TRANSFORMATION OF occam PROGRAMS

M. H. Goldsmith and A. W. Roscoe

University of Oxford, UK

INTRODUCTION

The value of formal techniques in conventional software engineering is starting to gain recognition even outside academic computer science departments. We maintain that their use is essential in taming the extra complexity that accompanies the extra power that concurrency brings: for while in a FORTRAN program, say, the contribution of the 'program counter' to the number of states that need to be considered is the sum of the contributions of its component modules, in a highly parallel program it is their product.

Various paradigms of parallel programming have been suggested, all with much the same expressive power. We choose to deal with one of the most tractable: synchronising communications. The mathematical process calculi CSP (Hoare (1), (2)) and CCS (Milner (3)) both allow independent concurrent processes to influence one another only by cooperating in *events* which require the simultaneous participation of those involved.

occam

The programming language occam (INMOS (4)) owes much to these theoretical predecessors. Any memory location shared between two parallel processes is read-only to both of them, and the synchronising events are provided by point-to-point communications. Programs are built up from assignments, input ('?') and output ('!') processes, by combining them in sequence or in parallel, and by choosing between them by conditional expressions or arbitration between alternative communications. Rather than begin/end brackets, block structure is expressed in the layout; grouped processes have the same fixed indentation relative to the surrounding code.

A more recent variant of the language, occam 2 (INMOS (5)) adds simple data-typing and communication channel classification. Both were conceived primarily as a very-high-level assembler for the transputer family of microprocessors, which make it easy both to combine many processors to cooperate on a single problem, and equally to support efficiently many processes time-sliced on an individual processor.

This paper describes our work on formal manipulation of occam, and gives an example of a practical application of program transformation techniques.

PROGRAM TRANSFORMATION

By program transformation we mean replacing (some part of) a program by another (in the same language) with the same 'meaning'. (What we mean by 'meaning' is the topic of the next section.) On the face of it this may not seem a very profitable thing to do, but there are several reasons why one might want to.

One is in order to take a description of a system by a piece of software, and to transform it so that the resulting equivalent form is built out of 'clichés' which are particularly easy to implement as hardware. In this way a transformation phase would form the 'front-end' of a silicon compiler. This is a technique which is under active investigation by at least one chip manufacturer, and a related application (in microcode development) has already been used in the design of a successful commercially produced processor (Shepherd (6)).

Another, perhaps more obvious, one is simply to improve the performance of a program; source code optimisation. An automated transformation might serve as a first pass in a conventional software compiler too. This is again a promising line of research, although it is somewhat hindered by the fact that introducing concurrency is a much harder problem than eliminating it.

In the immediate future it will probably still be necessary to rely on the skills of experienced programmers to produce efficient parallel solutions to problems. These solutions will not necessarily be easy to relate to their specification. But all is not lost: if a (possibly very inefficient) solution can also be produced which is easily proved to satisfy the specification, then we may exploit a very useful property of our semantics. Any loop-free program is equivalent to one in a canonical normal form—this happens to be a decision tree of alternate conditionals and arbitrations, with assignments at the leaves and carefully controlled variable usage, but the precise form is not important—and indeed the same is true of any program with bounds on the number of iterations that any loop may make. For programs with potentially infinite behaviours there is a similar but infinite normal form, which is in a rigorous sense the limit of finite approximations, which can be calculated. Thus the normal forms of the two solutions can be directly compared, and any discrepancies in the efficient implementation will show up.

Other less dramatic, but equally useful, transformations are possible: the use of local variables, for instance, can be manipulated, so as to minimise the store requirements of a program; this may be important in embedded systems, or where the speed-up from using on-chip memory is vital. At an even lower level, global renaming of a particular variable—not all variables which happen to have the same name—can be expressed as a transformation in this sense, and carried out by our transformation tool.

A further application area is explored below, that of fitting logical process structure onto physical processor topologies. Among the transformations this needs is one ensuring that arbitrations exhibit short-term fairness.

SEMANTIC DESCRIPTIONS

A semantic description of a programming language is some systematic way of giving a mathematical meaning to program text. There are at least three major styles of approach to this task, and a considerable body of research in each of them has been carried through from CSP and CCS to apply to occam.

Denotational Semantics

The most direct way to assign an abstract meaning to a program is by denotational semantics. This consists of a set of functions, one for each syntactic category (e.g., 'expression', 'process') of the language, mapping into a mathematical domain of 'meanings'; and so constructed that the meaning of a compound construction is entirely determined by the meaning, not the form, of its subcomponents.

