synchronize with producer} :

Procedure producer_process =
  Variable character item :
  Do in sequence
    {manufacture character item}
    Producer send item   {synchronize with warehouse}
    when top = stack_max
      wait    :
```

Wait suspends its parent process until the interrupt condition is no longer true, whereafter the producer loop resumes, unhindered by the *top* = *stack_max* test.

Only one interrupt action may be executed at a time should interrupts occur simultaneously in a process which has a number of interrupt conditions. The interrupt action sequences are regarded as indivisible, and so cannot be interrupted by further interrupt signals. The order in which interrupt condition values are evaluated and sent to the interrupt-generating variables is non-deterministic, which prevents the programmer from assuming a particular order of execution. The Petri net model [14] of figure 2 shows exclusive execution of the interrupt actions for the sequential loop listed below, which has two interrupt conditions.

```
do in sequence
  { statement sequence }
  when x = a
    { action }
  when y = b
    { action }
```

An *intact* qualifier may be used in conjunction with the *do* control structure to ensure that the current iteration of the structure will execute to completion before any interrupt action is taken. This option only has meaning in a control structure which includes a *when* statement, or in a component process of such a structure which might be aborted by the premature termination of the structure.

In the example below, a sequential loop to cycle through the processes which control the lights on a traffic signal (the parameter [*i*] of the light switching processes indicates the signal number) has had a *when* statement added to its process body to show how an interrupt-generating Boolean variable *no_cars_allowed* can be used to switch the traffic signal permanently to red. The *intact* qualifier guarantees that a green-amber-red cycle is completed, to avoid the possibility of a direct change from green to red, or of two lights being switched on simultaneously.

