OCCAM 2: an overview from a software engineering perspective

OCCAM provides mechanisms for concurrency and communication as the basis of program development. **Russell Wayman** assesses how this approach distinguishes the language, particularly the latest version OCCAM 2, from other programming languages

The programming language OCCAM has been developed for the design and implementation of systems of concurrent processes communicating over channels. It is primarily intended for use as a programming language for the Inmos transputer. This paper has two aims: to review the major features of OCCAM, and to show how the language relates to various stages of the software development process. It also reviews some of the decisions in the design of the language in the light of the criteria outlined in the introductory paper to this series on high-level languages.

Software engineering **high-level languages** **OCCAM 2**

The programming language OCCAM was first introduced by Inmos in 1983; the initial version of OCCAM — known as OCCAM 1 — is described in Reference 1. Since its introduction OCCAM has been used in a wide variety of applications, such as embedded control systems, compilers, operating systems, financial modelling and scientific simulation. The language itself has developed further with the introduction of types (and a number of other features) in a new version called OCCAM 2[2].

OCCAM differs in many respects from the other programming languages (PASCAL, MODULA-2 and ADA) described in this series and it is worth mentioning how some of these differences arise. PASCAL was invented as a teaching language, to teach the principles of good program and data structuring. MODULA-2 and ADA are both derived from PASCAL, with the addition of features to aid in systems programming. In particular both MODULA-2 and ADA have added concurrency to allow applications to be split up into semi-independent processes. However, both languages have a form of concurrency which assumes that the aim is to have a set of processes sharing a single computer. OCCAM comes from a different direction, which lies in the conception of the transputer as a single-chip microprocessor intended for connection into collections of processing elements working cooperatively on a task. OCCAM is intended as a language for programming such multitransputer systems and the choice of features in the language has been motivated by the need for a distributed implementation. It has been heavily influenced by the work of Hoare on Communicating Sequential Processes (CSP)[3], which gives a mathematically-based notation for specifying the behaviour of parallel processes. OCCAM is based on the CSP model of computation, but with features chosen to ensure efficiency of implementation.

Inmos Ltd, 1000 Aztec West, Almondsbury, Bristol, BS12 4SQ, UK

ASPECTS OF OCCAM

This section discusses some aspects of OCCAM, beginning with the concepts of concurrency and communication which form the core of the language, and then going on to discuss program structure, data types and the basic constructs used for programming sequential processes. Some of the new language features introduced in OCCAM 2 are emphasized.

Concurrency

At the heart of OCCAM is its facility for expressing concurrency. OCCAM allows an application to be decomposed into a collection of parallel processes communicating over channels. A channel is an object which is used for communicating data between two processes, one of which uses it for input and the other of which uses it for output. Processes may not communicate by means of shared data, only by sending messages along channels. OCCAM compilers perform checks on a program, known as 'usage checking', to ensure that channels and variables are being used appropriately by parallel processes.

0141-9331/87/08413-10 $03.00 © 1987 Butterworth & Co. (Publishers) Ltd

It is worth emphasizing here the distributed nature of the OCCAM model of concurrency. If we have a program executing on multiple transputers, each with its own memory, we would expect that a process running on one transputer could only influence the behaviour of a process on another transputer by sending a message to it. On the other hand, if we have two processes running on the same transputer, it is certainly possible for them to communicate by shared variables. However, this is not allowed in OCCAM; message passing is the only form of communication between parallel processes. Thus the same model of concurrency is used both within processors and between processors. This has two advantages: it eradicates a class of programming errors associated with shared-variable communication, and it makes it easier to change the way that a particular program is split up between processors. However, it also has some implications for the approach to programming. Programs do not exist in a 'global environment'; they can only communicate with the outside world over channels. For example, if a program is required to store data in a filing system, that filing system is available over a set of channels connecting the program to the outside world.

The parallel construct PAR in OCCAM indicates that a number of component processes are to be executed together. For example

```
PAR
  editor(term.in, term.out)
  keyboard.handler(keyboard, term.in)
  screen.handler(term.out, screen)
```

shows the top-level decomposition of an editor, with separate processes to handle the screen and keyboard I/O. Channels 'term.in' and 'term.out' carry the information between the processes.

OCCAM programs lend themselves naturally to depiction, with boxes or circles to represent the processes and arrowed lines to represent the channels. Figure 1 shows a pictorial representation of the editor.

The parallel construct differs from the parallel constructs of the other concurrent languages described in this series in the level of granularity of parallelism which may be expressed. In OCCAM any two or more processes may be combined to run in parallel, allowing the expression of very fine-grain parallelism. For example, it is possible to specify a set of assignment statements to execute together in the following way:

```
PAR
  x:= 5
  y:= a + b
  z:= 0
```

It would not be normal practice to combine assignment statements in this way, as the result would be less efficient than sequential composition, but there are many circumstances where a parallel composition is the most natural. This is only made feasible by the careful choice of features in the language to ensure the efficiency of concurrency and communication.

At any particular time in its execution an OCCAM program may be thought of as a collection of parallel processes, with data flowing over the channels connecting the processes. In addition, parallel processes are combined in sequence, so that as a program executes, its parallel structure will change as processes fork and rejoin.

