AN EVALUATION OF THE OCCAM2 PROGRAMMING LANGUAGE

Robert D. Carleton
April 1993

Occam2 was developed as an entirely new computer programming language to compliment an entirely new computer architecture. Without benefit of a mellifluous name such as UCSD Pascal (an early, 1970's, variant of the then-fledgling language), Modula2, or C++, superficially, at least, the language has a low probability of ever enjoying widespread acceptance. While more substantive reasons exist for Occam's potential failure in the marketplace, the language, nonetheless, embodies several concepts likely to be found in future computer programming languages. Still, one may wonder why a new programming language or for that matter a new computer architecture is desirable or even necessary.

The impetus for the development of new imperative computer programming languages in the recent past has stemmed from the need to make the most efficient use of existing hardware. Frequently, rather than give rise to a completely new language, improvements in hardware design and increased memory capacity have resulted in adaptation of and enhancement to existing familiar imperative languages. At present, computer designers have turned their attention away from the development of improved processors and instead are using existing or slightly modified processor designs in combination with one another to produce parallel processing computers. Efforts in new supercomputer design have also been scuttled in favor of this approach. This departure from single to multi-processor architectures represents a significant change from the traditional and highly successful sequential aspect of John von Nuemann's computer model.

It is intuitively, but perhaps deceptively, easy to predict the inevitable success of parallel architecture. Regardless of how far engineers can push the speed of an individual processor, whether CISC or RISC based, 10 of them running in parallel will still potentially execute 10 times as many instructions per second. Further, from a purely intuitive standpoint we can recognize the efficacy of this approach. Time, while having been shown to be one of but two forms of pure a-priori knowledge (Kant 1855), can be adequately described as being composed of many concurrent events, each of which is happening against a sequential backdrop. Thus while we program a line of code, write a sentence of prose, or breathe one breath after another, we can also realize that others are engaged in the same, or different activities, concurrently. Finally, all appeals to the somewhat metaphysical aside, the concept of concurrent processes is currently being used to implement multi-tasking on some advanced operating systems with great success. Therefore, to adequately model the real world it would seem intuitively obvious to have multiple processors all working at the same time on the same program. Currently no widely accepted imperative programming language exists to take advantage of this architecture, a fact which may not be surprising given the necessity for programmers and computer designers alike to substitute comfortable sequential approaches to solving problems in favor of a concurrent approach.

Clearly then, what is needed is a programming language which takes advantage of this new architecture and this was precisely the reason Occam was developed. The language was
created for use solely in a parallel environment and is based on communicating processes (Graham & King 1990). It has been described by its developers as a high-level language comparable to C or Pascal. Does this language represent the future of imperative languages? Further, how does it compare with existing high-level languages and how universally applicable might the language be?

To answer these questions, a complete evaluation of Occam2 using the criteria suggested by Sebesta (1993 a) will be employed. Additional criteria will be added to evaluate the degree to which the language supports parallelism, particularly multiple concurrent processes, multiple processes, multiple processors, inter-process communications, and the control of process scheduling. Before beginning the evaluation as outlined, let us first examine the platform under which Occam1 and 2 were developed with attention to the primitive constructs supported by its native processor family for implementing parallelism. It must be noted that, while perhaps relevant to an investigation of parallel computing languages, this is an area of Computer Science more appropriately examined under the topic of operating systems, due to the emphasis upon managing resources. Thus its direct treatment will be brief. However, it is decidedly necessary to achieve an overall understanding of how this language performs functions normally assumed by operating systems, not only because it represents a radical departure from the von Neumann architecture for an imperative programming language but also because the language has incorporated features of some advanced operating systems which have been successful in multi-tasking.

Occam was developed in parallel with the hardware with which it was to be used by INMOS Ltd., part of the SGS-Thompson Micro-Electronics group. Their first product, the IMS T414, introduced in 1986 using Occam1, has today given way to the TDS (Transputer Development System) using Occam2 and processors which contain four serial links for inter-processor communication. The name Occam derives from William of Occam (1270-1349), an English scholar and philosopher, to whom is attributed the saying, 'Entities are not to be multiplied without reason'. The underlying entity, or processor, to be multiplied in this case is an INMOS design of which several variants are available.

