INTERFERENCE CONTROL IN SUPERPASCAL

Process interference due to shared variables is a serious problem in parallel programs written in insecure programming languages. The only effective remedy has been known since the 1970s: a parallel programming language must be designed to permit detection of process interference during compilation. This sound principle has seldom been adopted and enforced consistently for block-structured parallel languages. This paper discusses syntactic control of interference in SuperPascal, a block-structured programming language for parallel scientific computing. SuperPascal omits the insecure concepts of Pascal, and adds parallel statements and synchronous communication channels. Restrictions on the use of variables permit a single-pass compiler to check that parallel processes are disjoint, even if they use procedures with global variables.

1 INTRODUCTION

Scientific computer users are now facing the same problem that operating system designers encountered in the 1960s: How do we prevent *time-dependent errors* that make parallel computations behave in a completely erratic manner? This serious problem is also known as *process interference*.

If parallel processes update the same variable at unpredictable times, the combined effect is time-dependent. Similarly, if two parallel processes both attempt to send (or receive) messages through the same channel at unpredictable times, the net effect is time-dependent. This paper discusses process interference caused by *variable conflicts* but ignores interference due to *channel conflicts*.

The effects of time-dependent errors are both unpredictable and irreproducible. Consequently, it is rarely possible to locate interference by program testing. In the 1970s, a radical solution to this problem was proposed: *a parallel programming language must be designed to permit detection of process interference during compilation* (Hoare 1972, Brinch Hansen 1973).

P. Brinch Hansen, Interference control in SuperPascal—A block-structured language parallel language. *The Computer Journal 37*, 5 (1994), 399–406.

Concurrent Pascal was the first programming language designed to permit syntactic detection of process interference (Brinch Hansen 1975). The key idea was *modular parallelism*. The variables and algorithm of a process were combined into a syntactic unit. The scope rules made the local variables of each module inaccessible to other modules.

Unfortunately, the simple modular approach to interference control has not been widely adopted by a new generation of programmers who are exploiting parallelism in insecure extensions of Fortran, C and Pascal.

Although I regard the trend towards non-modular parallelism as a step backwards, I also recognize it as another mountain to climb. So I will approach the interference problem from a different angle by asking the question: Is it feasible to detect variable conflicts in a block-structured language with *non-modular parallelism*? The answer is yes, provided the language has certain pleasant properties:

1. *Structured parallelism.* It must be possible by scanning a program text to identify all statements that may be executed in parallel.

2. *Transparent contexts.* It must be possible by scanning a program text to identify all variables that may be updated or used as expression operands during the execution of any statement.

3. *Disjoint variables.* Distinct variable identifiers occurring in the same statement must denote distinct variables.

These properties are also helpful to a person who reads a program text.

I will discuss syntactic control of interference in *SuperPascal*, a block-structured programming language for parallel scientific computing. SuperPascal omits the insecure concepts of Pascal and adds parallel statements and synchronous communication channels (Welsh 1977, IEEE 1983, Brinch Hansen 1994a, b). I will ignore communication and concentrate on making parallel variable access secure. In SuperPascal, restrictions on the use of variables permit a single-pass compiler to detect variable conflicts, even if parallel processes use procedures with global variables.

My emphasis on single-pass compilation is a personal preference. A language designed for single-pass compilation enables both compilers and programmers to read program texts from the beginning without forward references. The challenge of language design is to select a notation that offers an abstract model of the essential properties of a computer. The right compromise will simplify both programming and compilation. A language

that requires multi-pass compilation with extensive optimization is, in my opinion, a poor choice.

The general guidelines for extending a block structured language with secure parallelism have been known for two decades but have seldom been adopted and enforced consistently (Hoare 1971, 1972, Brinch Hansen 1973, Inmos 1988. I will recapitulate these ideas and reach a surprising conclusion: The selection of secure programming concepts for a block-structured parallel language is dominated by the issue of interference control.

This is one of several publications that illustrate the merits of using SuperPascal to publish parallel scientific algorithms in their entirety as executable programs expressed in a clear, secure notation.

In the *language tutorial* (Brinch Hansen 1994a), I argue that universities should adopt a common programming language for publication of papers and textbooks on parallel scientific computing. The programming language Pascal has played a major role as a publication language for sequential computing. Building on that tradition, I have developed SuperPascal as a publication language for computational science. The tutorial also discusses detection of channel conflicts and compares SuperPascal to occam (Inmos 1988).