It is important to realise that there is quite a wide degree of freedom in the choice of such functions, depending on the level of abstraction desired. At two (ridiculous) extremes there is on the one hand the functions which identify absolutely every object, and on the other those that distinguish every two objects which are not syntactically identical. In the middle ground there may be several contenders with equal claims to capture the essence of the language: for instance, one debatable point is whether *deadlock*, where no two processes have an action in common to perform and none is able to communicate with the outside world, should be identified with *divergence*, where either a network is able to carry on an infinite internal conversation, where the actions are hidden from the observer, or else some process is able to consume unlimited amounts of processor time without making any progress. In both cases, the immediate view from the outside world is the same: the network remains unwilling to communicate, forever. But in the one case the electricity bill will be considerably smaller than in the other!

The published denotational semantics of a large subset of occam (Roscoe (7)) owes much to the failures-divergence models of CSP (Brookes and Roscoe (8)), and does distinguish these two conditions. Two processes are identified if they are able to engage in the same sequences of communications with their environment, and if at each stage they are able to (non-deterministically) refuse the same sets of communications; and if moreover, whenever they terminate successfully, they have the same possible combinations of values stored in their free variables. This abstracts away such notions as efficiency, timing and amount of true concurrency.

Operational Semantics

The approach of operational semantics is to consider the space of possible states of an ideal machine implementing the language, and the transitions between them corresponding to atomic actions. The modern style is to present a natural deduction logic which again allows the behaviour of compound terms to be deduced from the behaviour of their components. Such a semantics of CCS is well established (Milner (9)) and one has been developed for CSP (Brookes *et al* (10)), which is proved to be congruent to the denotational semantics of (8).

The behaviour of occam processes has much in common with those of both CCS and CSP, with some extra complications. This semantics is the subject of current research (Barrett (11)), especially dealing with aspects of scheduling and arbitration when the program assigns priority to particular alternatives and pseudo-parallel components.

Algebraic Semantics

Both the previous approaches may be able to attach an intuitive, as well as a mathematical, meaning to a program, if the models are well chosen and easily understood. But one of the main reasons to consider the semantics of a language, and the most important for this work, is to decide when two programs mean 'the same'; and so one with more desirable properties can be substituted for another.

Repeated calculation of the meanings of programs for this purpose would soon become exhaustingly tedious. Algebraic semantics takes the notion of program equivalence as central, and directly encodes the relationship by means of *laws*: axiom schemata relating equivalent forms. Such an axiomatisation (Roscoe and Hoare (12)) completely generates the equivalence induced by (7).

Such equivalences, which amount to intersubstitution preserving some notion of gross behaviour, lie at the heart of program transformation. It is here that the choice of the level of abstraction takes effect: if too few processes are identified, then there is little room for manœuvre; if too many, then the gross behaviour of the program will be different, and it may no longer meet its specification.

The characterisation of the language of (7) and (12) is arguably the weakest useful one: for any two programs which it does not identify can potentially be told apart by some context. That is to say, there is some testing environment in which one process is bound to 'succeed' but the other may not. The notion of testing is similar to that of Hennessy (13). A program is driven to perform a particular sequence of external communications, by the environment refusing any others; any deadlock at this stage is regarded as success. There are then four conditions which may be tested: absence of divergence, that is that the program can be forced to deadlock; safety, that it must refuse some undesirable communication; liveness, that it cannot refuse all of some set of desirable communications; and termination, that it has reached a successful conclusion with some desired relation holding between its free variables. Note that this does not mean that the programs can be distinguished by such tests in reality: non-determinism means that executing the programs, however often, may never actually complete the sequence of communications after which the behaviours differ, even though that sequence might have arisen had the internal choices been resolved differently; and the stability required to show divergence freedom may take an arbitrary length of time to manifest itself. The mathematics, on the other hand, is able to take an overview of all potential experiments, and notice every possibility of success or failure.

The properties which are ignored in the equivalence include all matters of timing, which on the one hand means that an inefficient but obviously correct program can be shown to have the same overall behaviour as a more efficient and opaque one that is thus seen to be equally correct; but on the other means that reasoning about meeting real-time deadlines, for instance, must be carried on in a more discriminating semantic framework. Another aspect which requires a stricter equivalence is fairness in arbitration of alternatives: whether a communication which one partner is prepared to engage in can be indefinitely held up by the other repeatedly choosing another alternative, when it might have chosen the one in question. Conversely, in some circumstances, such as reasoning about the effect of buffers (see below), it may be desirable to relax the distinctions somewhat. But in a wide area of application, the semantics of (12) capture precisely what matters towards a process meeting its specification, while leaving the rest unconstrained.