Coming in either 16 or 32 bit widths, with or without on-board math co-processors, the INMOS processor’s instruction sets are similar in quantity to CISC architectures. Significant differences are attributable to those instructions added to support process scheduling and communication. Several micro-coded instructions exist for the use of two process queues, each consisting of a pair of on-board registers corresponding to a low and high priority queue. Elements of the queues are active processes; two pairs of registers point to the 'fronts' and 'backs' of processes. A timer and timer process queue accompanies each of the pairs, allowing control of the processes by the process scheduler.

A process is the central concept of the transputer and thus of Occam as well. A process is the individual thread of control that is followed during execution of a program or of several programs. In an operating system environment, the concept of a process controls multi-tasking, switching between processes among several programs, whereas in Occam the concept is used to define the control processes within a single program. Communication between processes is accomplished through what are called 'channels'. A channel may be used to both convey information between processes in the same processor as well as transmit information to and from different processors.

With this brief background on the hardware and operating systems issues relevant to the Occam language in mind, a description of the programming language can proceed with special emphasis placed upon the factors influencing the read- and writability and reliability. Following this analysis will be an discussion of the factors influencing the cost of Occam
usage. It will be convenient to evaluate certain aspects of the language as they are described however, an overall evaluation will follow after all criteria have been respectively applied.

All Occam programs are built from combinations of three kinds of primitive processes: assignment, input and output. Before proceeding with the objective-based criteria analysis, it is appropriate to briefly examine these processes termed 'actions' by INMOS. The ' := ' symbol, familiar from Pascal, serves as the assignment statement operator. The modified* BNF formal syntax description of multiple assignment extends beyond the syntax for simple assignment. It can be considered the most important single statement in an imperative language and for occam2 is (INMOS 1988p.6):

\[
\begin{align*}
\text{assignment} & = \text{variable.list} := \text{expression list} \\
\text{variable.list} & = \{1, \text{variable}\} \\
\text{expression.list} & = \{1, \text{expression}\}
\end{align*}
\]

*the '1' notation used in \{1,variable\} or \{1,expression\} refers to a list of one or more variables or expressions, respectively, separated by commas. The syntax for expressions and operands are, again using the modified BNF form (INMOS 1988p.43):

\[
\begin{align*}
\text{expression} & = \text{unary.operator operand} \\
& | \text{operand binary.operator operand} \\
& | \text{conversion} \\
& | \text{operand} \\
\text{operand} & = \text{element} \\
& | \text{literal} \\
& | \text{table} \\
& | \text{expression}
\end{align*}
\]

Operator associativity is defined with the use of parentheses, much in the manner of ADA, where the expression inside of the parenthesis is considered to be an operand of the operator immediately following it. All of the elements of an expression, the operands, must be of the same type and produce a result of that type. Comments are denoted using the ADA style '--comment'. Statements appear one per line but have no termination or separation symbol such as the ' ; ; '. The semi-colon instead is used as the sequential protocol separator. This is clearly a detriment to readability and writability for those accustomed to using this symbol to separate program statements. The syntax for the language is very similar to that of Pascal in general. There are exceptions which will be enumerated as they arise.

The input process inputs a value from a channel into a variable using the '? ' symbol. Channels communicate values by providing unbuffered, unidirectional point-to-point communication of values between two concurrent processes. The output process is very similar, designating the output destination of a variable from a channel using the '!' operator. From these three primitive actions larger processes are built by combining smaller processes with the result termed a 'construction'. With the assignment statement, expression syntax and two basic control statements defined, the data types used by Occam2 should next be considered.

The data types in Occam2 may be assigned to a variable in either an assignment statement or input. The primitive data types available are:

1. boolean, consisting of true and false
2. byte, consisting of integer values from 0 to 255
3. integer, consisting of four types, all represented in 2’s compliment, with values ranging from the implemented word size, to 16, 32, and 64 bit unsigned maximums,
4. real, consisting of either 32 or 64 bit IEEE/ANSI standard 754-1985 representation.
These types are comparable to most high-level languages, however there is no unsigned integer, as in C, thus limiting the absolute range of the type.

