From Object Oriented Analysis to Implementation using Occam

Richard Beton
Roke Manor Research Limited, Romsey, SO51 0ZN, UK

Abstract. Shlaer-Mellor Object Oriented Analysis is discussed. The features necessary to implement an OOA model are compared with those offered by occam. It is found that many features normally requiring a complex software architecture are provided by the language. Finally, a tool is described which assists the job of converting an OOA model, captured using Team*work*, into occam source code.

1. Overview

Object Oriented Analysis (OOA) has become widely popular with implementers of information-based systems due to its methodology for capturing the system by first analysing the information it holds. Typically, when OOA is complete, a recursive design approach is taken. By adopting certain implementation paradigms, the solution to the system can be expressed in a standard sequential language - C++, Ada, etc. Unfortunately, the drawback of this approach is in mapping from the analysis phase to the implementation phase, with the solutions possibly being awkward and inefficient.

An alternative is readily available. The OOA approach can be summarised in three stages: capturing the information model, analysing the state models, and deriving the process models. This involves the creation, amongst other things, of an Object Communication Model (OCM), which is an enhanced form of the traditional Data Flow Diagram. The OCM is inherently occam[1]-like (even down to the name!) in that it consists of objects, containing independent state, which communicate by passing messages called events. This paper outlines a methodology for mapping OOA models to occam. Using an OCM captured via Team*work*[2], a Perl script called *yacht* generates the parallel occam code automatically, to which the definitions of each occam process are added by hand. *yacht* is published along with this paper.

2. Summary of the Shlaer-Mellor OOA Approach

Object Oriented Analysis has become widely popular with implementers of information-based systems due to its methodology for capturing the system by first analysing the information it holds. This paper considers the Shlaer-Mellor methodology [1,2] in particular. This can be summarised in three steps:

---

[1] occam is a registered trademark of SGS-Thompson Microelectronics Limited.

[2] Team*work* is a registered trademark of Cadre Technologies Inc.

- **Information models**. In this step, the focus is on abstracting the conceptual entities in the problem in terms of objects and attributes. The associations that pertain between the entities are formalised in relationships that are based on the policies, rules and physical laws that prevail in the real world.

- **State models**. The second step of the method is concerned with the behaviour of objects and relationships over time. In OOA, each object and relationship has a lifecycle, i.e. an orderly pattern of dynamic behaviour. State models are used to formalise the lifecycles of both objects and relationships. The state models, which are expressed in state transition diagrams and tables, communicate with one another by means of events, and are organised in layers to make the system of communication orderly and understandable.

- **Process models**. All of the processing required by the problem is contained in the actions of the state models. In this, the third step of the method, the actions are dissected into fundamental and reusable processes, and are depicted by the Action Data Flow Diagram.

Typically, when OO Analysis is complete, a recursive design approach is taken. Commonly, this relies upon the existence of a *software architecture* - a set of architectural classes providing support for state machines, timers, communication between objects, etc. By building on the software architecture, the solution to the system can be expressed in a standard sequential language - C++, Ada, etc. This is completely flexible in that it allows a wide range of target computation paradigms to be addressed. The drawback is that clumsy and inefficient solutions may result, and this is largely the cost of building upon a complex software architecture.

3. The occam Model

An occam [3] program consists of processes which execute concurrently and communicate by passing messages. In one sense, this makes occam one of the most pure object-oriented languages available. Each 'object' (i.e. process) contains its own private data and public channel interface - this strong encapsulation of data is very much an OO feature. However, occam differs from what many would consider to be object-oriented programming in four ways:

- occam processes can be considered to be executing truly concurrently (whether this is in fact implemented using a timesharing processor is immaterial), whereas the objects in, say, C++ are just bundles of procedures, functions, and data, with (normally) only one thread of execution. In occam, the clearly-defined semantic of the execution model is important; in OO, classes are used to encapsulate *at the source code level,* but the execution semantic usually follows the traditional unencapsulated von Neumann sequential approach.

- Because occam processes execute concurrently, occam embodies strict rules about the sharing of data between processes: this is necessary in order to maintain data integrity[1]. This differs from the use of OO object accessor methods to change data in other objects at arbitrary times: such an approach leaves data integrity entirely up to the designer and

[1] Note that the shared data usage rules in occam do allow the direct passing of data in memory between short-lived parallel processes, in a Concurrent Read / Exclusive Write (CREW) manner. The processes described in this paper which represent objects are typically long-lived state machines, however.

coder[1]. In both cases, strong encapsulation may have a cost in execution time taken to access non-local data.