The *language report* (Brinch Hansen 1994b) defines the parallel features of SuperPascal concisely using the terminology and syntax notation of the Standard Pascal report (IEEE 1983). The report also summarizes the differences between SuperPascal and Pascal.

A collection of SuperPascal programs for computational science, summarized in Brinch Hansen (1993), is described in detail in a forthcoming book (Brinch Hansen 1994c). These programs have also been rewritten in the implementation language occam and tested on a Computing Surface with 40 T800 transputers.

I have developed a *portable implementation* of SuperPascal on a Sun workstation under Unix. It consists of a compiler and an interpreter written in Pascal. You can obtain the SuperPascal software by using anonymous FTP from the directory *pbh* at *top.cis.syr.edu*.

At Syracuse University, undergraduate and graduate students have used the SuperPascal software in computational science courses.

2 PROCESSES AND VARIABLES

I begin by introducing some terminology. Every occurrence of a variable access, an expression, a statement or a statement sequence is called a *com-*

mand. The evaluation or execution of a command is called a *process*. A structured process is a sequential or parallel composition of processes. The components of a parallel composition are called *parallel processes*. As long as they do not communicate, parallel processes proceed independently at unpredictable speeds until all of them have terminated.

In a program text, an entire variable is a syntactic entity that has an identifier, a type and a textual scope.

During recursive and parallel activations of a block, multiple instances of the local variables exist. Each variable instance is a dynamic entity that has a location, a current value, and a finite lifetime in memory.

The distinction between a *variable* as a syntactic entity in the program text and a class of dynamic entities in memory is usually clear from the context. Where it is : necessary, I will distinguish between *syntactic variables* and *variable instances*.

I will impose the following requirement on a set of parallel processes P_1, P_2, ..., P_n:

The variable instances used by the n processes must be divided into $n + 1$ disjoint sets. Each process P_i has access to a set of variable instances that may be updated and accessed by P_i only. The processes share a set of variable instances that may be accessed, but not updated, by any of them. Parallel processes that satisfy this requirement are called *disjoint processes*.

3 SYNTACTIC VARIABLES

3.1 Variable contexts

To facilitate the detection of variable conflicts, I associate a context of entire variables with every command. An *entire variable* is a variable denoted by an identifier only. The *variable context* of a command C consists of two sets of entire variables, called the *target* and *expression variables* of C. If the process denoted by C may assign a value to an entire variable v (or one of its components), then v is a target variable of C. If the process may use the value of v (or one of its components) as an operand, then v is an expression variable of C. Since variable conflicts are detected during compilation, target and expression variables are syntactic variables.

An entire variable may be both a target and an expression variable of the same command.

Example: The target and expression variables of the assignment statement

$$x := x + y$$

are the sets $\{x\}$ and $\{x, y\}$.

3.2 Structured variables

During compilation, it is generally impossible to distinguish between different components of a structured variable. So I regard an operation on the component of a variable as an operation on the entire variable. That is why the concept of variable context is defined in terms of sets of entire variables.

Example: The target and expression variables of the assignment statement

$$a[i] := a[j] - b[j]$$

are the sets $\{a\}$ and $\{a, b, i, j\}$.

3.3 Structured commands

A compiler cannot predict whether or not a particular component of a conditional statement will be executed, so I adopt the conservative rule that the variable context of a structured command is the union of the contexts of its components.

Example: The target and expression variables of the *if* statement

$$\textbf{if } x > y \textbf{ then } x := x - y \textbf{ else } y := y - x$$

are the sets $\{x, y\}$ and $\{x, y\}$.

3.4 Pointer types

A compiler can only determine the variable context of a command if every entire variable is *declared* in the program text. Since a pointer variable x may refer to an undeclared variable $x^\wedge$ pointer types are omitted from SuperPascal.

4 STRUCTURED PARALLELISM

4.1 Algorithmic parallelism

Algorithmic parallelism is parallel execution of different algorithms using different data structures. It is expressed by parallel statements.

The effect of a *parallel statement*

$$\textbf{parallel } S_1|S_2|\ldots|S_n \textbf{ end}$$

is to execute the process statements S_1, S_2, ..., S_n as parallel processes until all of them have terminated.

Example: The parallel statement

$$\textbf{parallel } \text{master(b,a)}|\text{pipeline(a,b)} \textbf{ end}$$

denotes parallel execution of a master process and a pipeline process. Each process is defined by a procedure statement with two value parameters.