```
do intact in sequence
  green[i]
  amber[i]
  red[i]
  when no_cars_allowed
    stay_red[i] {switch the red light on and leave it on}
```

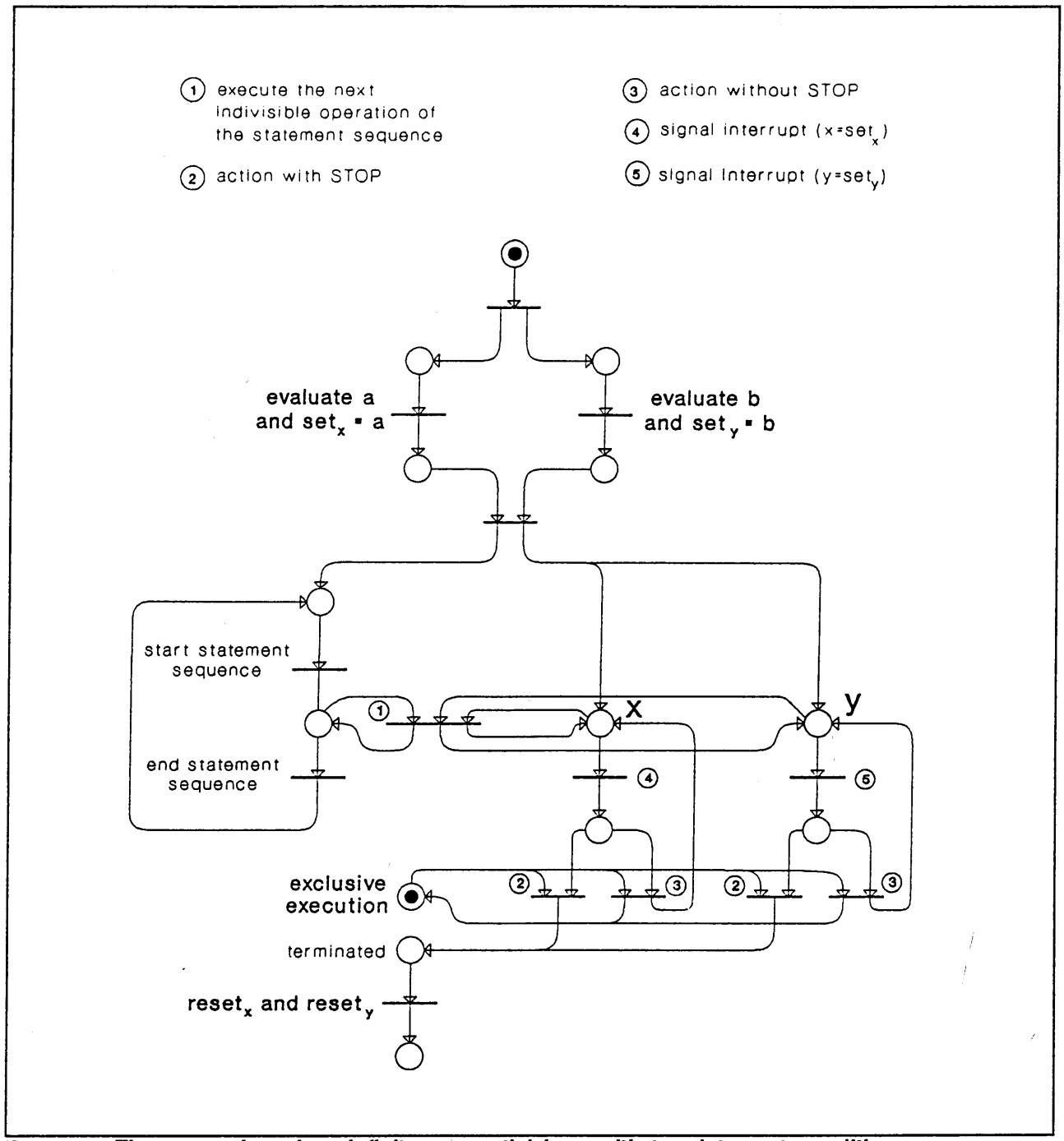

Figure 2. The semantics of an infinite sequential loop with two interrupt conditions.

Formally, the semantic effect of the *intact* qualifier can be described as in specification S-5. Any interrupt action is deferred until the end of each iteration. The process to be interrupted shares the same processor as the interrupt action sequences, but is given initial exclusive execution privilege. The Petri net model of figure 2, modified for *intact* execution, is listed below, and the semantics which this modification directs are shown in figure 3. In this Petri net, the process body of the control structure ensures its own exclusive execution by providing the token for the *exclusive execution* (of interrupt actions) place at the end of an iteration.

```
do intact in sequence
  { statement sequence }
  when x = a
    { action }
  when y = b
    { action }
```

The *intact* option is used with parallel control structures to produce the same form of deferment as for the sequential case. The action of any interrupt is deferred until the last concurrent sub-process has executed to completion, within a particular iteration.

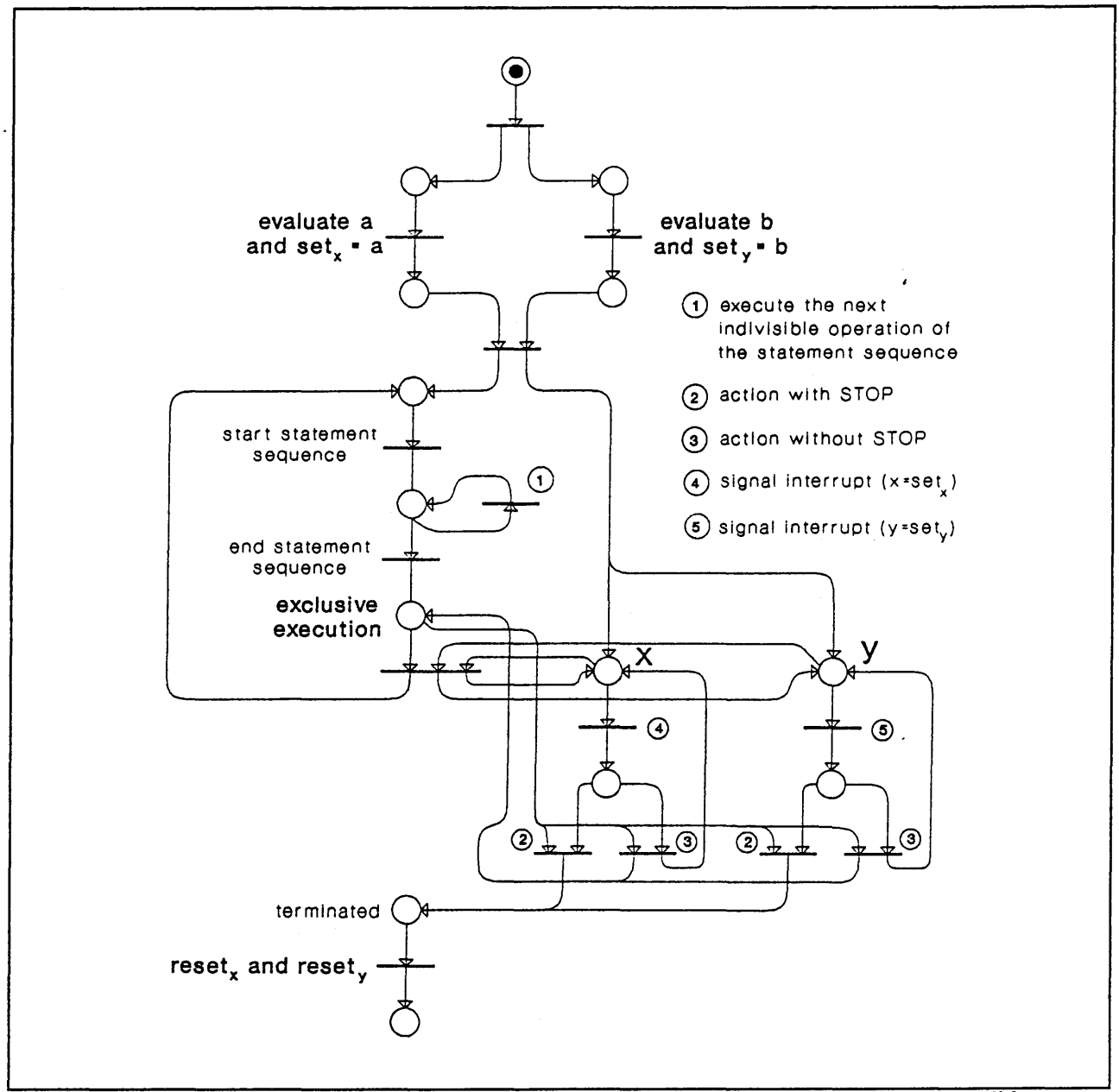

Figure 3. The semantics of an infinite sequential loop with two deferred interrupt conditions.

The semantics of a concurrent loop with one or more normal interrupt conditions allow for the execution of each of the component sub-processes to be held up while the appropriate action is taken on an interrupt.

5.2 Interrupt-generating variables