- occam processes pass messages between themselves along interconnecting channels. Few other languages, with the notable exception of SmallTalk, actually use message passing between objects. Rather, object accessor methods are normally implemented as procedure or function calls.

- occam does not provide classes and therefore does not provide inheritance between classes. Different provision is made for source code re-use. In occam2.1 [3], this is in the form of support for libraries which may contain arbitrary collections of code. In occam3 [4], modules provide a stronger expressive power for encapsulating processes and their interfaces. This is closely related to the client-server relationships that can be expressed in occam3. Additionally, the concept of libraries is supplied as a language feature, akin to Ada packages.

It is worth considering that the conventional OO approach is an evolutionary development from earlier methods. For example, the name 'C with Classes', as C++ was once known, highlights how classes were added to C initially to solve the problem of partitioning the global name space. occam's more revolutionary thinking has both opened new opportunities for expressing parallelism and therefore for describing concurrent systems, and at the same time hampered its wider acceptance because it requires people to think differently.

4. Comparison of Shlaer-Mellor and occam Paradigms

Concurrency and communication are prime concepts of both OOA and occam paradigms. OOA is in its very nature an analysis methodology, whereas occam is an implementation technology which can also be used for design, because of its rich set of transformation rules. The opportunity therefore exists for OOA to address the task of understanding the requirements of a large system, complemented by the use of occam to capture a design to meet that requirement and to refine it to the point at which it can be executed. Other languages, e.g. Ada, also offer scope for this approach, but apparently only occam supports the approach at an acceptable execution cost, due to its lightweight process semantic giving fast context switch speeds[2].

The combination of OOA and occam requires some care, however. It is not possible to take an OOA description, run some magical tool, and hope that a complete occam implementation of the system will result[3]. Besides the similarities between an OOA description and the occam model, there are some important differences:

- The Object Communication Model (OCM) depicts the interaction of object state machines by the passing of *asynchronous* events between them. To implement object state machines as occam processes, the channels between the processes would have to pass through queues on the input to each process. The OCM makes no provision for avoiding the otherwise inevitable deadlock. The addition of queues causes disadvantages which are discussed later.

---

[1] who may then ignore the issue completely, as we shall see later.

[2] the recent availability of occam compilers for Unix has permitted the author to confirm this comparing occam and Ada context switching and communications speeds, but this is beyond the scope of this paper.

[3] or implementation in any other language, for that matter.

- The Object Access Model (OAM) depicts the synchronous interaction between objects by means of reading and writing each others' internal data. This is not directly possible in occam and causes some difficulty, discussed further below.

5. Mapping OOA to occam

Information and state modelling is the domain of OOA. Process modelling overlaps with the specification and refinement of an occam program using occam as its own design tool. The Object Access Model and Action Data Flow Diagram recommended by Shlaer & Mellor are just two of many possible techniques for this sort of refinement: in this case, they should be seen as complementing other techniques.

5.1. Communication and Timers

occam does not need inter-process communication support to be provided by a software architecture. Neither does it need support for timers. Both are provided by the language. In this respect, the mapping of an OOA model to occam is very straightforward. occam channels always synchronise at the moment of data exchange. OOA events are asynchronous, in the sense that the sending object and receiving object do not synchronise to pass the event. This can be implemented in occam by decoupling event channels using buffers, although introducing buffering brings problems associated with deciding a suitable buffer size and with proving correct behaviour, since the program is no longer a simple CSP construction[1].

The set of events passing between a given source object and target object may be described in occam using a corresponding set of tags. This may be represented either using a variant channel protocol, or by using a set of integer constants. In either case, the main body of each OOA object is a finite state machine. The state machines derived from the OOA model can be implemented in occam using `CASE` statements on the events arriving at the input(s) to an object. Input alternation (the `ALT` construct) allows selection of an event from a variety of potential sources. More details follow later.

If the system is implemented on a multi-processor platform and is required to exhibit fault-tolerance in the event of processors failing, additional protection must be added to an occam program for the channels that pass between processors. Much has been written on this subject elsewhere (see e.g. the Fault Tolerant General Communications Harness described in [5,6]).