Names of variables must start with an alphabetic character and have no length restrictions. Variable names are case-sensitive and consist of a sequence of alphanumeric characters. The only permissible connector is the dot. All keywords are reserved and thus may not be used by the programmer for anything other than their intended purpose. Reserved words are exclusively upper case. A complete list is included in the appendix. Aliasing is supported through the use of abbreviation. Several aspects of the naming conventions in Occam2 are detrimental to read- and writability. The dot connector symbol is clearly untenable due to likely confusion as a record structure qualifier. The support for aliasing is unrelated to the existence of either pointer variables or record structures. An abbreviation is assigned a value by the use of the 'VAL' keyword and may be an expression or name with the keyword 'IS' in between. An abbreviation may also refer to an element of an array. Thus, this use of aliasing is merely shorthand and detracts from the overall readability of the language. The case-sensitivity yields a reduced level of readability when alternative spellings of minuend, Minuend, and MINUEND may all refer to different quantities.

Variable types must be explicitly declared and are therefore statically bound and storage is allocated at compile time by a multi-pass method of compilation. Variables may be only static, precluding the ability of the programmer to use recursive looping methods. This is a serious detriment to writability because certain algorithms, for example 'The Tower of Hanoi' are easily solved when recursive loop structures are employed, but become extremely complex when only iterative loops are permitted.

There is no provision for explicit or implicit dynamic variables. Some of the reasons for this will be discussed further when channels are considered in more detail and when the language’s support for abstraction is examined. The strong versus weak data typing of Occam2 is an issue that is influenced most heavily by the abbreviation concept which is functionally the same as the Fortran 'Equivalence' keyword. 'Val'd variables are not subject to type checking. Variables cannot be simultaneously declared and initialized.

The scoping rules for Occam2 variables are static with the exception of the case of procedure parameters passed through a channel. Otherwise, variables, channels, and named objects are local to the process that immediately follows their specification. A global variable cannot be declared for several processes because any local re-definition of the variable will supersede prior declarations of the variable. The lifetime of an Occam2 variable is the length of time a process is active. Constants may have a lifetime of the entire program. The referencing environment of a statement in Occam2 is the process.

There is such a paucity of data types in Occam2 that the claim of 'high-level language' could be argued. There are no user-defined types or record structures available to the programmer. Multi-dimensional arrays are available and use the familiar '[ ]' symbols to enclose the array upper bound with an assumed lower bound of zero. There is no dynamic allocation of an array space during run-time, the size of the array must be known at compile time and cannot be changed during execution. Array values can be generated in a somewhat unique fashion. An expression can be used to fill in the values of an array using a table, then the array can be assigned to a variable. Array types can be real, integer, boolean or character. The absence of user-defined types or record structures makes Occam2 comparable to Fortran77 or earlier. These features have assisted programmers in making code more readable and writable and have also been an aid to structuring programs. These features should be included in new languages claiming to be 'high-level'.

109
There is no high-level formal support for pointer types in Occam2, however at a very low level one may think of having access to addresses and perhaps performing address arithmetic. This feature of the language has apparently been found most useful when specifying hardware links. It can be used to directly probe or address memory. With \texttt{i} declared as an integer, the 'PLACE' keyword, as used in the statement, 'PLACE \texttt{i} AT 28', results in the integer \texttt{i} being loaded into storage location \texttt{0x8038} on a 16-bit processor. This feature, while perhaps necessary for addressing hardware links, is extremely detrimental to reliability. Two processes can effectively place variables at the same address and access them with no control.

The statement-level control structures in Occam2 are an inherent part of the construction and cannot be evaluated without reference to concurrency issues. The first of these to be examined is the 'SEQ' directive. SEQ is used to combine processes into a construction in which one process follows another. SEQ combines two or more actions which are to be performed sequentially, that is, not in parallel. There are two conditional statements and one selection statement, 'IF', 'WHILE', and 'CASE', respectively. The conditionals combine a number of processes, each of which is controlled by a loop variable, evaluated to be true or false, and thus serving to continue or terminate the loop. When the conditional is evaluated as false, an implied 'STOP' is invoked. 'STOP' is a primitive process which starts, performs no action and never terminates. The loop may also be exited by use of the 'SKIP' statement as in the following statement (Pountain/May 1987 p.41):

\begin{verbatim}
IF x < y
  x := x - 1
x >= y
SKIP
\end{verbatim}

Here, the primitive process SKIP, starts, performs no action, and never terminates. The difference between the two is further illustrated when used with the SEQ process as in (INMOX 1988 p.11):

\begin{verbatim}
SEQ
keyboard ? char
SKIP
screen ! char
SEQ
keyboard ?
STOP
screen ! char
\end{verbatim}