The following example is intended to indicate how sequential and parallel composition can be used together in a program. The example shows a parallel graphics pipeline surrounded by sequential phases during which its database may be updated. The process called 'view' reads the database and extracts the data associated with a viewpoint, sending the data down the channel 'view.out'. The remaining pipeline of processes performs a sequence of transformations on the data, until the final process, 'display', sends the results to the screen.

```
... Data base declaration
SEQ
  ... initialize database
  WHILE going
    SEQ
      PAR
        view(data.base, viewpoint, view.out)
        clip(view.out, clip.out)
        perspective(clip.out, persp.out)
        hidden.surface(persp.out, surface.out)
        display(surface.out, screen)
      ... update database
```

Figure 2 shows a pictorial representation of this program, the sequential composition shown by the boxes with arrows linking them. The parallel components are shown as circles, with arrows representing the channels between them. Note the use of the ellipsis (...) in the above example to indicate a section of program which is not shown in full. This convention is used in other examples in this paper.

A replicated PAR allows the creation of an array of parallel processes in a compact manner. For example

```
PAR i = 0 FOR 10
  element(chan.array[i], chan.array[i + 1])
```

creates a pipeline of processes (see Figure 3). The behaviour of each process is described by the procedure 'element'. Each element is passed two channels as parameters (one for input and one for output) selected from an array of channels 'chan.array'.

Communication

For a communication to occur on a channel between two processes, one process must output some data and the other process must input the data. In OCCAM, communication is *synchronized*, i.e. before the communication may occur, the outputting process must have reached its output action, and the inputting process must have

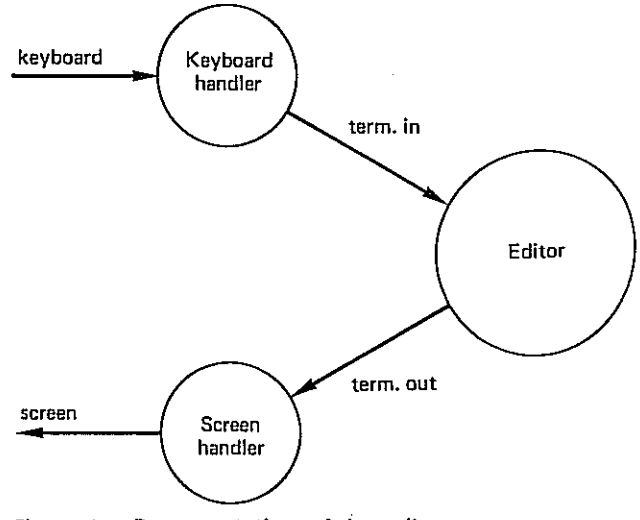

Figure 1. Representation of the editor

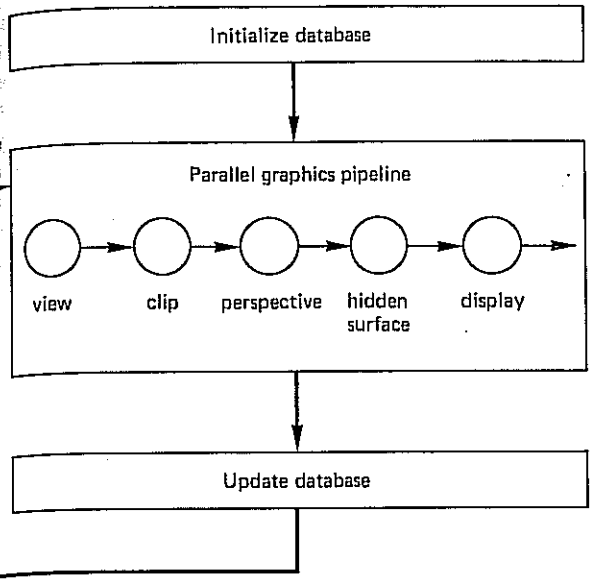

Figure 2. Representation of the joint sequential and parallel program example

reached its input action. The process which reaches its communication first is blocked until the corresponding action is reached at the other end of the channel. This means that communication can be implemented without any buffering of data within the channel.

The input and output actions in OCCAM are as follows.

- Input: an input process c ? v inputs a value from a channel c and assigns it to a variable v.
- Output: an output process c ! e outputs the value of an expression e to a channel c.

In OCCAM programs it is sometimes necessary for a process to input from any one of several other concurrent processes. For this purpose, OCCAM includes the alternation construct ALT, which executes one of a number of alternative processes depending on the readiness or otherwise of a number of channels. A channel is ready for input when the process at the other end has reached its output. For example

```
WHILE TRUE
  ALT
    left.chan ? x
      output ! x
    right.chan ? x
      output ! x
```

This process continuously merges two streams of data, one coming from the channel 'left.chan', and one coming from the channel 'right.chan', and outputs the merged data to the channel 'output'. Each branch of the ALT consists of an input action (called a 'guard') and a process to be executed if that guard is ready.