5.2. Sharing Data Between Objects

The actions associated with the State Model for each object reveal the propagation of events between objects and thus leads to the construction of the OCM. They may also reveal interaction between objects and attributes belonging to other objects. This interaction - the communication between objects by means of reading and writing each others' internal data - is depicted in the OAM. Since OOA objects are considered to be essentially concurrent, in respect of this interaction, the Shlaer-Mellor approach ceases to be rigorous about data integrity. However, this is not normally a problem because a sequential implementation language is used and the lack of ordering of event processing is overlooked.

occam has rigorous parallel data usage rules which raise the question of how inter-object data sharing might be implemented in occam. There are several possible approaches. Firstly,

[1] However, an in-depth discussion regarding the advantages and disadvantages of the Shlaer-Mellor methodology is beyond the scope of this paper.

one might seek to minimise, or even eliminate, such interactions by changing the model. Those that remain could be handled simply by sharing memory, therefore breaking occam's usage rules (current occam compilers do actually allow their usage checkers to be turned off). This is an ad-hoc approach which may work in some circumstances but is hardly to be encouraged. Furthermore, it should only be applied to scalar variables, because record variables or arrays may or may not be updated atomically, depending on the compiler and CPU implementation, leading to the worrying possibility of shared variables being accessed whilst in an inconsistent state.

There is an alternative and preferable approach. occam OOA objects may provide access to their internal state via communication channels carrying accessor protocols. Whereas in occam2 this requires a pair of channels and a particular coding style, in occam3 the provision of shared channels enables any process to provide accessors to the internal data using the `GRANT` keyword. These accessors must be `ALT`ernated with the input of the main asynchronous OOA events in each object's state machine.

The immediate problem arising from introducing synchronous inter-object communication is that of the danger of deadlock - the OAM may contain cyclic (i.e. mutually dependent) references which are problematic. According to Welch et al [8], removal of these cycles leads to guaranteed deadlock-freedom, as long as the accessor protocols match the *client-server* paradigm describe in [8] and thereby meet the restrictions on the body of a `CLAIM` described in [4] on page 60[1] (both of which happen for free with occam3 shared channels). Two approaches may be used to remove cycles. In some cases, it may be apparent that small changes to the model can be applied, removing certain cycles from the OAM. Otherwise, cycles can be broken by extracting shared data from selected objects into new passive objects, with which all the original sharing objects communicate. Since these passive objects do not have any dependencies of their own, they operate simply as servers in the sense used by Welch et al, thus guaranteeing deadlock freedom.

Although accessor interfaces implemented using channels eliminate any race problems that might arise from unsynchronised sharing of memory, it is not clear that they introduce any greater clarity, since the relative order of event execution and of shared data updating is undefined by the Shlaer-Mellor approach, highlighting an inherent weakness. The justification for this seems to be: if a significant dependency exists between objects, then it should be expressed in the model using event interactions; if a trivial dependency exists, then it does not matter if occasionally an out-of-date value is accessed. It is up to the modeller to be aware of this and protect significant dependencies appropriately. Otherwise, there is plenty of scope for making an unpredictable system.

5.3. Implementing a State Machine

The behaviour of an OOA object is decribed by a state machine model (although the state machines in some objects may be trivially simple), which receives events thereby undergoing state transitions and causing new events to other objects. To implement this as an occam process requires a process which continuously waits for input; each time input is received, the corresponding state action is executed. The state action may cause the generation of further events to other objects.

An OOA object consists of a set of instances of the state machine being modelled. This set could be viewed as a 1-dimensional array or table of instances. Considering the general case of an OOA object which has many instances, two alternative approaches can be considered for implementing the instance table.

---

[1] the body of a `CLAIM` may not contain another `CLAIM` nor communication along any other channels.

Firstly and most obviously, each instance could be a separate occam process. In this case, the object as a whole is implemented using replication of the instance process:

```
...  PROC object.A.instance
...  etc
PAR
  ...  some processes
  PAR i = 0 FOR number.of.instances.of.A
    object.A.instance (...communication channels...)
  ...  some other processes
```

Each object then holds one (scalar) record describing the object state. This approach implements fully the parallelism in the model. However, it may not prove to be as simple to implement as it first seems, because, frequently, communication is required between the different instances of an object. Occasionally, communication is required to the object itself rather than instances of the object. Neither of these difficulties is insurmountable, but usually the effort is not necessary.

