Security aspects of occam 2

INMOS Technical Note 32

Roger Shepherd

October 1987
72-TCH-032-00

Contents

1 Introduction **4**

2 The data types of occam 2 **4**

3 Channel protocols **5**

4 Numerical behaviour **9**

5 Abbreviations **10**

6 Alias checking **11**

7 Checking the validity of parallel constructs **15**

8 Run-time error handling in occam 2 **16**

9 Conclusion **17**

References **17**

1 Introduction

The major reason for the design of the occam 2 [1] programming language was a desire to incorporate floating point arithmetic into occam. This had to be done without breaching the security of the language. As a result occam 2 is well defined [2], many programming errors are detectable at compile time, and run-time errors are reliably and cheaply detectable. This paper describes various aspects of the language design which relate to security; some of these, such as 'channel protocols' overcome problems caused by the introduction of new data-types to the language, others, such as alias checking, tackle security problems which are present in most other programming languages.

2 The data types of occam 2

The occam 1 programming language provided concurrency, message passing and a limited set of data types; the word, the channel and vectors of words or of channels. Although this was sufficient for many purposes, there were instances where a language which had a richer set of data types would offer significant advantages. In particular:

1 We wanted to support numerical programming; for occam to become the FORTRAN of parallel processing we would have to support floating-point arithmetic and multiple dimensional arrays.

2 We wanted to be able to pass messages of length greater than a single word. This is because much of the cost of passing a message is due to process synchronisation rather than data transfer. A language which would permit several words to be communicated in a single transfer would be more efficient than one which could transfer only single words.

3 We wanted to program systems of processors which did not share a common wordlength. In such systems communication would have to be in terms of some unit other than the word.

As a result occam 2 supports several primitive data types. There is a machine dependent data-type, INT, which loosely corresponds to the VAR of occam 1. INT is the type of signed integer values most efficiently provided by the machine. As this normally corresponds with the size of an address in the a machine values of type INT are used for such purposes as replicator indices and array subscripts. Unlike occam 1, there is a separate type BOOL, which represents boolean values, and a BYTE type, which represents unsigned integers in the range 0 to 255. (Note that although occam 1 could pack and unpack values into bytes, the byte was not a proper data-type; all arithmetic and message passing was done in terms of words).

There are occasions where the use of a machine dependent type such as INT is not satisfactory; for example, where a message is to be passed between two machines of differing word length, or where a calculation has to be performed to a particular precision, regardless of the machine on which it is to be performed. To cope with these situations, occam 2 has a further three integer types, INT16, INT32 and INT64, which represent signed integers with a length of 16, 32 and 64 bits respectively.

There are two floating-point types called REAL32 and REAL64. These correspond to the single and double length floating point numbers of the IEEE Standard for Binary Floating Point. In fact occam does not support the multiple error symbols of the standard as this would undermine the substitution semantics of the language; for example, in full IEEE arithmetic, x = y does not imply that x op z = y op z.

In addition to the primitive types, occam has array types. The components of an array may be of any single type. As an array may have components of an array type, occam 2 does provide multi-dimensional arrays. An array may be subscripted (giving a component of the array), assigned, passed as a message and used as a parameter to a procedure or a function.

3 Channel protocols

The presence of several different data types in occam 2 introduces the problem of how to extend the communication model to handle them. The problem arises from the need to ensure that when a message is passed, the type of data sent by the transmitter matches the type of data expected by the receiver. It is desirable to provide some sort of checking of channel communication for two reasons. Firstly, it is very easy to make mistakes in communication and anything which enables these mistakes to be detected at compile time is helpful. Secondly, the effect of run-time errors in communication can be at least as devastating as subscript errors; it can cause store to be overwritten arbitrarily, or can cause the breakdown of process synchonisation.

A number of different proposals were considered during the design of occam 2. Some, such as restricting a channel to communicating a single type, were rejected due to lack of flexibility. Others, such as type-checking all communication at run-time, were rejected as they carried too much run-time overhead.