TRANSFORMATION TOOL

It is entirely possible to sit down and calculate transformations with pencil and paper. As the programs to transform grow larger, however, and the transformations to perform become more complex, this process necessarily grows more time consuming and prone to error. This is an area where mechanisation as a computer program can help on both those counts, and also provide a more powerful tool.

One of the fruits of a research project we have been involved in for the past two and a half years is an interactive occam Transformation System. This consists of a suite of routines in the functional language Standard ML (Harper *et al* (14), Wikstrom (15)) implementing an abstract syntax for occam, together with a number of operations on it.

A fuller description can be found in Goldsmith (16), and operational details in Goldsmith (17).

Laws

The most important operations are implementations of the laws of (12), interpreted as rewrite rules and implemented as functions mapping processes to processes. The laws are mainly quite trivial, expressing such facts as the commutativity of the parallel operator:

$$\begin{array}{l} \texttt{PAR} \\ \quad P \\ \quad Q \end{array} \equiv \begin{array}{l} \texttt{PAR} \\ \quad Q \\ \quad P \end{array} \qquad (1)$$

There are one or two of slightly greater profundity, such as that elaborating the behaviour of the parallel composition of two `ALT` arbitrations: the processes guarded by external communications may proceed in parallel with the other branch after performing it, as may those which may be selected non-deterministically by an internal `SKIP` guard; internal communications which are not permitted by both halves are blocked, and cannot proceed at this point; and those which do have the consent of both parties elide into a `SKIP` guard followed by, in sequence, an assignment of the value output to the input variable and the parallel composition of the two processes which were guarded.

In addition to the laws of (12) certain others have also been coded explicitly, either to shortcut long derivations for the sake of efficiency, or to avoid the use of induction which would be required in their proof from the basic axioms. They are nearly all at the same low level of complexity.

Actions

In order to operate on real programs, routines are required to parse and display textual representations of the processes; these are provided. In addition, on top of the simple process type an abstract datatype maintains a stack of contexts, allowing the user to browse through a process, to concentrate attention on one part, and to apply laws there rather than to the whole program. One of the features of the browser is a facility to 'fold up' extraneous detail, so as to be able to see the general structure of a large program all at once.

All these features are available through a window-based interface for SUN workstations, with syntax-directed cursor movement and user-definable keys. On other machines the interface is via the standard 'read-eval-print' loop of the ML interpreter.

Strategies

Although the basic laws have a minimalist quality, and a large number of applications in sequence are required to achieve any substantial transformation, the presence of the underlying functional programming system allows these to be composed by higher-order functionals. Recursive strategies may thus be encoded, which apply a transformation throughout a process, or otherwise work at a higher level.

In particular a strategy has been programmed to implement the normal-form reduction. In fact, two have: one operating depth-first, suitable for finite (loop-free) programs; and one which will calculate any desired approximation to an infinite normal form, breadth-first. Both of these are hand-crafted recursive programs, which rely for their validity on careful application of specialist knowledge and programming skills. Other strategies could be similarly encoded, but again this would require a high degree of expertise.

To allow application oriented strategies to be programmed by less specialist users, the basic laws may be treated as the 'machine-code' for a transformation engine. There is a small procedural language whose atomic commands are 'apply L', for each law L, together with the browser commands for moving around the program structure. These can be combined with combinators such as THEN (sequencing) and ELSE (failure trapping) and recursion to give strategies which are guaranteed to implement valid transformations provided all the laws appealed to are themselves valid.

Among strategies which have been encoded in this way is one which appeals to a powerful theoretical result (a unique fixed-point theorem) to demonstrate the equivalence

$$\begin{array}{l} \texttt{CHAN } c, \ldots : \\ \texttt{PAR} \\ \quad \texttt{WHILE } b_0 \\ \quad\quad P_0 \\ \quad \texttt{WHILE } b_1 \\ \quad\quad P_1 \end{array} \equiv \begin{array}{l} \texttt{WHILE } b' \\ \quad P' \end{array} \qquad (2)$$

(when it holds) under certain not very restrictive conditions on the processes involved.