Secondly, an alternative solution is that a multi-instance object could be implemented as a single (unreplicated) occam process. This less-parallel solution is probably to be desired wherever high parallelism is not an issue. The instance state data is held as an array; each element of the state array is a record representing the state of a particular instance.

For example:

```
PROC object.A (CHAN OF EVENT in,
               CHAN OF EVENT out.to.object.B,
               CHAN OF EVENT out.to.object.C)
  ...  PROC event.1.action
  ...  PROC event.2.action
  ...  declaration of state variables
  ...  declaration of event variables
  SEQ
    ...  initialisation of object state
    WHILE TRUE
      in ? CASE
        event.type.1; instance; event.1.data
          event.1.action (state[instance], event.1.data)
        event.type.2; instance; event.2.data
          event.2.action (state[instance], event.2.data)
        -- etc
:
```

In this simple example, there is only one input channel - in general there would be more, using a fair `ALT` to select from them. Note that the use of input queues makes fair alternation important: occam makes no guarantees about the fairness of its `ALT` implementation. Well-known techniques exist for writing fair `ALTs` - see [7]. occam3 shared channels would be preferable - they have naturally fair behaviour and execute with less overhead than the equivalent occam2 fair `ALT`. The continuous nature of the state machine is inherent in its use of `WHILE TRUE` - this process shall never terminate. In the example, two possible event types may be received. Each has an action procedure which fulfills the state transition model. An instance number is used to index the instance state, which is passed to these action procedures so that it can be altered. Event data is also passed to the action procedures. Those channels used in the generation of output events could be passed in, but this is not generally necessary because they are already in scope within the action procedures.

In the example, the three channels all share a common protocol EVENT. Alternatively, input and output event protocols may be unique to each pair of interacting objects. Unfortunately, in occam this can lead to large program size and poor sharing of common functions, so a superset event protocol is commonly adopted. An even simpler protocol involves using an integer tag rather than a variant protocol (it is easier to write buffers in this case). Selecting one of these protocol paradigms has to be a design decision which trades-off type-safeness against code complexity for the particular system being implemented.

The previous example is reproduced below in occam3 using shared channels and the SERVER construct.

```
PROC object.A (SHARED EVENT in,
               SHARED EVENT out.to.object.B,
               SHARED EVENT out.to.object.C)
  ...  PROC event.1.action
  ...  PROC event.2.action
  ...  declaration of state variables

  INITIAL
    ...  initialisation of object state
  :

  SERVER
    ACCEPT
      GRANT in
        ...  declaration of event variables
        in[request] ? CASE
          event.type.1; instance; event.1.data
            event.1.action (state[instance], event.1.data)
          event.type.2; instance; event.2.data
            event.2.action (state[instance], event.2.data)
          -- etc
  :
:
```

Within the event actions, CLAIMs may be made on the services of the output channels.

6. Using Team*work* for OOA

The Team*work* tool from Cadre has optional support for OOA. Notably, this takes the form of support for using the data flow diagram (DFD) editor to produce OCM diagrams. Little more needs to be said here concerning this, besides remarking that the opportunity exists for a semi-automatic tool to access the Team*work* model database in order to produce occam from an OCM. The *yacht* tool was written for this purpose.

6.1. yacht - yet another code harness tool

yacht is a Perl [9,10] script to convert a Team*work* OCM to partially-complete occam source code. It accesses the model database using the *twk_get* Team*work* utility. Each bubble in the OCM corresponds to an occam process or replicated array of processes. Each data flow corresponds to an occam channel or array of channels for synchronous communication. Each control flow corresponds to an occam channel or array of channels for asynchronous communication. Each asynchronous channel is converted automatically into a pair of synchronous channels passing through a buffer process. Channels which flow elsewhere are required to have terminators attached.

yacht generates the parallel harness code, consisting of:

- Optionally, a stub process declaration for each of the OCM bubbles (i.e. a `PROC` for which the body of the `PROC` is just `SKIP`). Similarly, buffer processes are generated as required: for intrinsic channel protocols (e.g. `CHAN OF INT`, `CHAN OF BYTE`, etc) these are fully expressed, but for user-defined protocols, stubs only are generated.

- A list of channel declarations according to the flows used in the OCM. Channels connected between a bubble and a terminator are assumed to be declared elsewhere. Each asynchronous channel is expanded to two occam channels, between which a buffer process is placed. The exception is asynchronous channels with a terminator at the output end - buffering is already provided on asynchronous channel inputs, and need not be provided twice.