The solution adopted in occam 2, the channel protocol, allows great freedom over what is communicated on a channel, but ensures security. Whenever a channel is declared the structure of all communication occuring on that

channel must also be declared as a channel protocol. This enables most communication to be checked at compile-time, and simplifies any remaining run-time checks.

The simplest protocol permits communication of a single type which may, of course, be an array type. For example, if the channel greeting is being used to communicate strings which are to be displayed on a 12-character LCD display, it might be declared

```
CHAN OF [12]BYTE greeting :
```

The compiler can subsequently check that all inputs and outputs correspond to this protocol. Thus the compiler would accept

```
greeting ! "Hello world!"
```

but would reject

```
greeting ! "Goodbye world!"
```

as the string has too many characters.

Whilst this example is perfect for the application described it does raise the question of how to deal with arrays whose size is determined at run-time. As this is a fairly general requirement occam has a protocol which corresponds to a counted array. When a message is passed on a channel with a counted array protocol, the length of the array is communicated and then the array is communicated. Suppose in the previous example, the string was to be displayed, not on a 12-character LCD display, but as a line on a terminal. We might then declare a channel terminal as

```
CHAN OF INT::[]BYTE terminal :
```

indicating that inputs would be of the form

```
terminal ? count::array
```

which first receives the number of elements to be input into the variable count and then inputs into the first count elements of array. Similarly, outputs would be of the form

```
terminal ! count::array
```

which first outputs the value of count and then outputs the first count elements of the array array.

Where a channel has been declared with a counted array as its protocol, some checks can be made at compile-time but others must wait until run-time. For example, with the channel terminal declared above, the compiler would reject the output

```
terminal ! "Hello world!"
```

as it is not of the correct form. However, it would accept the following

```
[4000]BYTE lengthy :
SEQ
  ...
  terminal ! SIZE lengthy::lengthy
```

which would cause a run-time error when input by the following fragment of program

```
[80]BYTE line.buffer :
SEQ
  ...
  terminal ? count::line.buffer
```

as the compiler inserts code to check the value of count against the length of the array line. buffer.

In addition to the protocols described above, called simple protocols, which permit a single item to be communicated, occam 2 has sequential protocols which permit a specified number of items to be transmitted by a single input or output. Suppose that we wanted to extend the previous example so that the message passed along the channel terminal specified the line on which the string was to be displayed. We want to send messages which first send the line number and then the text to be displayed. We can name a suitable protocol and then use it to declare the channel. (The previous examples have used simple protocols which do not require naming).

```
PROTOCOL line IS INT; INT::[]BYTE :
CHAN OF line terminal :
```

As a result of this declaration the compiler is able to check that all communications are of the correct form. For example, the following would be detected as an error

```
terminal ! 12::"Hello world!"; line.number
```

since the order of the line number and the line have been swapped.

Often a single channel is used to pass messages with different structures. For example, suppose we are writing a program to control a pen plotter which has a number of simple operations of the form 'pen up' or 'pen down', and a single draw operation 'move pen' which requires a pair of co-ordinates. We can indicate that we wish to send messages of two different structures by using a variant protocol. In this case we would declare a protocol plotter. control consisting of two tagged protocols.

```
PROTOCOL plotter.control
  CASE
    simple.command; INT
    move.command; REAL32; REAL32
```

The compiler only permits outputs which first output one of the tags, followed by an output matching the remainder of the tagged protocol. For example, the following output will cause the plotter to move to the origin

```
plotter ! move.command; 0.0(REAL32); 0.0(REAL32)
```

If we had made a mistake here such as omitting the move. command tag or sending the co-ordinates as integers, the compiler would detect the error.

An input on a channel which has a variant protocol is necessarily more complex than an output. The actual form the input will take depends on the tag received. To cope with this occam 2 has a 'case input' which first inputs a tag and then selects a matching input and then executes an associated process. For example, the program which actually drives the plotter would have an input such as

```
plotter ? CASE
  simple.command; command
    execute.command(command)
  move.command; x; y
    move.pen.to(x, y)
```

When this is executed a tag is input from the channel plotter and used to select the matching input. For example, if the tag input is move. command then an input to x and y will occur, followed by the execution of the procedure move. pen. to.