An ALT with more than one ready guard makes an arbitrary (or 'nondeterministic') choice between the ready alternatives. This is quite often misunderstood by programmers starting to use OCCAM; it does not indicate that a random choice is made which somehow guarantees the 'fairness' of the choice; it simply means that the

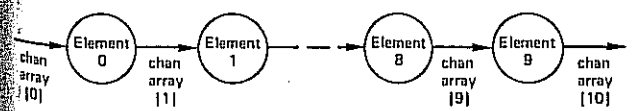

Figure 3. Pipeline of processes

programmer has no direct control over which guard is chosen. Nevertheless, it is possible for the programmer to add to the ALT in order to control the choice. Boolean expressions can be included in ALT guards to control which of the ready channels are considered in the choice. Only those guards for which the boolean expression evaluates to TRUE are considered.

Program structure

The structure of OCCAM programs is determined by two factors: process structure, which comes from the way that processes can be combined together in parallel to produce more complex processes, and procedural abstraction which, as in other languages, allows a self-contained or commonly occurring section of program to be given a name and then instanced when required by use of its name. Note that these two concepts are orthogonal; it is not necessary to give a section of code a name in order to run it as a process.

OCCAM constructs such as SEQ, PAR and ALT are used to compose simple OCCAM processes into more complex structures. The components of the structure are indicated by indenting them from the construct keyword. This indentation in OCCAM takes the place of the curly brackets of C or the Begin–End pairs of PASCAL. With these other languages, programming standards are often used to ensure that programmers follow a uniform style of indentation; in OCCAM the indentation is part of the language and is checked by the compiler.

Declarations of channels and variables may occur anywhere in a program, preceding the section of code to which they apply. The scope rules for OCCAM are simple, and are flexible in allowing the placement of declarations at the appropriate position in a program.

In OCCAM procedural abstraction is supported by the use of procedure and function declarations. As for variables and channels, there is great flexibility about where these declarations may be placed in a program.

A procedure declaration introduces a procedure name and a set of formal parameters which are substituted by the actual parameters at the instance of the procedure. An actual parameter may be the value of an expression, a variable or a channel (or an array of any of these). A procedure cannot be passed as a parameter to another procedure. Data parameters may be passed as values, in which case they may not be modified, or they may be passed as variables, allowing modification within the body of the procedure.

Function declarations are also supported in OCCAM 2. Functions are like procedures which return one or more results, and which may be used in expressions. Functions are not allowed to have side effects, i.e. they are not allowed to change the value of any variable external to them, or to engage in communication.

Recursive calling of procedures and functions is not allowed in OCCAM. Recursion is a powerful programming technique, particularly when used in conjunction with recursive data structures, but it places certain requirements on the runtime system of the language. The standard technique for implementing recursion in sequential languages is the allocation of memory from a single runtime stack; in OCCAM a different technique would be required, as the memory also needs to be divided between the concurrent processes. The absence of

recursion from OCCAM results from the need for an efficient runtime system, rather than from any fundamental difficulty with recursion in a concurrent language.

Types and data structures

In contrast to OCCAM 1, OCCAM 2 is a typed language, and both variables and channels must be declared with a type. The basic set of types in the language is

BOOL	Boolean
BYTE	Byte
INT	Integer
REAL	Real

In current implementations, integers may be of length 16, 32 or 64 bit. Reals may be of length 32 or 64 bit.

OCCAM 2 is strongly typed in that no automatic type conversion is provided in expression evaluation. The type declaration keywords may be used in expressions to convert a value into the appropriate type. For example

```
x:= INT 'a'
```

converts the value of the byte 'a' (ASCII) into a value of integer type. In addition, the operators ROUND and TRUNC specify rounding and truncation operations when converting between real and integer representations.

OCCAM supports the declaration of arrays of channels and variables. In contrast to OCCAM 1, which only allowed one-dimensional arrays, OCCAM 2 allows arrays with any number of dimensions to be declared. The size of an array must be fixed when it is declared. The following are examples of array declarations. Note that the dimensions are given first.

[20] INT fred:	Vector of integers
[100] CHAN OF INT switch:	Vector of channels
[8] [8] BYTE chessboard:	2D array of bytes

To provide low-level access to the machine, variables of integer type may be treated as machine words, allowing the use of bit-wise operators and shifts. OCCAM does not currently provide support for user-defined types or for record structures.

Abbreviations

A new feature of OCCAM 2 is that of the abbreviation. An abbreviation is a form of declaration, but instead of introducing a new name and a new object with that name, it introduces a new name for an existing object or for the value of an expression. This is particularly useful in selecting an element of an array in order to carry out some operations on it. For example

```
curr.node IS nodes[x.index] [y.index]:
```

This abbreviation allows the program to use the name 'curr.node' instead of 'nodes[x.index] [y.index]', within the scope of the abbreviation.

An abbreviation can also introduce a new name for a portion of an existing array. For example, an existing vector can be split up into two smaller arrays by the following abbreviations (the operator SIZE, when applied to an array, produces its number of elements):

```
[100] INT vector:
VAL half.size IS (SIZE vector)/2:
bottom.half IS [vector FROM 0 FOR half.size]:
top.half IS [vector FROM half.size FOR half.size]:
```

In many programming languages the introduction of a procedure call will often result in a new name (that of the formal parameter) being introduced for an existing object (the actual parameter). The abbreviation mechanism of OCCAM is a generalization of this to allow the same thing to be done independently of the procedure mechanism. The renaming of parameters in an OCCAM procedure call can be defined in terms of abbreviations.