The following restriction ensures that a parallel statement defines disjoint processes: *in a parallel statement, a target variable of one process statement cannot be a target or an expression variable of another process statement.*

Example: The parallel statement

$$\textbf{parallel } x := 1|y := 2 \textbf{ end}$$

denotes disjoint processes (provided x and y are not aliases of the same syntactic variable).

Example: The parallel statement

$$\textbf{parallel } a[i] := 1|a[j] := 2 \textbf{ end}$$

is incorrect because the array variable a is a target variable of both process statements.

Example: The parallel statement

$$\textbf{parallel } i := j + 1|a[i] := a[j] \textbf{ end}$$

is incorrect because i is a target variable of the first process statement and an expression variable of the second process statement.

4.2 Data parallelism

Data parallelism is parallel execution of the same algorithm applied to different parts of the same data structure. It is expressed by *forall* statements.

A *forall* statement

$$\textbf{forall } i := e_1 \textbf{ to } e_2 \textbf{ do } S$$

denotes a (possibly empty) array of parallel processes, called *element processes*, and a corresponding range of values, called *process indices*. The lower and upper bounds of the index range are denoted by two expressions, e_1 and e_2, of the same simple type. Every index value corresponds to a separate element process defined by an *index variable* i and an *element statement* S.

The *index variable declaration*

$$i := e_1 \textbf{ to } e_2$$

introduces the variable i that is local to S.

A *forall* statement is executed as follows:

1. The expressions e_1 and e_2 are evaluated. If $e_1 > e_2$, the execution of the *forall* statement terminates; otherwise, step 2 takes place.

2. $e_2 - e_1 + 1$ element processes run in parallel until all of them have terminated. Every element process creates a local instance of the index variable i, assigns the corresponding process index to the variable, and executes the element statement S. When an element process terminates, its local instance of the index variable ceases to exist.

Example: The *forall* statement

$$\begin{array}{l}\textbf{forall } i := 1 \textbf{ to } p \textbf{ do} \\ \quad \text{node}(i,\, c[i-1],\, c[i])\end{array}$$

denotes parallel execution of identical processes with indices from 1 to p. Each process is defined by a procedure statement with three value parameters.

If the element statement has a target variable, the parallel processes may share a single instance of this variable in a time-dependent manner. To prevent this kind of interference, I impose a severe restriction: *in a forall statement, the element statement cannot use target variables.*

This restriction implies that a process array must output its results through a communication channel or a file. Otherwise, the results will be lost when the element processes terminate and their local variables cease to exist.

Example: The *forall* statement

$$\textbf{forall } i := 1 \textbf{ to } n \textbf{ do } a[i] := 0$$

is incorrect because the array variable a is a target variable of the element statement.

Example: The *forall* statement

$$\textbf{forall } i := 1 \textbf{ to } n \textbf{ do } i := i + 1$$

is incorrect because the index variable i is a target variable of the element statement.

Example: The *forall* statement

$$\textbf{forall } i := 1 \textbf{ to } i + 1 \textbf{ do } S$$

is incorrect because the index variable i is unknown outside the element statement S.

4.3 Goto statements

The syntax of the parallel and *forall* statements makes it easy to scan a program text and recognize statements that may be executed in parallel. However, if you permit arbitrary jumps in statements, a compiler cannot always determine whether certain statements may be executed in parallel. Consequently, *goto statements and labels are omitted* from SuperPascal.

5 PROCEDURES

5.1 Global variables

The use of global variables in procedures complicates syntactic detection of process interference considerably.

Example: In procedure P, the global variable x and the variable parameter y are target variables of different process statements:

```
var x: integer;

procedure P(var y: integer);
begin
  parallel x := 1|y := 2 end
end;
```

If the identifiers x and y denote distinct variables, the parallel statement defines disjoint processes. However, if the global variable x occurs as an actual parameter in a procedure statement $P(x)$, the parallel statement is incorrect because the identifiers x and y now refer to the same variable.

Example: In procedure Q, the global variable x is an expression variable of one process statement, and the variable parameter y is a target variable of another process statement:

```
var x: integer;

procedure Q(var y: integer);
begin
  parallel write(x)|y := 1 end
end
```

If the global variable x occurs as an actual parameter in a procedure statement $Q(x)$, the parallel statement is incorrect because the identifiers x and y are aliases of the same variable.