CONFIGURING NETWORKS

In this section we examine one of the large number of applications of program transformation. This is getting a logical network of occam programs to work on a network of transputers which it does not match, typically because the physical network does not have enough links. The main virtue of automated transformations here is that the user is freed from the need to do a lot of delicate and difficult programming, and of course that the results are guaranteed to be semantically valid.

The Problem

Let us suppose we are given an occam program which is the parallel composition of a number (N, say) of simpler programs P_i. Any or all of these can be connected by occam channels, and each pair of processes may have any number of channels between them. The most convenient representation of this structure is a directed graph $\mathcal{L}$ with a node for each P_i and an edge from P_i to P_j just when there is a channel from P_i to P_j. This will be termed the *logical* network. There is no reason why the individual P_i cannot contain any PAR constructs, even at the highest level. The decomposition into P_i will simply be the desired granularity for the actual parallel implementation of the system.

We are also given a network of M transputers, which can be thought of as a graph $\mathcal{P}$ with a node for each transputer and a pair of directed edges for each bidirectional link between connected transputers. Since each transputer has four links, there can be at most four pairs of edges leading to and from each node. This will be termed the *physical* network.

Of course the partitioning of a program or algorithm into P_i will be done with the available architecture in mind and there will usually be many ways of going about this. And often the available transputer network will not be fixed, so that it possible to re-configure the physical network to better match the logical one. When designing parallel programs these are both very important issues and the decisions that are made will be crucially dependent on which logical networks are implementable (and how efficiently) on which networks of processors. In the interest of separation of concerns and brevity, however, in this paper we will do no more than touch on them from time to time.

The logical network is *consistent* with the physical one if there is some assignment of nodes and edges in $\mathcal{L}$ to nodes and edges in $\mathcal{P}$ in such a way that no two logical nodes or edges are mapped to the same physical one and such that a logical edge joining a pair of P_i must be mapped to a physical one joining the images of this pair. Note that this implies that $M \geq N$.

Where $\mathcal{L}$ is consistent with $\mathcal{P}$ there is no problem: it is possible to implement the program with no tricks. We will concern ourselves with the case where they are not consistent. We will assume that $M \geq N$, for if not it is clearly impossible to run the P_i truly in parallel and so it is necessary either to increase M or re-partition the network (possibly combining groups of the old P_i in parallel to make single new ones). We will see later on that it can be useful to have $M > N$, so we will not assume that equality necessarily holds.

The most common case of inconsistency will be where the logical network is simply too highly connected to be mapped onto any system of transputers (i.e., some P_i have more than their allowance of four pairs of input and output channels), though a simple inconsistency between a fixed physical array and the logical network will also appear sometimes.

We should perhaps note here that there will be widely different cases of this problem: ranging from (i) a large network where every process needs a channel to every other, through (ii) a two- or three-dimensional rectangular array where each process needs to be connected in a regular way to more than four neighbours (perhaps 8 in two dimensions, or 6, 18 or 26 in three), to (iii) the case where there is simply the need to have more than one pair of channels between adjacent transputers. It should therefore come as no surprise that there is also a wide range of possible solutions. We will shortly introduce a number of transformations, essentially dealing with these situations in reverse order. The solution adopted for any network will probably be a mixture of ordinarily implemented channels and channels implemented using one or more of these transformations, for there is no reason why all our solutions cannot be mixed arbitrarily with each other and ordinary link communication.

All the methods below carry a significant overhead in terms of communication speed and bandwidth. Therefore they should be avoided where possible in applications where these are close to being overall limiting factors.

Multiplexing

We may very well want a large number of channels between adjacent transputers, even though there will almost always be only one pair of links between them. This might happen because there are channels of different types or because there are in fact several processes running on each transputer which require connection to one or more on the other. The solution to this problem is to *multiplex* all the channels into a single pair.

The transformation requires an extra process (multiplexer) to be run on each of the two transputers. We will refer to all the other processes as the user processes. The various logical channels are then connected to the multiplexer on the same transputer. Note that half of them will have to be renamed. The two multiplexers are then connected by the link pair of channels, and have the function of conveying the communications between the relevant processes. This has the effect of replacing each channel by a buffer, which is not in general semantic-preserving: see the later section on buffer tolerance. But, as described in that section it is possible to eliminate these buffers at the cost of doubling each output by a user process and only allowing the multiplexer to accept the second output when its mate has told it that the relevant communication has been delivered at the other end.