- A `PAR` block instantiating each of the OCM bubbles. Any process may be marked as having high priority. Automatically-inserted buffers are put at high priority. A replicator may be supplied which is used to specify an array of processes.

The behaviour of processes must be defined manually. So too must be the behaviour of buffers which decouple user-defined protocols.

6.2. yacht example 1

As an example, figure 1 shows three process-types. The process-types may represent OOA objects, but this need not necessarily be the case. The `worker` process is replicated, indicated by the replication syntax within square brackets. The channels are of the protocol specified in curly braces, without which '`CHAN OF ANY`' is assumed. They are arrays of channels, as indicated by the empty square brackets.

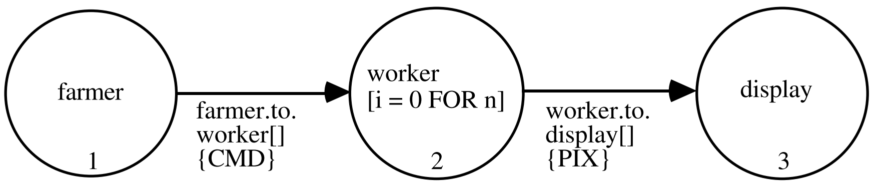

Figure 1: Simple Process Farm OCM

Note that the solid arrows denote data flows - these are interpreted as synchronous (i.e. normal) occam channels; buffered channels are discussed later. If the Team*work* model RDB_EXAMPLES contained this OCM in DFD 0, the following output might be generated:

```
-- Simple farm - OCM 0 from RDB_EXAMPLES, last written 21.11.95
-- Generated by 'yacht' for rdb on Tue Nov 21 12:48:15 GMT 1995

--{{{  External Module Stubs

PROC farmer ([]CHAN OF CMD farmer.to.worker)
  SKIP
:
```

```
PROC worker (VAL INT i,
             CHAN OF CMD farmer.to.worker,
             CHAN OF PIX worker.to.display)
  SKIP
:

PROC display ([]CHAN OF PIX worker.to.display)
  SKIP
:

--}}}  End of External Module Stubs

--{{{  Channel Declarations

[n]CHAN OF CMD farmer.to.worker :
[n]CHAN OF PIX worker.to.display :

--}}}  End of Channel Declarations

--{{{  Parallel Process Structure

PAR
  farmer (farmer.to.worker)
  PAR i = 0 FOR n
    worker (i, farmer.to.worker[i], worker.to.display[i])
  display (worker.to.display)

--}}}  End of Parallel Process Structure
```

The output from *yacht* can be redirected to a file, whence it can be `#INCLUDE`d into enclosing source code that provides hand written declarations.

Had the process-types in this example represented OOA objects in an OCM, the question of whether to show those objects as replicated processes is an interesting one - see the previous discussion in section 5.3.

6.3. yacht example 2

Another simple example illustrates how replicated processes may have 'daisy-chain' channels.

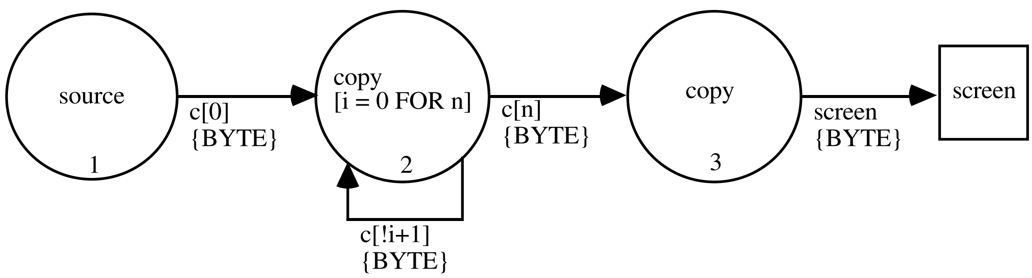

Figure 2: OCM including a daisy-chain

Of note here, the channel which has both ends connected to the replicated process forms a daisy-chain. Within the square brackets is an output operator, `!`, and an index expression, `i+1`, which is is used instead of the default array index expression in the outputting end. A daisy-chain could be formed in the opposite direction using `[?i+1]` to replace the default array index expression in the inputting end. Similarly, some of the channels specify indices (`[0]`