HUL adopts the model of specification S-2 for its interrupt-generating variables, and the *do* control structure has been designed to exploit this model. Interrupt-generating variables may be defined for all data types supported by HUL.

In designing the syntax of HUL, an attempt has been made to make the low level operation of interrupt-generating variables as transparent as possible, so that the programmer does not need to conceptualize an interrupt-generating variable as being different from an ordinary variable. This is emphasized by the omission from the syntax of HUL of any reserved word which designates a variable as being interrupt-generating. To identify interrupt-generating variables for code generation purposes the HUL compiler makes use of a first pass which locates instances of identifiers used in interrupt-generating contexts, and distinguishes them from normal variables in the compiler's symbol table, even though they were all declared in exactly the same

way. Thus there should be no conceptual difference between the use of interrupt-generating variables and ordinary variables. The difference is in the way the two classes of variables are implemented. The programmer simply has to be aware that only those variables which are used to form interrupt conditions may be declared globally and referenced as shared data by concurrent processes, and that two constraints apply to these variables: they may appear in only one interrupt condition at a time[vii], and they may not be referenced unless they are part of an active interrupt condition. The former constraint avoids situations which could cause deadlock, and the latter constraint ensures that shared variables are implemented as interrupt-generating processes. In programs for which efficient object code is required, the programmer may have to take cognisance of the fact that references to interrupt-generating variables incur a higher run-time cost than normal memory references (because they are implemented as communicating processes).