It is not necessary to list all possible tags in a case input. When a case input receives a tag which does not match any of the tagged inputs this is treated as an error. There are occasion where a program is expecting a specific tag to be received; in these cases a special form of input can be used. For example, if the pen plotter driver is expecting a simple. command then the program would look like

```
SEQ
  ...
  plotter ? CASE simple.command; command
  ...
```

This special case input is equivalent to

```
plotter ? CASE
  simple.command; command
    SKIP
```

4 Numerical behaviour

The numerical behaviour of operations in occam 2 is well defined. Usually overflow, division by zero, et cetera are treated as errors. However, it is recognised that sometimes it is necessary to perform calculations where these events are not considered to be errors. To this end occam 2 provides the PLUS, MINUS and TIMES operators which are unchecked.

The presence of the large number of concrete data types in the language raises the question of how constant values should be represented. In occam 2, only INT literals expressed as undecorated decimal strings (eg 123) or as hexadecimal strings (eg `#FA77FE16`), BYTE literals expressed as a quoted character (eg 'Z') and the BOOL constants (TRUE and FALSE) take their types implicitly; all other constants are explicit about their type. Whilst the requirement for explicit typing may seen unnecessary it does ensure that arithmetic on constants is performed correctly. For example, the result of the calculation 16777216.0 + 1.0 depends on whether the numbers are interpreted as REAL32s (in which case the result is 16777216.0) or as REAL64s (in which case the result is 16777217.0). It is for similar reasons that occam requires that all conversions between types is stated explicitly.

A similar decision has been taken in deciding to perform arithmetic according to the type of data on which it is being performed. All intermediate results are calculated as if they were of the same type as the result. This is unusual; it is quite common for the intermediate results of floating point

calculations to be held in an extended format. Whilst this may seem advantageous, it actually has two important drawbacks. The first is that it can lead to 'double rounding' and thus a less accurate result than if the arithmetic were performed to the correct precision. For example, in the program

```
VAL REAL32 a IS 2-100(1+2-23) :
VAL REAL32 b IS 2-27(1-3 x 2-23) :
r := a * b
```

a different result from that obtained by rounding straight into single length format is obtained if the calculation is first rounded to an extended precision and then into the correct precision. The second drawback is that storing a result becomes an arithmetic operation which undermines the substitution semantics of the language. This has important consequences as various common optimisations, such as common sub-expression elimination, would not longer be valid. For example, the two programs below would not be equivalent

```
REAL32 dummy :
SEQ
  dummy := x * y          SEQ
  a := dummy + a            a := (x * y) + a
  b := dummy + b            b := (x * y) + b
```

5 Abbreviations

The occam 2 language defines procedure calling in terms of 'abbreviation' and textual substitution. An abbreviation enables a name to be given to a variable or array element or to an expression. For example

```
INT element IS array[subscript] :
```

introduces the name element to identify the array component array [subscript] and

```
VAL INT twice.x IS 2 * x :
```

introduces twice.x as a name for the expression 2 * x.

In order to keep the semantics of abbreviation simple, and the implementation of abbreviation and parameter passing efficient, various rules concerning

abbreviations are enforced. One such rule is that the abbreviation of an expression is only valid if its scope contains no assignment to a variable in the expression. For example, consider the following program

```
VAL x IS y[i][j] :
SEQ
  ...
  z := x
  ...
```

The rule mentioned above ensures that there are (at least) three possible implementations of the abbreviation. The first simply replaces the occurence of x in z := x with y[i] [j]. The second assigns the value of y [i] [j] to a new 'variable' x when the abbreviation is executed and uses that variable in the'assignment. Finally, the third sets up a pointer to y [i] [j] when the abbreviation occurs and de-references that pointer when the assignments occurs.

One important consequence of defining parameter passing in terms of abbreviation is that VAL parameters can be passed either by copying the value (suitable for single word values), or by passing a pointer to the value (suitable for arrays).