The first sequence executes the input keyboard ? char, then executes SKIP, which performs no action. The sequence continues, and the output screen ! char is executed. The second SEQ performs the input keyboard ? as before, then executes STOP, which starts but does not terminate and so does not allow the sequence to continue. The output screen ! char is never executed. Both the SEQ process and the IF conditional process may be replicated or caused to invoke themselves repeatedly. In the case of SEQ this is accomplished by using the keyword FOR as in (INMOS 1988 p.12):

\begin{verbatim}
SEQ i = 14 FOR 2
stream ! data.array[ i ]
\end{verbatim}

In the case of the conditional process IF (INMOS 1988 p.12):

\begin{verbatim}
IF
  IF i = 1 FOR length
    string[ i ] <> object[ i ]
  found := FALSE
TRUE
  found := TRUE
\end{verbatim}
Here, if the length variable has been initialized to 4, the object array of length one will first be compared with the string length array value of 1 and the process will be repeated for values of 2, 3, and 4. It should be noted that the use of 'TRUE' here amounts to an else or otherwise clause.

Selection is performed by the familiar 'CASE' conditional which combines a number of options, one of which is selected by matching the value of the selector with the value of a constant expression associated with the option. The CASE statement accepts an ELSE clause to handle all other possibilities. The 'WHILE' loop conditional performs exactly as would be expected in the Pascal programming language with SEQ creating the construction as in this example (INMOS 1988 p.14):

```
WHILE buffer <> eof
  SEQ
    in ? buffer
    out ! buffer
```

The 'PAR' or parallel construction is one of the most useful constructs for concurrency of the language by combining a number of processes which are to be performed concurrently. The use of this construct is best seen by example (INMOS 1988f):

```
WHILE next <> eof
  SEQ
    j := next
    PAR
    in ? next
    out ! j * j
```

Here the parallel construction is permitting communication and computation to proceed together. The next value to be processed from one channel is being input while the last value is being processed and output on another channel. PARs may also be nested and used to invoke themselves.

The final method used in a construction is the 'ALT' or alternation, which combines a number of processes 'guarded' by inputs. The term guard refers to a construct which, at base, consists of processor-level instructions regarding parameters of the input. These are a boolean part, then either check for a channel, a timeout, or nothing at all. The concept of a guard is similar here to the one used by the ADA programming language for message passing in concurrent programs (Sebesta 1993b). Each of these latter parameters has its own instructions to enable or disable a channel address, enable SKIP or STOP respectively. Fortunately for the programmer these low level instructions are filled in automatically when the ' ? ' or ' ! ' is used as in (INMOS 1988 p.18):

```
ALT
  left ? packet
  stream ! packet
  right ? packet
  stream ! packet
```

Here, the effect is to merge the input from the two channels right and left on to the channel stream. If the left channel is ready, and the right channel is not, then the input left ? packet is selected and any associated process performed. The converse is also true, but if neither input is ready then the alternation waits until an input becomes ready. If both are ready only one is performed and which one is undefined! This is truly untenable and further does not enhance reliability to a great extent. Even if code is written to overcome the situation and
eventually input the data from the un-selected channel, both programmers and engineers detest situations in which an outcome is undefined. The keyword 'PRI' may be used to predicate either ALT or PAR. If a process is given priority with PRI then it is executed first and lower priority processes may begin or resume only when the the prioritized process is unable to continue. This ameliorates to some degree the aforementioned problem with ALT but documentation with Occam2 does not require nor suggest its usage there.

The use of procedures and functions are supported by Occam2 and are also considered processes. Procedures are declared with the 'PROC' statement with syntax identical to that of Pascal. Parameters in all Occam2 procedures are nearly equivalent to VAR Pascal parameters or other call by reference types. Should it be preferable to have a parameter passed by value, the programmer can implement this by decanting the parameter into a local variable. One other rule for procedures obtains and that is a channel parameter may not be used for both input and output in the same procedure. The parameter passing method can, however, still be considered an in-out mode method. Called subprograms do have access to the parameter for both input and output, they simply must do so by using different channels. Aliases can be created for parameters by using the abbreviation statement and thus collisions can occur and reduce reliability.