To prevent concurrent processes from demanding simultaneous interrupt signals from the same interrupt-generating variable, HUL enforces the following strict ownership rule[viii]:

> ***The first process to set the interrupt value of an interrupt-generating variable is given exclusive ownership of the variable until it explicitly relinquishes ownership. Only the owner process will receive interrupt signals from an interrupt-generating variable, although the variable may be referenced for fetching and storing by other processes executing concurrently with the owner process.***

Since interrupt-generating variables are implemented as processes, ownership is claimed by sending the interrupt-generating process a *set* message, which specifies the interrupt value to be stored in the *test* register. A *reset* message relinquishes ownership. HUL's *do* control structure automatically generates these signals to claim ownership (for the duration of the control structure) of any interrupt-generating variables which are named in *when* statements within its scope. The following example illustrates this mechanism.

```
Do
  {process body}
  when X = 3
    {action}
```

The interrupt value is evaluated (the simple constant 3 in this example) and a *set* message is produced to send this value to the interrupt-generating variable X before the control structure begins executing. This sets up an interrupt condition sustained by X, thereby claiming exclusive ownership of X. The process body of the control structure is executed (repeatedly in this example). During the execution of the process body, an interrupt signal is generated each time X is set to the value 3, which suspends the process body and executes the action sequence of the *when* statement. A *reset* message relinquishes ownership of X when the control structure terminates.

6. Conclusions

A distributed parallel processing environment for HUL has been implemented on an IBM PC microcomputer network and on a network of transputers. In these test environments, the interrupt-generating model is implemented as an active process in software, which executes in parallel with the processes of the application program with which it communicates. References to an interrupt-generating variable, and interrupt signals produced by it, are handled using the same form of synchronous message passing that is used to effect communication between processes of the application program[ix]. In the object form of a program, processes generated by the compiler to emulate variables have the same form as the processes of the source program. No special communication facilities are needed to support the interrupt-generating variable concept; the model is intended to form a natural extension of the general communication layer.

The *do* control structure of HUL provides a versatile programming construct which improves on existing concurrent control structures in accommodating the expression of occasional events. This interrupt driven construct also avoids the polling bias evident in many applications which require a process to choose from a number of possible rendezvous.

The test implementations of the support environment for HUL have shown the proposal for the interrupt-generating variable to have a number of desirable qualities.

- The interrupt-generating variable provides an easily used facility for shared memory concurrency in multiprocessor systems in which processors have no memory in common. This facility relieves programmers of having to map concurrent solutions in which shared variables are a fundamental aspect of the problem area onto sets of message exchanges. However, point to point communication and pipelined applications are still more appropriately handled by channels.

- The interrupt-generating variable provides a structured way to control access to a shared variable. The model proposed in this paper

includes built-in exclusive access of the shared item, with an interrupt generation mechanism to support condition synchronization. The proposal does not support imperative synchronization, and the established message passing notation of CSP was incorporated into HUL for this purpose.

- The interrupt-generating model exhibits reasonable safety properties. Built-in mutual exclusion avoids interference in references to a shared variable controlled by this structure, and the ownership rule facilitates contention free condition synchronization.

- The proposal is sufficiently simple to be implemented in the form of an active process with reasonable efficiency. The test implementations of HUL have shown that the combined run-time overhead of interrupt-generating processes and associated message routing processes is not prohibitively large, provided these processes are placed wisely in relation to the application processes they serve. The measured overhead fell into the range 8 to 40% of the total execution time, measured by comparing the execution times of test programs which used interrupt-generating variables with equivalent message passing programs which were specially tailored to run on the system without system processes.