The introduction of new names for existing objects is a common cause of programming errors, since a section of program may appear to be operating on two totally different objects, but in fact is operating on only one. This is called 'aliasing'. Due to the likelihood of programming errors as a result of aliasing, OCCAM does not allow it to occur. When an object is given a new name as a result of an abbreviation, the old name for the object may not be used within the scope of the abbreviation.

Channel protocols

When a communication occurs on a channel, it is important that the sending process and the receiving process agree on the type and size of the data being communicated. To achieve this, channels must be declared to be of a particular type, which is known as the 'protocol' of the channel. For example, declaring a channel as CHAN OF INT allows it to carry integer values. This is the simplest form of protocol definition. However, it is equally important to be able to declare a channel which carries a mixture of types, and so more structured protocol definitions are provided.

A sequential protocol specifies that a certain sequence of values, possibly of mixed types, may be sent over a channel. The output statement for a channel with a sequential protocol consists of a list of values of the appropriate types, and the input statement is a corresponding list of variable names. For example

```
PROTOCOL BYTE.INT.PAIR IS BYTE; INT:
CHAN OF BYTE.INT.PAIR c:
BYTE b:
INT i:
PAR
  c ! 'a'; 23
  c ? b; i
```

The sequential protocol, as a method for structuring communications, at first sight appears to be similar to the use of record types as a method for structuring data. This naturally leads to the question: why not just introduce record types to the language, and then allow declaration of a channel of record type? However, there is a subtle difference between records and protocols. A record is normally implemented using a contiguous block of store, and the communication of a record could be done using a single block transfer. The components of a protocol communication may be scattered in store, both at the output and at the input, and the most appropriate implementation is a sequence of communications, one for each element in the protocol.

Sequential protocols are useful when there is a particular format of message which is always passed on a

channel. Sometimes, however, it is useful to be able to communicate one of a number of different message formats. For this, OCCAM provides the variant protocol and a new kind of input called the 'CASE input'. An example of a variant protocol is

```
PROTOCOL DATA.VARIANT
  CASE
    char; BYTE
    number; INT
    string; [256] BYTE
    nothing
:
```

The protocol defines a number of possible message formats, each of which starts with a tag. The tags in the example are 'char', 'number', 'string' and 'nothing'. In the definition each tag is followed by the types of the following elements in the message. A CASE input for a channel with this protocol might appear as

```
CHAN OF DATA.VARIANT data.chan:
BYTE ch:
INT i:
[256] BYTE str:
SEQ
  ... do something
  data.chan ? CASE
    char; ch
      ... action for char
    number; i
      ... action for number
    string; str
      ... action for string
    nothing
      ... action for nothing
```

Each branch of the input consists of a list of variable names to hold the variables in the message following the tag, and an action to be executed if this element is chosen. The CASE input will input the tag and then choose the appropriate branch to be executed.

Control flow

The following control flow constructs are supplied in OCCAM. They are all single-entry single-exit, and there is no 'goto' statement.

IF	conditional
CASE	selection
Replicated SEQ	counted repetition
WHILE	tested repetition

The IF construct in OCCAM allows a program to perform one of a number of actions, depending on a sequence of tests. The components of the IF are evaluated one after another, and the first tested expression (called a 'guard') to be found TRUE causes that action to be executed. The IF construct in OCCAM differs from that in some other languages in its behaviour. If none of the guards are found to be TRUE, the IF construct does not terminate (i.e. it just stops dead). This forces the programmer to ensure that all possibilities are covered in the branches of the IF.

The CASE construct allows the choice of one of a number of actions, depending on the value of a selecting expression. A component of a CASE construct preceded by ELSE is selected if none of the optional values matches the selector. The CASE construct has been introduced in OCCAM 2.

Two forms of repetition are provided in OCCAM: counted repetition causes an action to be repeated a specified number of times; tested repetition causes an action to be repeated until a tested condition becomes false. A replicated SEQ causes a sequential process to be executed repetitively a number of times, and hence is similar to a conventional FOR loop. A WHILE construct executes repeatedly as long as the value of the associated expression is TRUE.

The control structures provided in OCCAM are intended to be minimal but sufficient. For example, the WHILE construct is the only form of tested repetition provided, whereas other languages often provide different variants of tested repetition, allowing exits at different parts of the block being repeatedly executed.

Realtime programming

There are two aspects of OCCAM related to realtime programming. The first is that a process can obtain the value of a realtime clock in order to regulate its behaviour in time, and the second is that a process can be prioritized to improve its performance in meeting realtime constraints.

Access to a realtime clock is achieved in an OCCAM program by the use of TIMERs. A process may ask for the time from a TIMER, and may also ask to be suspended for a fixed period of time. For example, the following procedure waits for a delay specified by the parameter delay.

```
PROC wait(VAL INT delay)
  INT now:
  TIMER clock:
  SEQ
    clock ? now
    clock ? AFTER now PLUS delay
:
```

A process structure can be prioritized using PRI PAR. This ensures that one process takes priority over another if both of them are ready to proceed. This is useful in ensuring that priority is given to handling external events over processing tasks which are not so time critical. For example

```
PRI PAR
  event.handler(data.in, buffered.data)
  data.processor(buffered.data, data.out)
```

Here the process 'event.handler' is given priority over 'data.processor' by placing it earlier in the list, to allow the event handler to respond to data coming in, and buffer it as required.