and [n]) which imply that these are scalar elements of a channel array using the index provided. The optional process stubs have been omitted from this listing:

```
-- Chain of processes - OCM 10 from RDB_EXAMPLES, last written 21.11.95
-- Generated by 'yacht' for rdb on Tue Nov 21 12:55:37 GMT 1995

--{{{  Channel Declarations

[n + 1]CHAN OF BYTE c :
-- CHAN OF BYTE screen : -- declared elsewhere

--}}}  End of Channel Declarations

--{{{  Parallel Process Structure

PAR
  source (c[0])
  PAR i = 0 FOR n
    copy (i, c[i], c[i+1])
  copy (c[n], screen)

--}}}  End of Parallel Process Structure
```

6.4. yacht example 3

This third example illustrates two OOA 'objects' connected by a pair of asynchronous channels denoted by Team*work* control flow arrows. The INT protocol is simpler than would normally be used, for the purpose of clarity.

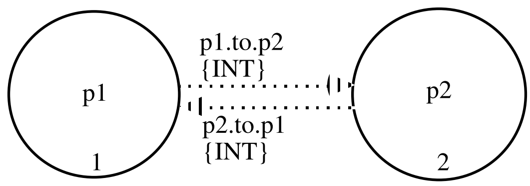

Figure 3: OCM for Two Connected OOA Objects

The following output might be generated; note that the arrays p1.to.p2.buffer and p2.to.p1.buffer must be declared earlier for this to compile - these arrays are []INT of a size determined by the programmer.

```
-- Simple farm - OCM 10 from RDB_EXAMPLES, last written 22.11.95
-- Generated by 'yacht' for rdb on Wed Nov 22 18:20:10 GMT 1995

--{{{  Buffer Declarations

--{{{  PROC buffer.INT.BLOCK

PROC buffer.INT.BLOCK (CHAN OF INT in, out, []INT array)

  ...  store.INT.BLOCK
  ...  requester
```

```
  CHAN OF INT thru:
  CHAN OF BOOL req:
  PAR
    store.INT.BLOCK (in, thru, req)
    requester (thru, out, req)
:

--}}}  End of PROC buffer.INT.BLOCK

--}}}  End of Buffer Declarations

--{{{  External Module Stubs

PROC p1 (CHAN OF INT p2.to.p1.bu,  -- in
         CHAN OF INT p1.to.p2)
  SKIP
:

PROC p2 (CHAN OF INT p1.to.p2.bu,  -- in
         CHAN OF INT p2.to.p1)
  SKIP
:

--}}}  End of External Module Stubs

--{{{  Channel Declarations

CHAN OF INT p1.to.p2 :
CHAN OF INT p1.to.p2.bu :
CHAN OF INT p2.to.p1 :
CHAN OF INT p2.to.p1.bu :

--}}}  End of Channel Declarations

--{{{  Parallel Process Structure

PRI PAR
  PAR
    buffer.INT.BLOCK (p1.to.p2, p1.to.p2.bu, p1.to.p2.buffer)
    buffer.INT.BLOCK (p2.to.p1, p2.to.p1.bu, p2.to.p1.buffer)

  PAR
    p1 (p1.to.p2, p2.to.p1.bu)
    p2 (p2.to.p1, p1.to.p2.bu)

--}}}  End of Parallel Process Structure
```

The buffer inserted here is named according to its channel protocol (`INT` in this case) and has a suffix denoting how the buffer responds when it overflows. There are three types of buffer supported - `BLOCK` for blocking buffers, `OWB` for over-writing buffers, and `STOP` for aborting buffers. Blocking buffers are the default; when full, a blocking buffer refuses to input any more data. Aside from the possibility of a deadlock, data integrity is guaranteed. An overwriting buffer will never deadlock, but overflow causes loss of data (the oldest is lost). A stopping buffer will halt the program if it overflows, allowing the use of debugging tools to diagnose the situation. The selection of buffer type is done on a per-channel basis by adding the buffer mode in the curly braces used to specify the channel protocol: for example a control flow labelled `a.to.b {INT, OWB}` specifies the insertion of an overwriting buffer for `INTs` between the channels `a.to.b` and `a.to.b.bu`.

The three examples illustrate just some of the features of *yacht*. Lack of space prohibits illustration of explicit high priority processes, but those familiar with occam will have no difficulty with this concept.