Variable aliasing violates the sound principle that distinct variable identifiers used in the same statement should denote distinct syntactic variables. To discuss the aliasing problem concisely, I need more terminology.

Consider a procedure P with the statement part S. An *implicit parameter* of P is an entire variable v that is global to P and is part of the variable context of S.If v is a target variable of S, then v is an *implicit variable parameter* of P.If v is an expression variable of S, then v is an implicit value parameter of P. The implicit parameters of a procedure P are also regarded as implicit parameters of every procedure statement that denotes activation of P. So, the *actual parameters* of a procedure statement are the *explicit parameters* that occur in the actual parameter list and the *implicit parameters* of the corresponding procedure.

Using this terminology, I define a restriction that prevents the aliasing problems illustrated by the two previous examples: an *implicit parameter of a procedure statement S cannot occur as an explicit variable parameter in S.*

This rule is a special case of a more general restriction that will be imposed later.

An entire variable v is a target variable of a procedure statement S if v (or one of its components) is an explicit variable parameter of S, or if v is an implicit variable parameter of S.

The rules of implicit parameters imply that one procedure can inherit the implicit parameters of another procedure.

Example: Procedure Q has an implicit variable parameter x. Since procedure P contains the procedure statement $Q(a + 1)$, x is a target variable of the statement part of P. And, since x is global to P, x is also an implicit variable parameter of P.

```
var x: integer;

procedure Q(b: integer);
begin x := b end;

procedure P(a: integer);
begin Q(a + 1) end;
```

This insight reveals that the parallel statement

```
parallel x := 1 |P(1) end
```

is incorrect because x is a target variable of both process statements. Notice that x is a target variable of the procedure statement $P(1)$, although x does not occur in the statement.

5.2 Procedural parameters

I assume that a compiler has determined the variable context of any command C when it reaches the end of C. However, in general, a single-pass compiler cannot detect the implicit parameters of procedural parameters.

Example: When a single-pass compiler reaches the procedure statement Q in procedure P, the procedural parameter Q is unknown:

```
var x: 1nteger;

procedure P(procedure Q);
begin Q end;

procedure R;
begin x := 1 end;

P(R)
```

In general, Q varies from one activation of P to another. For the activation $P(R)$, the procedural parameter Q denotes procedure R, which uses x as an implicit variable parameter.

The simplest solution to this problem is to *omit procedural parameters* from SuperPascal.

5.3 Recursive procedures

A single-pass compiler cannot always detect the implicit parameters of recursive procedures.

Example: When a single-pass compiler reaches the parallel statement in procedure P, it has not yet scanned the rest of P:

```
var x: integer;

procedure P(y: integer);
begin
  if y > 2 then
    parallel P(y − 1)|P(y − 2) end
  else x := y
end;
```

At this point the compiler is unable to detect that x is an implicit variable parameter of P, which means that x is a shared target variable of the parallel recursive activations of P.

I solve this problem, too, by outlawing it: a *recursive procedure cannot use implicit parameters*. To enforce this restriction, the compiler must be able to detect recursion.

If the heading of a procedure precedes every use of the procedure identifier in the program text, there are only two possible forms of recursion, both of which are easy to detect. Either a procedure P activates itself

```
procedure P;
begin P end;
```

or a procedure Q activates a surrounding procedure P which, in turn, activates Q recursively

```
procedure P;
  procedure Q;
  begin P end;
begin Q end;
```

In either case, an activation of a procedure P is recursive if the corresponding procedure statement occurs within the declaration of P.

5.4 Forward declarations

Forward declarations of procedures make syntactic detection of recursion impossible during single-pass compilation.

Example: During compilation of procedure P it cannot be determined that procedures P and Q are mutually recursive:

```
procedure Q; forward;
procedure P; begin Q end;
procedure Q; begin P end;
```

Again, I eliminate the problem by *omitting forward declarations* from SuperPascal.

5.5 File procedures

When you consider the complications introduced by implicit parameters, it is tempting to exclude them altogether. I have not done that. Part of the challenge of extending a block-structured language with secure parallelism is to face the problems of global variables head-on.

The issue can hardly be ignored, since even the required procedures of Pascal use implicit parameters. In Pascal, the required textfile *input* is an implicit variable parameter of any *read* statement that omits the file identifier. The required textfile *output* is an implicit variable parameter of any *write* statement that omits the file identifier. This convention enables a compiler to prevent simultaneous access to the same file.