An ALT may also be prioritized. Using PRI ALT instead of ALT indicates that the guards earlier in the list should be given preference over those later in the list.

Multiprocessor configuration

An important objective in the design of OCCAM was to use the same concurrent programming techniques both for a single computer and for a network of computers. This means that the decision about how to distribute the system over multiple processors may be made fairly late in the development cycle. OCCAM programs can be mapped onto a network of processors by means of a PLACED PAR

statement which assigns processes to processors, and assigns channels to communication links.

ASPECTS OF SOFTWARE ENGINEERING

To examine the relationship between a programming language and software engineering practice, we need to see how features in the language affect the various activities in the design and construction of software systems. These activities are normally categorized as

- requirements analysis
- system specification
- design of software structure
- coding
- testing and debugging

The programming language used tends to influence the last three activities more than the first two, so we will concentrate on these.

In contrast to a number of other modern programming languages, OCCAM does not include any features specifically designed for modularizing large programs. Instead, OCCAM programmers divide up a system into processes, communicating over channels. This approach leads to a style of program development significantly different from that associated with conventional sequential languages, and it is the intention of this section to convey something of that style.

Designing process structures

An OCCAM program is constructed as a set of concurrent processes. Each process has a set of input and output channels, and for each channel there is a defined protocol specifying the set of messages allowed on that channel. The OCCAM programmer constructs such a process using OCCAM constructs such as SEQ, PAR and ALT. There is no special syntax to show the structure of a process (as there is for ADA tasks, for example).

During the design stage, an OCCAM programmer identifies the process structure appropriate to the application being considered and specifies the protocols on the channels in the system. In addition to this top-down decomposition it is also appropriate in the design stage to identify common functionality between different parts of the system, and to design libraries of procedures and functions which are to be shared across the system. Each process in such a design repeatedly accepts messages from its environment and performs some action as a result of each message. So each process normally has the form

```
... Procedure and variable definitions
SEQ
  going:= TRUE
  WHILE going
      SEQ
        ... input a message
        ... perform some action
```

A process might have only a single input channel on which to receive messages; alternatively a process might respond to messages from a number of client processes and share its time between them. In this case the main loop will look something like

```
WHILE going
    ALT i = 0 FOR no.of.clients
      client[i] ? message
        ... perform action
```

This shows an example of the replicated ALT, corresponding to the replicated PAR and SEQ already seen; it indicates that the ALT is to choose from one of an array of channels connecting this process to its clients. A process which has this structure often manages some data structure or communications path within the system, and is usually called a resource manager.

Using processes and channels as a method of structuring systems is different in some respects from the use of modules or tasks with entry names (as in ADA, for example). The interfaces between processes are relatively simple and processes are guaranteed not to interfere with one another. Internal representations of data may be hidden, since data is always copied between processes. The dependencies between different parts of the system are always visible in the communication channels, and may not be compromised without introducing more channels into the system.

Protocols

The set of messages which a process is willing to accept over each of its input channels is defined by the protocol of that channel. As an example, a simple data structure manager storing blocks of data in an indexed array on behalf of a number of clients might look like

```
PROTOCOL TO.SERVER
  CASE
    add; INT; [256]BYTE
    retrieve; INT
    test; INT
:
PROTOCOL FROM.SERVER
  CASE
    data; [256]BYTE
    result; BOOL
:
...
WHILE going
  ALT i = 0 FOR no.of.clients
    from.client[i] ? CASE
      add; index; store[index]
        full[index]:= TRUE
      retrieve; index
        SEQ
          to.client[i] ! data; store[index]
          full[index]:= FALSE
      test; index
        to.client[i] ! result; full[index]
```

Here the tag in the variant protocol is used to indicate which of the actions is required. This is appropriate when, as here, each of the actions corresponds to a different message format. Alternatively, it may be appropriate to define a set of message formats, each containing data indicating which action is required.

The sequence of messages which passes over a channel during its existence is called a 'channel trace'. Specifying the traces of channels in the system is an important part of designing OCCAM programs. The trace of

a channel may be defined by a Backus-Naur form (BNF) definition. For example

```
request          = add      |
                   retrieve |
                   test

trace(TO.SERVER) = {request}
```

This trace shows simply that the channel may pass any sequence of *add*, *retrieve* or *test*. However, constraints on the ordering can also be specified, for example if the program is to be written so that a *test* message always precedes any *add* or *retrieve* message. As part of the design it may also be desirable to note the relationships between channels, for example to say that a *retrieve* is always followed by a message to the client which is a *data* block.

When designing OCCAM programs the definition of the principal channel protocols often plays the same role in program design as does the design of the principal data structures in the design of sequential programs. However, the methods for specifying protocols are still at an early stage of development; better methods and software tools are required to support them.

Design methods

What design methods are available to aid in designing process structures? A number of commercially supported software design methods are based on a model of computation similar to that of OCCAM: these include Mascot and Jackson System Development (JSD).