6 Alias checking

Aliasing occurs when, within a scope, there are two or more names which identify the same object. When aliasing is present, the meaning of programs becomes obscure, because assignment to one name can affect the value of another name.

For example, the following procedure clearly leaves the value of its parameter x unchanged (note that the use of the PLUS and MINUS operators ensures that arithmetic overflow is not a problem)

```
PROC nonsense(INT x, VAL INT y)
  SEQ
    x := x PLUS  y
    x := x MINUS y
```

as is demonstrated by the following expansion of nonsense(n, 3)

```
INT x IS n :
VAL INT y IS 3 :
```

```
SEQ
  x := x PLUS  y
  x := x MINUS y
```

which is equivalent to

```
SEQ
  n := n PLUS  3
  n := n MINUS 3
```

which can be shown to be equivalent to n := n which is, in turn, equivalent to SKIP.

However, consider what would be the expansion of nonsense (n, n)

```
INT x IS n :
VAL INT y IS n :    -- invalid abbreviation
SEQ
  x := x PLUS  y
  x := x MINUS y
```

which would be equivalent to

```
SEQ
  n := n PLUS  n
  n := n MINUS n
```

The value of n after this instance of nonsense, were it valid, would be 0. Similarly, the instance nonsense(i, v[i]),were it valid would be equivalent to

```
SEQ
  i := i PLUS  v[i]
  i := i MINUS v[i]
```

the effect of which is very difficult to predict as in each of the assignments v[i] would probably reference a different component of v.

It is now recognised that aliasing can be the source of particularly insidious program bugs and, to counter this, aliasing is forbidden in some modern languages, for example, Euclid [3]. In occam 2, aliasing is restricted, not only for the reason outlined above but also to simplify checking the validity of parallel constructs. The rules imposed in occam 2 forbid the use of an element which been has abbreviated within the scope of that abbreviation. In the expansion of the instance of nonsense (n, n) given above the abbreviation

```
VAL INT y IS n:
```

is invalid because the name n has occured on the right hand side of an abbreviation which is currently in scope.

The majority of the anti-aliasing rules of occam 2 can be checked at compile time, however, those which permit an array to be used in a second abbreviation provided that the same element of the array is not abbreviated can require run-time checking. For example, consider

```
first  IS order[1] :
second IS order[2] :
```

which can be checked at compile time. However, the abbreviations

```
first  IS order[1] :
n.th   IS order[n] :
```

cannot as the second abbreviation is only valid if n is not equal to 1. The compiler will insert code to check this at run-time. (Although it may seem strange to perform this sort of check at run time, rather than compile-time, it is really no different from range checking subscripts at run-time!).

The imposition of rules forbidding aliasing does have a perhaps unexpected impact in the use of procedures. The anti-aliasing rules require that when calling a procedure all non-VAL parameters are distinct, and are distinct from any VAL-parameters. These rules can lead to some procedure instances being unexpectedly rejected. For example, the procedure

```
PROC factorial(INT result, VAL INT argument)
  SEQ
    result := 1
    SEQ i = 1 FOR argument
      result := result * i
:
```

returns as its result the factorial of its argument. (Note that a negative argument will cause the replicated sequence to behave like STOP). The instance factorial (result, 3) will set result to 6. However, consider factorial (n, n) which is supposed to set n to n factorial. This instance is, in fact, illegal according to the anti-aliasing rules. The reason for this can be seen if the instance is expanded

```
INT result IS n :
VAL INT argument IS n : -- invalid abbreviation
SEQ
  result := 1
  SEQ i = 1 FOR argument
    result := result * i
```

which, if it were legal, would be equivalent to

```
SEQ
  n := 1
  SEQ i = 1 FOR n
    n := n * i
```

which, in turn, would be equivalent to n := 1, not n factorial! To get the effect originally desired we have to write

```
INT temp :
SEQ
  factorial(temp, n)
  n := temp
```

The explicit introduction of temporary variables is undesirable and can be avoided in occam 2 because of the presence of functions. These permit us to define

```
INT FUNCTION factorial(VAL INT argument)
  INT product :
  VALOF
    product := 1
    SEQ i = 1 FOR argument
      product := product * i
    RESULT product
:
```

and to write n := factorial(n).