Example: The parallel statement

parallel write(x)|writeln **end**

is incorrect because the file *output* is a target variable of both process statements.

5.6 Variable parameters

Formal variable parameters are another potential source of variable aliasing.

Example: The parallel statement in procedure P is incorrect if the variable parameters x and y denote the same actual parameter:

```
procedure P(var x, y: integer);
begin
  parallel x := 1|y := 2 end
end
```

This happens if the same variable z occurs twice in a procedure statement $P(z, z)$

This loophole is plugged by introducing a new concept and a stronger restriction for procedure statements: the *restricted actual parameters* of a procedure statement are the explicit variable parameters that occur in the statement and the implicit parameters of the corresponding procedure block. *The restricted actual parameters of a procedure statement must be distinct entire variables (or components of such variables).*

Example: The exchange of the values of two array elements $a[i]$ and $a[j]$ cannot be denoted by a procedure statement

swap($a[i]$, $a[j]$)

in which the array variable a occurs twice as an actual variable parameter. It can, however, be expressed by a procedure statement with the actual variable parameter a and the value parameters i and j:

swap(a, i, j)

6 FUNCTIONS

6.1. Variable parameters

Functions with side-effects complicate the meaning of even the most elementary concepts in a programming language. Such a language does not even have the algebraic property that $e = e$ for any expression e.

Example: The function f has an implicit variable parameter x:

```
var x: integer;

function f(y: integer): integer;
begin x := x + y; f := x end;
```

The function designator $f(1)$ is an expression with the sad property that $f(1) <> f(1)$.

Example: The function f has an explicit variable parameter x:

```
function f(var x: integer): integer;
begin x := x + 1; f :- x end;
```

The function designator $f(y)$ uses an actual variable parameter y. For this expression, you also have $f(y) <> f(y)$.

A function concept that does not satisfy the fundamental rules of arithmetic expressions is an absurd choice for a language designer who is striving for simplicity. The choice is clear. I need simple rules to prevent side-effects of functions:

1. *A function cannot have implicit or explicit variable parameters.*

2. *A function cannot use required or declared procedures.*

The second rule prevents indirect side-effects due to message communication and file input/output.

6.2 File functions

I am still left with one question: Should I permit functions to use global variables as *implicit value parameters*?

The answer is yes, for the simple reason that some required functions of Pascal already use such parameters. The required textfile *input* is an implicit

value parameter of any *eof* (or *eoln*) function designator that omits the file identifier.

Example: The parallel statement

```
parallel
  read(x)|if eof then writeln
end
```

is incorrect, because the file *input* is a target variable of the first process statement and an expression variable of the second process statement.

6.3 Function results

Finally, when the identifier of a function occurs on the left side of an assignment operator in the statement part of the function block, I regard it as the identifier of a local target variable that holds the function result. This convention facilitates the detection of a meaningless attempt to assign multiple values to the same function designator.

Example:

```
function f(x: integer): integer;
begin
  parallel f:= x|f := 2 * x end
end
```

7 TARGET AND EXPRESSION VARIABLES

A command denotes a process that uses a set of target variables and a set of expression variables. I am now ready to give concise recursive definitions of these two sets of entire variables.

An *entire variable* is a variable denoted by one of the following kinds of identifiers:

1. A variable identifier introduced by a variable declaration or a *forall* statement.

2. A function identifier that occurs as the left part of an assignment statement in the statement part of the corresponding function block.

An entire variable v is a *target variable* of a command C in the following cases:

1. The variable identifier v occurs in an assignment statement C that denotes assignment to v (or one of its components).

2. The variable identifier v occurs in a *for* statement C that uses v as the control variable.

3. The variable identifier v occurs in a procedure statement C that uses v (or one of its components) as an actual variable parameter.

4. The variable v is an implicit variable parameter of a procedure block B and C is a procedure statement that denotes activation of B.

5. The variable v is a target variable of a command D and C is a structured command that contains D.

An entire variable v is an expression variable of a command C in the following cases:

1. The variable identifier v occurs in an expression C that uses v (or one of its components) as an operand.

2. The variable identifier v occurs in the element statement C of a *forall* statement that introduces v as the index variable.

3. The variable v is an implicit value parameter of a function block B and C is a function designator that denotes activation of B.

4. The variable v is an implicit value parameter of a procedure block B and C is a procedure statement that denotes activation of B.