One of the most common program design techniques associated with such methods is data flow, in which a data flow diagram is constructed showing the data flowing in and out of the system, with various transformations being carried out on the data as it flows through the system. With OCCAM the process structure for a system can often follow directly from the data flow analysis: pipelines of processing elements are inserted to perform the transformations on the data flow, and resource managers are inserted to provide data storage within the system. The protocol on a channel in the system can be derived from the data required to flow across that interface. The result is a process decomposition in which different processes perform different parts of the overall algorithm, such as the parallel graphics pipeline shown above.

Data flow is not the only technique used in the construction of OCCAM programs, however. The appropriate technique will often depend on the characteristics of the application. Many applications may be decomposed in such a manner that the same processing operation is applied to different parts of a data structure. In such cases the process structure may be derived from data structure decomposition, where each parallel process performs the same operation on different parts of a data structure. This is particularly appropriate for simulations of physical systems where the locality imposed by the physical structure of the system can be reflected in the process structure. For example, a statistical mechanical simulation of a system of 'spins' in a 2D lattice has been carried out on an array of transputers programmed in OCCAM[4]. A data structure representing the lattice can be split up between a number of transputers, each dealing with a small portion of the problem. Each processor performs the same actions and interacts only with its neighbours.

A variant of data structure decomposition can be seen in the processor farm, where a controller process hands out portions of a data structure to be worked on by one of an array of similar processing elements. When a processing element has finished its work, it is handed another portion of the data to work on. This technique is particularly appropriate where a data structure decomposition results in differences in the amount of computation required on different parts of the system. Many examples of applications appropriate to a processor farm exist, for example the ray-tracing program described in Reference 5. Figure 4 shows a pictorial representation of these three decomposition methods.

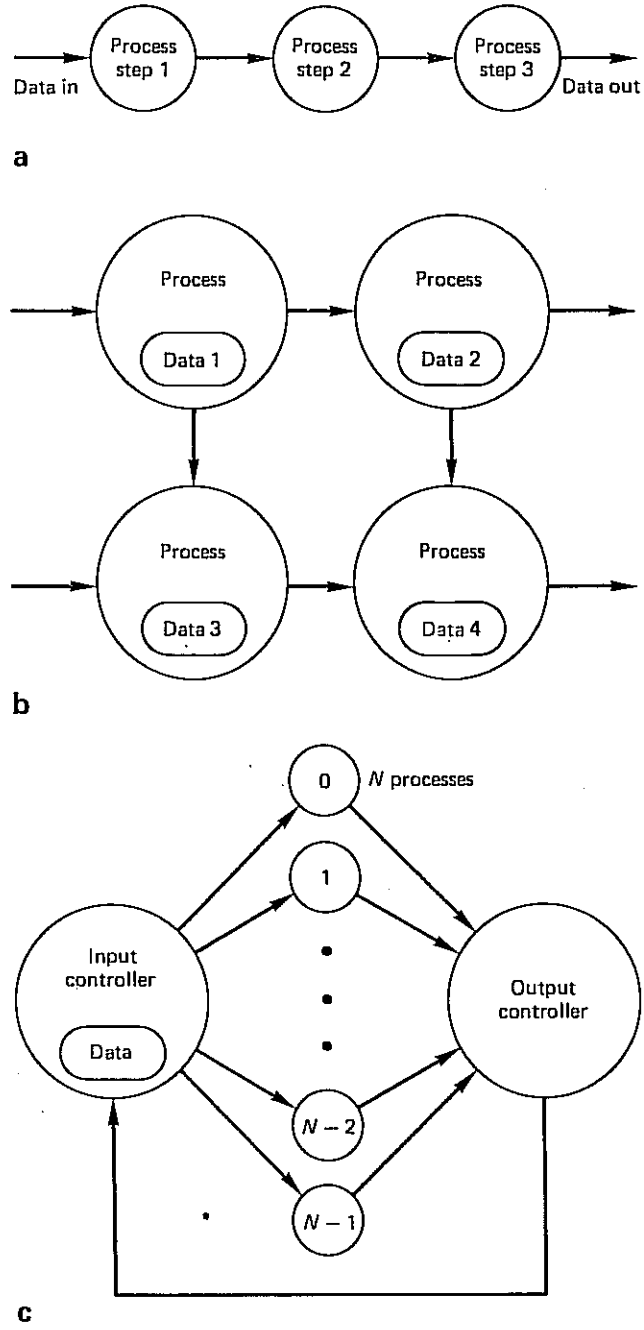

Figure 4. Decomposition methods: a, data flow decomposition, in which each process performs a different processing step on the data; b, data structure decomposition, in which each process performs a similar action on a different part of the data; c, processor farm, in which each action process performs a similar action on each data item it receives from the input controller

These approaches to designing process structures are not mutually exclusive but may be used together in the same system, in a hierarchical fashion. For example, a multitransputer circuit simulator being constructed at Inmos consists of a number of nodes. Each node contains a ring of transputers, each transputer performing a different part of the simulation algorithm. A large circuit can be simulated by placing different parts of the circuit on different nodes. This example uses data flow decomposition within a node, and data structure decomposition across nodes.