The construction of the multiplexer is only summarised here because of lack of space: the reader is referred to Jones and Goldsmith (18) for the more detailed occam code of such a multiplexer. For simplicity, we suppose that the program is to run forever, and thus avoid the added complexities of organising distributed termination.

Each multiplexer is in fact the parallel composition of a number of parts: this is necessary because the multiplexer must at times be able to output on several channels at once, and while it is possible for a sequential process to be ready to communicate on a number of channels at the same time in an ALT arbitration, the language (to allow efficient implementation) imposes a restriction that all these communications should be inputs. It is convenient to split each up into $K+2$ processes, where K is the number of channels being multiplexed: there are processes P_{in} and P_{out} respectively handling the multiplexer inputs and outputs from the link and one for each external channel. An external channel c which is an input to the multiplexer is given a process of the form

```
              VAR x:
              WHILE TRUE
$Q_c$   =       SEQ
                  c ? x
                  d ! x
                  e ? x
```

where d is connected to P_{out} and e to P_{in}. If it were desired to eliminate the buffering referred to earlier, an extra communication on c should be placed after that on e to match the extra communication which the transformed user process will be performing. Alternatively the Q_c process could be omitted altogether since the transformed user process would then be performing the main function of Q_c, which is that of waiting for an acknowledgement before trying to output again. If this last optimisation were adopted the correct functioning of the whole multiplexer system would depend on the good behaviour of one of the user processes. This will be safe if the coding is performed by the automated transformation system, but is probably unwise in other circumstances!

An external channel which is an output to the multiplexer corresponds to a process of the form

```
              VAR x:
              WHILE TRUE
$R_c$   =       SEQ
                  d ? x
                  c ! x
                  e ? x
```

where d is connected to P_{in} and e to P_{out}. P_{out} is constructed as an infinitely looping alternative construct that alternately accepts an input from any of the Q_c and R_c and transmits it, together with information on which channel it refers to, over the link. This would be one place where the use of *fair* alternative constructs would be desirable (see Roscoe (19) for transformations which realise these). P_{in} is constructed to repeatedly input values from the link, to decode them into destination and content, and pass them on to the waiting Q_c or R_c. The reason why this system works is that no message which comes into P_{in} can ever be refused by Q_c or R_c: the end-to-end acknowledgement which is executed by each pair Q_c or R_c ensures this.

P_{in} and P_{out} will very probably use (in occam 2) a variant type channel between them, since it is very likely that the different channels being multiplexed will themselves be of different types, and even if not the tag of variant types may be an efficient way of encoding which channel is being transmitted. Besides, half the traffic is simple acknowledgements and therefore can be straightforward tagged synchronisations. If the users' channels are themselves of variant types the encoding problem may get a little more complex, but should not be impossible to perform automatically.

This completes our description of the transformation which multiplexes multiple channels between transputers. Like all the others we will be describing, it involves a relatively small change to the original program except for the addition of the transport mechanism.

Dedicated Channels

Given that we can now implement as many channels as we please between adjacent processors, it is a small step to implement channels between distant parts of the network. For all one has to do to connect processor T_i with processor T_j is to choose a route between them and implement on each intervening processor T_k an extra buffer process (to run in parallel with what was already running on T_k). These buffers are then simply connected together using extra channels to form one long buffer connecting T_i and T_j. If no countermeasures were taken the buffering introduced could be considerable, but it is easy to implement end-to-end acknowledgement to eliminate it altogether if desired. This is perhaps the most obvious and transparent way of implementing 'long channels'.

This method can be quite effective if only a few extra channels are required or if all extra channels are short, such as the example mentioned earlier of joining all processes in an array to those diagonally adjacent as well as those immediately adjacent. However even here the overhead is not insignificant, particularly as the required multiplexing slows down not only the long channels but also the ones with which their communications are multiplexed. If the communication patterns in the network are very regular, as is the case in most numerical programs on grids, for example, it will almost certainly be far more efficient to program the indirect passing of information directly into the main program.

Highly Connected Networks

The method outlined in the last section becomes very expensive in terms of space if there are very many channels to implement between distant processors. However it is possible to achieve the same effect in a way in which the space consumed in each processor is a constant plus a constant times the number of channels used by that processor. What one does is to implement the processes Q_c and R_c as before to receive and transmit messages from and to users, but have these processes communicating with some general mail service rather than just two-way links. Provided the mail service satisfies the following conditions this will implement a one-place buffer on every channel.