Functions are special kinds of processes known as value processes. A value process produces a result of primitive data type (but not of array type), which may appear in expressions. Value processes may produce more than one result, which may be assigned in multiple assignment. Occam2 functions are not subject to side-effects because they may only assign to variables of local scope. Value processes may not contain inputs, outputs, alterations, or parallels. A procedure called within a function must also obey these rules. The keyword 'RESULT' is used to return a value from a function. The function process in Occam2, like the procedure process, is subject to a morass of rules which clearly undermine writability.

Two additional structures, 'PROTOCOL's and 'TIMERS's are used by Occam2 to facilitate communication between concurrent processes and synchronization of two concurrent processes, respectively. Recall that a channel communicates values between two concurrent processes. The format and data type of these values is specified by the channel protocol when the channel is declared. Each I/O operation must be compatible with the protocol of the channel used. Type checking is employed to insure compliance. A protocol with an array type would be, for example, CHAN OF [24] BYTE string, which would declare a channel with a byte array identified by the name string. Sequential protocols consist of a sequence of simple protocols such as was defined in the 'CHAN' example above. The multiple type and name fields are separated by a semi-colon and have the result of passing pairs, triplets, or other multiples along the channel. The case statement may also be used to create a variant protocol useful for differentiating several types of I/O. Finally an 'anarchic' protocol exists to cover the case of unknown I/O, although how unknown output comes about is uncertain. The keyword 'ANY' takes input (or output) and 'casts' it to the type of the receiving variable. Type checking, in this case, is suspended.

Timers are the remaining specific of Occam2 to be discussed. Timers produce values representative of the passage of time that can be accessed by any number of concurrent processes. A timer receives a value from the timer named on the left side of the input symbol '?' as in, clock ? t, wherein t is declared as an integer and clock has been declared as a 'TIMER'. The keyword 'AFTER' may be introduced between the '?' and the 't' above to produce a delay. The usage of the timer is for hard coding or enforcing synchronization. Exactly when it is appropriate to use 'Timer' is left to the programmer's needs. It would seem to be especially useful in real-time data acquisition.
The key to evaluating Occam2 fairly may lie in comparing it with early high-level languages rather than those which predominate today. However, let us now look at the language more completely using the Sebesta criteria. Leaving the issue of concurrent programming aside for the moment, programming in the Occam2 programming language would seem to have some serious drawbacks. The lack of data structures, such as user-defined types and record structures, common to many of today’s high-level languages, certainly is not an advantage to read- or writability. Support for abstraction is similarly not high; a binary tree for example, would likely be implemented with three parallel arrays, rather than with pointers and a record structure. Orthogonality suffers from the rules associated with functions and procedures but is generally good for types since, for example, an array of arrays is permitted. Type checking is supported, but may be circumvented with the use of aliasing which is not fully restricted. Simplicity is high, but is degraded somewhat by the language's expressivity. The lack of an underscore connector for variables and the use of the dot for the purpose of connecting words in a descriptive variable name, dampens expressivity and may cause confusion with record qualifiers, reducing overall readability. The lack of distinct 'begin' and 'end' keywords for a process, sequential or parallel, while a plus for simplicity, cannot be considered a positive aspect of the syntax design. The control statements, neglecting, for the moment, those exclusively used to exploit concurrency, are straightforward and behave much like those of familiar high-level languages, especially Fortran.

This observation, that the language favorably compares with Fortran, is perhaps, easily understood. Fortran was originally designed for numerically intensive programming applications and it still enjoys widespread use in the scientific community for such purposes. The real world problems which will benefit most from running in concurrent form will be similarly mathematically demanding. For instance the University of Colorado at Boulder has been awarded a grant to develop the software for a massively parallel computer system which is to be used in modeling the characteristics of aircraft at hypersonic speeds. A language used for this purpose would not necessarily find the inclusion of a record structure paramount. Occam2 designers were probably more concerned with the scientific and academic end users purposes than those users considered a part of the general marketplace. There are, however, examples of traditional programming problems making effective use of Occam2. Conway's ubiquitous game of 'Life', Huffman minimum redundancy coding, and parallel sorting algorithms have been implemented using Occam2 (Jones 1987). The 'Cost' of training programmers, writing code, and compilation should be considered in light of this scientific environment. Occam2 is a development tool and, with its similarity to Fortran, should allow those interested in concurrent programming to migrate fairly easily. INMOS has also made the cost of a TDS quite reasonable, thus overall cost should be low. If the concurrent programming techniques one acquires from this language are generalizable, then this can become a positive factor in the maintenance of all concurrent programs irrespective of a specific language implementation.