Coding

When implementing a section of code, a programmer has a number of concerns. The main ones are the correctness of the code and its efficiency. In addition the programmer should ensure that the program is clear and understandable. A number of choices in the design of OCCAM are intended to aid in the writing of correct and understandable programs.

The design of OCCAM has proceeded in conjunction with the definition of a formal semantics of the language. This allows the use of standard techniques in proving the correctness of programs. In addition, one result of this work has been the derivation of a set of algebraic 'laws' which relate different constructs in the language[6]. These laws, properly applied, allow a programmer to transform a program into a different (for example, more efficient) version of the same algorithm, without altering the correctness of the program.

At the present state of program verification and transformation, these techniques are not yet widely used in practice, except for applications where correctness is critical. The use of formal methods in defining the language does have other practical benefits; in particular the language can be defined without having to appeal to the programmer's intuition of how it might be implemented, and thus there is less room for ambiguity in the definition. Features of the language design which permit clear definition of the semantics of the language also aid in the understandability of programs. For example, the rules about aliasing and the lack of side effects in functions aid both of these goals.

When writing expressions in OCCAM, all type coercions must be shown explicitly, and operator precedence must be explicitly indicated by the use of brackets. This can be somewhat tedious when writing expressions, but it guards against the subtle programming errors that can result from misunderstanding of implicit type coercions and complicated operator precedence.

Other aspects of program syntax which aid in the understanding of OCCAM programs are the standardized rules for program indentation, and the fact that identifiers may be of any required length. OCCAM code is usually developed using a structured ('folding') editor which increases the ease with which program indentation may be handled, and provides a visible hierarchical structure which reflects that of the program being developed.

On the negative side, the lack of record structuring facilities and the absence of recursion in OCCAM often makes for less succinct and clear code than would be possible with these features.

The choice of features in OCCAM has been heavily influenced by the need for efficiency in implementation, so there are no features which are regularly avoided by programmers concerned with efficiency. However, the efficiency of a program is often highly dependent on choosing an appropriate process decomposition and ensuring appropriate buffering of data flowing through the system. So, in addition to the normal concerns about the efficiency of algorithms and of access to data structures, the OCCAM programmer has another dimension of efficiency to be concerned with, that of the trade-off between computation and communication.

Efficient access to data structures is sometimes supported within a programming language by allowing pointers (variables that contain the addresses of other variables). OCCAM does not support pointers. All data structures are arrays, and accesses to elements of these may be range checked at run time. This avoids the programming errors associated with pointers and improves the security of programs, but may leave some programmers concerned about the efficiency of data access. However, abbreviations can often be used for this purpose. A number of techniques for improving the efficiency of OCCAM programs, including the use of abbreviations, are discussed in Reference 7.

Testing and debugging

The testing and debugging of OCCAM programs is different in some respects from the same activities with sequential programs, and here we concentrate on the differences. The interfaces between processes are defined in terms of messages, so the external behaviour of processes may be tested relatively easily by creating test harnesses which simulate their environment. In some respects this is easier than testing part of a sequential program, where the interface may consist of a set of entry names plus some exported state variables.

In the debugging of concurrent programs, novel problems are those of livelock and deadlock. Livelock occurs when several processes are interacting repetitively with one another, but never communicating with the outside world. It corresponds to 'infinite looping' in a sequential program. In fact livelock is relatively uncommon in OCCAM programs.

Deadlock occurs when a collection of processes is unable to proceed, as each process is waiting for an action from some other process. In OCCAM programming, deadlock occurs often as a result of programming errors. For example, a simple programming error may cause an item to be missed in the set of messages passing over a channel. The process waiting for this message will fail to receive it and so will become blocked. Processes dependent on it will then also become blocked, and the deadlock spreads through the system in this way. Deadlock is best avoided by careful design of protocols and process structures; a number of design techniques for avoiding deadlock are being investigated.[8]

A practical problem in debugging OCCAM programs arises when it is necessary to obtain more visibility of the values of variables and data structures within a process. In a conventional sequential language the programmer would use print statements to output information to a screen or to a file. In OCCAM, however, there is no global environment; communication with a screen or a filing system is done over a channel, which is owned by only one of the processes in the system. So it is necessary to

build into an OCCAM program the communication paths to allow any process to output debugging messages. This should be done at the design stage.

If an error, such as an arithmetic overflow, occurs in a running process, one of a number of approaches may be taken (depending on a compile-time option).

- The process in error may stop, allowing other processes to continue.
- The whole system may halt.
- The error may have an arbitrary effect.

No 'exception handling' mechanism is provided in the language. However, when only the process in error stops, other concurrent processes in the system continue to operate and can be designed to notice the absence of data flow from the stopped process, and take action accordingly. When an OCCAM program is running on a transputer, a runtime error can be made to assert the Error pin on the transputer, allowing another transputer (or other part of the system) to detect and handle the error, perhaps by rebooting and reloading the transputer in error.

HIGH-LEVEL LANGUAGE FEATURES

This section summarizes the features of OCCAM in the light of the criteria for high-level languages identified by the first paper in this series[9].