Future work may generalise *yacht* to handle multi-dimensional arrays of processes and channels. Parsing of the data dictionary would allow automatic generation of channel protocols, and also give more help with the state machine input case construction. Further, it is hoped to include the hierarchical navigation of the Team*work* DFD tree as a means of producing refined internal processes structure. The generation of occam 3 rather than occam 2 code, especially shared channels, may be added when occam 3 compilers become available.

7. Conclusion

The outline of a methodology for implementing an Object Oriented Analysis model in occam has been presented. In many ways, this is easier for occam than for purely sequential languages. The lack of inheritance in occam is no obstacle (OOA makes little use of inheritance anyway) - other satisfactory means for code reuse and software engineering exist. The resulting occam solutions exhibit parallelism closely related to that in the OOA model. This has several benefits: understandability and maintainability are better, parallel computing hardware can be used, and the lightweight software architecture in occam brings efficiency gains, albeit offset somewhat by the cost of process switching and message passing. Other opportunities to exploit parallelism, for example task farming, could be pursued separately as part of the design process if that was an important consideration.

Issues of efficiency have not been analysed in depth in this paper. The primary objective of using OOA is the rapid development of correct solutions to information processing applications, built from maintainable and re-usable components. These aims suit occam well. Inevitably, the abstraction provided by OOA has some execution cost. Furthermore, asynchronous communications between OOA objects require infinite buffering to ensure zero message loss - clearly impractical. Some non-trivial decision has to be made concerning buffer size to provide for a reasonable expectation of event density under busy conditions. Asynchronous events also make the formal proving of the system more difficult than would normally be the case for an occam system constructed without using OOA. Sharing of attribute data between objects has been shown to be non-rigorous - a feature of OOA which sits uneasily alongside occam's rigorous approach, although this does not prevent the mapping from OOA to occam.

A tool called *yacht* has been presented which assists the task of converting a Team*work* (tm) OOA object communication model directly into occam. *yacht* was created with a view to automating part of the implementation effort of an OOA system. It has wider benefit in that it can be used for any process description captured using Team*work* in this way, not just OOA systems.

Another important OOA tool, the proprietary IOOA tool from Kennedy & Carter, could have a similar software architecture mapping to occam. This would in principle be superior in the sense that the entire occam program could be generated automatically. It would also be superior to the existing C code generator in the sense that the parallelism of the OOA model is expressed in the implementation, allowing better use of multiprocessor computers, whereas the C code generator sequentialises the behaviour. The author is not aware of any current plans to add an occam code generator to IOOA.

8. *yacht* availability

yacht has been written by the author and is made available by kind permission of Roke Manor Research Limited from

`http://www.hensa.ac.uk/parallel/occam/utilities/yacht/`.

Thanks go to the Hensa Parallel Computing Archive for providing this WWW facility.

9. References

[1] S. Shlaer, S.J. Mellor, *Object Oriented Systems Analysis - Modeling the World in Data*, Yourdon Press 1992.

[2] S. Shlaer, S.J. Mellor, *Object Lifecycles - Modeling the World in States*, Yourdon Press 1992.

[3] Geoff Barrett, *occam 2.1 reference manual*, SGS-Thomson Microelectronics Ltd, Prentice-Hall (see `http://www.hensa.ac.uk/parallel/occam/documentation/oc21refman.ps.gz`)

[4] Geoff Barrett, *occam 3 reference manual (draft)*, SGS-Thomson Microelectronics Ltd (see `http://www.hensa.ac.uk/parallel/occam/documentation/manual3.ps.gz`)

[5] R.Beton, J.Kingdon, C.Upstill, *High Availability Transputing Systems*, Procs Transputing 91, April 1991

[6] R.Beton, D.Barrett, I.Relf, J.Bunn, *Fault Tolerance applied to a Transputer Vision System*, PACTA 1992

[7] Geraint Jones, *Carefully Scheduled Selection with ALT*, OUG Newsletter 10 (1989), pp 17-23

[8] P. Welch, G. Justo, C. Willcock, *High-Level Paradigms for Deadlock-Free High-Performance Systems*, Transputer Applications & Systems '93, IOS Press 1993

[9] Randal L. Schwartz, *Learning Perl*, O'Reilly & Associates, Inc. 1993.

[10] Larry Wall, Randal L. Schwartz, *Programming Perl*, O'Reilly & Associates, Inc. 1991.