The ability of Occam2 to deal effectively with the problems inherent in concurrent programming was clearly a paramount concern of the designers. Occam2 was designed to be used with a physically concurrent system, one which employs multiple processors. The major problem faced is that of synchronization (Sebesta 1993 b.). Cooperation and competition synchronization are both required when data is to be shared. Cooperation synchronization is required when two or more processes, or tasks, must wait on one another before either can fully execute. This can mean that process A must wait for process B to complete before A can continue its execution. Competition synchronization is required between two or more tasks needing access to a resource which cannot be shared simultaneously. The concept of
liveness is introduced in this context to describe the characteristic of a process, or program, to eventually execute to completion. In the case of concurrent programs, this is an issue because of resource competition. There are instances where a control of a shared resource is given to one process and a competing process is forever denied access. This can lead to one of the most serious problems that concurrent language programmers, and multi-tasking operating systems, have to be concerned with. The problem is deadlock and can occur when two or more processes require mutually exclusive use of two resources. The process is then considered without liveness or "dead". Suppose each is granted access to one of the resources, but refuses to relinquish control until it is granted access to the other. The result is deadlock, and Occam2 has several ways to deal with this issue, but none are perfect. Let us now look at how Occam2 provides constructs to deal with this issue and generally facilitate concurrent programming.

First, Occam2 designates channels as the means by which inter-process and data communications take place. A channel may, at a given moment, be designated as input or output and the 'ALT' directive can be invoked to alternate between the two. The 'PRI' directive can be used to access the process queue and thus create a schedule of processes. The use of 'PRI' can assist the programmer in avoiding deadlock by giving preference to a particular process that is engaged in competition. A form of arbitration may be devised which resolves conflicts. Another resource Occam2 provides the programmer is access to the 'TIMER' structure. This is somewhat low-level in its use but could be beneficial in the deadlock scenario. If deadlock has been detected, or has been anticipated to be unresolvable using other techniques, a time-out could be imposed and steps taken to remediate the situation.

Which processor is to get which process is determined either by the main program, in which case a "placed par" directive is used, or at compile time when a configuration file is created and the programmer enumerates the parallel processes with a designated processor. In either case the avoidance of deadlock and the most advantageous division of parallel processes amongst processors is left to the programmer. While this can be seen as an advantage to the experimenter by providing great flexibility, the programmer may view the situation as anarchical. Some relief from the need to determine how to allocate processes between processors may be soon available. INMOS has announced that a "Transpiler" is under development which would look for the parallelism in OCCAM2 programs and configure them as efficiently as possible.

In summary, Occam2 provides the programmer with a Fortran-like environment in which to explore the issues of concurrent programming. It may be favorably compared with the introduction, in the late 1970's, of the Southwest Technical Products Inc. micro-processor based personal computer. This product allowed easy access to computers for experimenters much the way Occam2 allows easy access to the concepts of concurrent programming. Occam2 also provides many tools to facilitate resolving algorithms into concurrent processes (Eckhouse 1992). It contains several constructs to deal directly with the issues of competing and cooperating resource synchronization. However, these constructs may not be sufficient. Working with simulations of neural networks in Occam2, it was discovered that there exists a potential for communication deadlock as a result of the unbuffered communication among Occam2 processes (Barbosa & Lima 1990). These issues have been a great subject of interest in the academic community for some time. Simulation of discrete event systems have been employed to address several aspects of the deadlock problem and Occam2 has been a useful tool in such investigations (Righter & Walrand 1989). Occam2, while not a great high-level language nor a perfect concurrent one, is filling a niche by bringing powerful low-cost concurrent programming to the academic and scientific community.
REFERENCES:
Barbosa, Valmir and Lima, Priscila (1990) On the Distributed Parallel Simulation of Hopfield’s Neural Networks Software-Practice and Experience (pp 967-83) Vol 20 October