Good program structure

OCCAM provides a minimal but sufficient set of control structures, all of which are single entry and single exit. The rules for declarations allow a hierarchical program structure, with declaration of procedures and functions at any level in the nesting. Programs are structured by a combination of process decomposition and procedural abstraction.

Support for compiler detection of errors

OCCAM 2 is a strongly typed language. There is no automatic type conversion, although a variety of type coercion operators are supplied to give flexibility to the programmer. Errors in misunderstanding of operator precedence are avoided by insisting on the use of brackets in expressions to indicate precedence. OCCAM compilers also carry out a number of other checks to enhance program security, in particular usage checking, which checks the use of channels and variables by parallel processes, and alias checking, which prevents a section of code from accessing the same data object by different names. The introduction of protocols in OCCAM 2 allows a channel to carry a mixture of different data types, while ensuring that the compiler can check that the processes on either end of the channel agree on the type and size of data being transmitted.

Support for readability

In OCCAM the names of variables, procedures and other declared objects may be of any length, allowing the programmer to choose meaningful names. The fixed format of OCCAM programs, in which the indentation of sections of program forms part of the syntax of the language and is checked by the compiler, forces a consistent style on OCCAM programmers and increases the ease with which one programmer may read another's work.

Support for abstraction

OCCAM provides less support for data abstraction than PASCAL and the other PASCAL-derived languages described in this series. There are no user-defined types and no support for record types. However, the process construct provides a powerful abstraction facility for the design of programs from a data flow analysis. Procedural abstraction is provided by means of procedure and function definitions. Recursive calling of procedures is not supported because of the implications this would have for the efficient implementation of processes.

Runtime errors

Careful attention has been paid to the handling of errors in OCCAM, and the behaviour of a program error is well defined. The design of the language reduces the possibility of particularly obscure runtime errors, in its compile-time checks and lack of pointers (allowing all accesses to data structures to be range checked).

Realtime applications

OCCAM is based on a simple and efficient model of concurrency and communication and provides an effective notation for the design and implementation of realtime systems. TIMER objects allow processes to control their behaviour with respect to an external clock. Configuration of programs to meet particular realtime constraints may be achieved by means of prioritized processes and multiprocessor configuration.

CONCLUSIONS

The computing community has had many years of experience in programming sequential systems, and many software engineering ideas have been developed and incorporated into language designs. In particular, ideas of program modularity and abstract data types have influenced a number of modern programming languages such as ADA. Experience of programming distributed multiprocessor systems is much more limited, however. In designing a language for such systems, one approach is to take all of the features of a language for sequential programming, such as PASCAL, and add to these the required mechanisms for concurrency and communication. Many concurrent language designs have taken this approach.

With OCCAM, the approach has been to provide the simple mechanisms for concurrency and communication first, and to see how this influences the style of program development. OCCAM 1 supported a process model of computation, but the type system was very simple, with the machine word as the only data type. OCCAM 2 is a strongly typed language, with a variety of basic types, but it does not yet incorporate the ideas of data structuring and abstract types which are now common in sequential languages.

The use of OCCAM in the design and development of software leads to significant differences in approach when compared with the use of sequential languages. A program is designed as a collection of communicating

processes, with data flowing across channels between the processes; the format of the data flow is defined by the channel protocols. Novel challenges in design are caused by the distributed nature of the system, such as the trade-offs between computation and communication, and the avoidance of deadlock.

In using OCCAM and similar languages to develop concurrent systems we will discover the appropriate abstraction mechanisms for designing processes, process structures and communication protocols. These abstraction mechanisms can then be incorporated in future 'higher-level' concurrent programming languages.

REFERENCES

1 **May, D and Taylor, R** 'OCCAM — an overview' *Microprocessors Microsyst.* Vol 8 No 2 (1984) pp 73–79

2 **May, D** *Occam 2 language definition* Inmos, Bristol, UK (1987)

3 **Hoare, C A R** *Communicating sequential processes* Prentice-Hall, Englewood Cliffs, NJ, USA (1985)

4 **Pritchard, D J *et al.*** 'Practical parallelism using transputer arrays' *PARLE Conf. Proc., Lecture Notes in Computer Science 258* Springer Verlag, Heidelberg, FRG (1987)

5 **Packer, J** 'Exploiting concurrency; a ray tracing example' *Inmos technical note 7* (1987)

6 **Roscoe, A W and Hoare, C A R** 'The laws of occam programming' *Technical monograph PRG-53* Programming Research Group, Oxford University, UK (1986)

7 **Atkin, P** 'Performance maximisation' *Inmos technical note 17* (1987)

8 **Roscoe, A W and Dathi, N** 'The pursuit of deadlock freedom' *Technical monograph PRG-57* Programming Research Group, Oxford University, UK (1986)

9 **Davies, A C** 'Features of high-level languages for microprocessors' *Microprocessors Microsyst.* Vol 11 No 2 (1987) pp 77–87

Russell Wayman leads the Development Systems Group at Inmos Ltd. He joined Inmos in 1982; since then he has worked on the company's Occam Programming System and Transputer Development System products. His interests include programming environments, concurrent programming languages and software design techniques. He received a degree in engineering and an MSc in computer science from Trinity College, Dublin, Ireland.