5. The variable v is an expression variable of a command D and C is a structured command that contains D.

8 UNRESTRICTED STATEMENTS

I have imposed the following *restrictions* on the use of variables and procedure statements, parallel statements and *forall* statements:

1. In a *procedure statement*, the restricted actual parameters must be distinct entire variables (or components of such variables).

2. In a *parallel statement*, a target variable of one process statement cannot be a target or an expression variable of another process statement.

3. In a *forall statement*, the element statement cannot use target variables.

Occasionally, it is necessary to override these restrictions on variable access to define operations on distinct elements of the same array.

An *unrestricted statement*

$$[\textbf{sic}]\ S$$

is a statement S prefixed by a *sic* clause. The above restrictions do not apply to unrestricted statements. Consequently, the programmer must prove that every unrestricted statement preserves the disjointness of parallel processes; otherwise the semantics of unrestricted statements are beyond the scope of the language definition.

Examples:

[**sic**] $\{i <> j\}$
  swap$(a[i], a[j])$

[**sic**] $\{i <> j\}$
  parallel $a[i] := 1 | a[j] := 2$

[**sic**] {disjoint elements $a[i]$}
  forall $i := 1$ **to** n **do** $a[i] := 0$

If a parallel program is written in a language without interference control, the programmer is completely responsible for avoiding variable conflicts. In SuperPascal, disjoint parallelism is the default and insecure parallelism is the exception. The syntax of unrestricted statements clearly marks the few places where the programmer has the obligation to prove disjointness.

9 CONCLUSION

The design of *SuperPascal* was guided by two objectives:

1. *Simplicity.* Derive an elegant programming language for parallel scientific computing by adding a minimal number of concepts to the block-structured language Pascal.

2. *Security.* Impose additional restrictions on the programming concepts of Pascal to permit a single-pass compiler to check that parallel processes are disjoint.

Simplicity was achieved by extending Pascal with three concepts only: parallel statements, *forall* statements and synchronous communication channels.

Security was ensured by imposing additional restrictions on procedures and functions and omitting some features of Pascal:

1. The explicit variable parameters and the implicit parameters of a procedure statement cannot be aliases.
2. Recursive functions and procedures cannot use global variables.
3. Functions cannot update global variables and cannot use variable parameters or procedure statements.
4. Procedures and functions cannot use procedures and functions as parameters.
5. Forward declarations of procedures and functions cannot be used.
6. Pointer types cannot be used.
7. Goto statements and labels cannot be used.

These were the most difficult choices made in the design of SuperPascal. The only remaining task was to eliminate well-known ambiguities and insecurities of Pascal.

So, I draw the following conclusion: *the selection of secure programming concepts for a block-structured parallel language is dominated by the issue of interference control.*

Acknowledgements

I thank Steve Cooper, Jonathan Greenfield, Peter O'Hearn, Dave Parnas and the anonymous referees who helped me improve this paper.

References

Brinch Hansen, P. (1973) *Operating System Principles.* Prentice-Hall, Englewood Cliffs, NJ.

Brinch Hansen, P. (1975) The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering 1*, 199-207.

Brinch Hansen, P. (1993) Model programs for computational science: a programming methodology for multicomputers. *Concurrency—Practice and Experience 5*, 407-423.

Brinch Hansen, P. (1994a) SuperPascal—a publication language for parallel scientific computing. *Concurrency—Practice and Experience 6*, to appear.

Brinch Hansen, P. (1994b) The programming language SuperPascal. *Software—Practice and Experience 24*, 467–483.

Brinch Hansen, P. (1994c) *Studies in Computational Science*, to appear.

Hoare, C. A. R. (1971) Procedures and parameters: an axiomatic approach. *Lecture Notes in Mathematics 188*, 102–116.

Hoare, C. A. R. (1972) Towards a theory of parallel programming. In Hoare, C. A. R. and Perrott, R. H. (eds.) *Operating Systems Techniques*, Academic Press, New York.

IEEE (1983) *IEEE Standard Pascal Computer Programming Language*. IEEE, New York.

Inmos (1988) *occam2 Reference Manual*. Prentice-Hall, Englewood Cliffs, NJ.

Welsh, J., Sneeringer, W. J. and Hoare, C. A. R. (1977) Ambiguities and insecurities in Pascal. *Software—Practice and Experience 7*, 685–696.