In occam 2 the functions are proper functions; they are side-effect free and deterministic. This is of great practical importance as it means that the compiler can compile replicated alternatives. Consider, for example the following alternative

```
ALT i = 0 FOR n
  f(i) & c ? a
    P(i)
```

where n is a variable. The occam compiler will generate code which evaluates each function instance, f M, twice; once when enabling the guards, once when disabling. Similarly, a compiler which used a polling implementation of alternative would also be correct.

7 Checking the validity of parallel constructs

The occam 2 language specifies that if a variable is assigned to or is used in an input then that variable may not be used in any parallel process. Thus the compiler will reject a program such as

```
PAR
  x := 42
  x := 69
```

However, consider the procedure parallel.assignment

```
PROC parallel.assignment(INT x, y)
  PAR
    x := 42
    y := 69
:
```

The validity of any instance of this procedure will depend on the parameters used in that instance. This suggests that the compiler must check the validity of each procedure instance by substituting the parameters into the body of the procedure. However, the fact that alias checking is performed means that the compiler can check the validity in two stages.

During the first stage of checking each procedure is checked on the assumption that all parameters and free variables are distinct. This will accept the procedure parallel. assignment but, for example, would reject

```
PROC invalid.parallel.assignment(INT x, y)
PAR
  x := 42
  x := 69
:
```

The second stage of checking is performed by the alias check which occurs for each procedure instance. For example

```
parallel.assignment(x, y)
```

would be accepted, but

```
parallel.assignment(x, x)
```

would be rejected.

It is important to notice that little more information is required about a procedure in order to perform parallel disjointness checking than is required for simple type-checking of its parameters. This opens the possiblity of constructing a completely secure system for the dynamic loading and execution of procedures.

8 Run-time error handling in occam 2

When a language such as occam 2 is used for the programming of secure or reliable systems, the behaviour of that system when an error occurs is of great concern. There seems to be no single method of dealing with errors,which is universally applicable to all systems. For this reason, occam 2 specifies that run-time errors are to be handled in one of three ways, each of which is suitable for use at different times.

The first mode is to ignore all run-time errors. This is potentially very dangerous and it is to be hoped that this will, one day, be made illegal except for systems which have been proved to be correct. This mode will most probably be used for benchmarking.

The other two modes detect run-time errors and prevent them from corrupting non-errant parts of the system. The first of these respectable modes causes all run-time errors to be signalled and to bring the whole system to a halt. This is known as 'halt' mode. In this mode the primitive process STOP is treated as if it caused an error. This mode is extremely useful for program debugging and is suitable for any system where an error is to be handled externally. For example, in at least one existing automobile engine management system, if the processor signals an error then the system reverts to its default settings by external analogue circuitry.

The second of the respectable modes, 'stop' mode, allows more control and containment of errors than does 'halt' mode. In stop mode all errant processes are mapped onto the process STOP. This will have the effect of gradually propagating the STOP process throughout the system. Although, at first sight, this does not seem very useful, it is possible for other parts of system to detect that one part has gone wrong, for example, by use of 'watchdog' timers. This allows multiply redundant systems, or gracefully degrading systems to be constructed.

9 Conclusion

The design of the occam 2 programming language has been influenced by the need to ensure that programming errors are as difficult to make as possible and that when they are made they should be detectable. The properties of the data types in the language have been carefully specified to ensure that they are consistent with the semantics of occam. The use of channel protocols makes possible the detection of many programming errors at compile time and ensures that total security can be attained at run-time with little cost. The insistence that names are not aliased detects some particularly obscure programming errors and greatly simplifies checking the validity of parallel constructs.

References

[1] Occam 2 language definition, David May, INMOS Limited

[2] Occam 2 Reference Manual, INMOS limited. 1987.

[3] Compile-Time Detection of Aliasing in Euclid Programs, James R. Cordy, Software - Practice and experience, Vol. 14(8) pp 755-768.