On the negative side, HUL has not succeeded in making the differences between normal variables and interrupt-generating variables fully transparent to the programmer. Although the physical appearance of program listings does not distinguish between the two types of variable, the programmer is made aware of the existence of the ownership rule if an attempt is made to use the same variable to support two interrupt conditions simultaneously, or to reference an interrupt-generating variable which is not *owned*; and although the model of interrupt-generating variables is based on the established theoretical foundation of CSP, a programmer wishing to perform formal proofs on a program which makes use of interrupt-generating variables will have to be aware of their distinctive behaviour.

Notes

i. Notable commercial examples are Inmos's transputer networks [17], NCUBE's NCUBE/10, and Intel's iPSC [10].
ii. *MOD is pronounced STAR-MOD.
iii. Ada [1], also influenced by CSP and DP, does not completely reflect a distributed architecture and contains certain features (identified by Stammers [16]) which rely on a single shared memory.
iv. Some multitasking operating systems (for example UNIX [3]) do allow system calls from high level languages which effect run-time interrupts in application programs.
v. Note that output guards have been allowed in the CSP notation to simplify the specifications.
vi. HUL is an acronym for HUMAN-LIKE.
vii. This is ensured by the exclusive ownership rule.
viii. Because the difference between normal and interrupt-generating variables is apparent only from their appearance in particular programming constructs, the compiler needs to issue a warning when an interrupt-generating variable is used to provide interrupt conditions for two or more processes which may be executed in parallel. Although some elaborate algorithms might make deliberate use of the ownership rule, this is not normally the case.
ix. A blocking message passing model is used [7].

References

[1] Ada Reference Manual, [1983], J Ichbiah, *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815A.

[2] G R Andrews and F B Schneider, [1983], Concepts and Notations for Concurrent Programming, *Computing Surveys*, **15**(1), 3-43.

[3] M J Bach, [1986], *The Design of the UNIX Operating System*, Prentice-Hall, Englewood Cliffs, N.J.

[4] P Brinch Hansen, [1978], Distributed Processes: A Concurrent Programming Concept, *Comm. ACM*, **21**(11), 934-941.

[5] P G Clayton, [1978], *HUL - Language Definition*, Tech. Doc. 87/21, Department of Computer Science, Rhodes University.

[6] R P Cook, [1988], *MOD - A Language for Distributed Programming, *IEEE Trans. Software Eng.*, **SE-6**(6), 563-571.

[7] W M Gentleman, [1981], Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept, *Software - Practice and Experience*, **11**(5), 435-466.

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

[9] C A R Hoare, [1985], *Communicating Sequential Processes*, Prentice-Hall, Englewood Cliffs, N.J.

[10] Intel, [1988], *A Technical Summary of the iPSC/2 Concurrent Computer*, Intel Scientific Computers, Beaverton.

[11] D G Kafura and K H Lee, [1989], Inheritance in Actor Based Concurrent Object-Oriented Languages, *The Computer Journal*, **32**(4), 297-304.

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

[13] D May, [1987], *Occam 2 Language Definition*, Inmos Ltd., Bristol.
[14] J L Peterson, [1981], *Petri Net Theory and the Modelling of Systems*, Prentice-Hall, Englewood Cliffs, N.J.
[15] F B Schneider and G R Andrews, [1985], Concepts for Concurrent Programming, in J W de Bakker, W P de Roever, and G Rozenberg (*Eds.*), *Current Trends in Concurrency*, Lecture Notes in Computer Science, **224**, Springer-Verlag, 669-716.
[16] R A Stammers, [1985], Ada on Distributed Hardware, in G L Reijns and E L Dagless (*Eds.*), *Concurrent Languages in Distributed Systems*, Elsevier, North Holland.
[17] P Walker, [1985], The Transputer: A Building Block for Parallel Processing, *Byte*, **10**(5), 219-235.

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 4, March 1991
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 4, Maart 1991
ISSN 1015-7999

Contents