The mail service should have a node on each relevant processor. The mail service must deliver all messages to their correct destinations in a finite time. Also, a node must not be able to refuse the input of a message from its user for ever. One would also hope that, allowing for nodes which become over-used or blocked, it will choose a sensible route for a message from its sender to destination, taking advantage of the full range of connections in the network. Because of the end-to-end acknowledge between Q_c and R_c the mail service can rely on its users never refusing to accept messages it may bring.

Designing such a mail service is a difficult problem. One solution was given in (19) which built on a correct but inefficient solution for rings of processes to give a practical solution for general connected networks which, except in heavy traffic, is able to pick shortest routes. An elegant solution for two-dimensional rectangular arrays is given in Jesshope and Yantchev (20) where each node is split into two halves: **A** and **B**. Suppose one wants to pass a message from (x, y) to (x', y'). The **A** layer is for increasing either the x or y co-ordinate of a message if necessary until it reaches (x^*, y^*) where $x^* = max(x, x')$ and $y^* = max(y, y')$. It is then passed to the **B** layer which is for decreasing co-ordinates as necessary until it reaches (x', y'). Each half node simply cycles between waiting for an input from any of its input channels to outputting what it receives in an appropriate direction. The results of Roscoe and Dathi (21) are used to prove this system deadlock free. (In fact, we remark that this solution wil work in any number of dimensions—still with two layers o processes performing exactly the same function as above Of course more than two dimensions would require mor than four links per transputer!)

BUFFER TOLERANCE

A *buffer* is a process with a single input channel (**in**, say and a single output channel (**out**, say) which satisfies the following specification. The formal semantics referred to is that of (7).

a) At all times the sequence of values output is an initial subsequence of the sequence of values inpu (formally $s{\upharpoonright}\mathbf{out} \leq s{\upharpoonright}\mathbf{in}$, for all of its traces s).
b) If it is empty then it cannot refuse to input (formally $s{\upharpoonright}\mathbf{out} = s{\upharpoonright}\mathbf{in} \Rightarrow \mathbf{in} \notin Ref$ for all its failures (s, Ref)).
c) If it is not empty then it cannot refuse to outpu (formally $s{\upharpoonright}\mathbf{out} < s{\upharpoonright}\mathbf{in} \Rightarrow \mathbf{out} \notin Ref$ for all its failures (s, Ref)).

To avoid considerations of fairness, we will assume additionally that a buffer never inputs infinitely often withou performing an output (a condition which trivially holds for buffers with any fixed bound).

All of the methods for overcoming the configuration problems described in the previous section introduce, in their simplest form, a buffer along all the channels other than those mapped directly onto links. Thus they typically replace a program such as

```
CHAN $c_1$, ..., $c_r$:
CHAN $d_1$, ..., $d_s$:
PAR
  $P_1$
  $P_2$
  .
  .
  $P_n$
```

where the c_i are the channels which are to be mapped directly onto links and the d_i are those which are to be implemented by the transformation, by one which is semantically equivalent to

```
CHAN $c_1$, ..., $c_r$:
CHAN $d_1$, ..., $d_s$:
CHAN $d'_1$, ..., $d'_s$:
PAR
  $P'_1$
  $P'_2$
  .
  .
  $P'_n$
  $B_1$
  .
  .
  $B_s$
```

where P'_i is P_i with each d-output (i.e., one of the form $d_j!e$) has been replaced by a d' one (i.e., $d'_j!e$) and B_j is a buffer whose input channel is d'_j and whose output channel is d_j. The length of the buffers may be fixed or may vary non-deterministically, depending on which transformation is used.

In each case these buffers can be eliminated and a system which is equivalent to the original one above implemented, but there is always a significant cost to this. In each of the solutions the method is the same: all outputs have to be doubled, so in the P'_i above we would replace each

```
                SEQ
$d'_j$ ! e  by    $d'_j$ ! e
                  $d'_j$ ! c
```

(where c is arbitrary) and the service does not permit the second output until the message has been delivered to its destination. This will be seen to be particularly costly where the distance between P and Q is large.

This buffer elimination is undoubtedly often necessary. In general the introduction of buffers permits a larger set of traces, which can often be strictly larger. To see this one only has to consider the following.

Example 1. Consider the parallel composition of two processes P and Q with a single channel between them. If each process repeatedly inputs an item and then outputs a single item in response, the introduction of a buffer in the internal channel does not change the sequence of input and output values, but can change the relative order between these. If some user of this system relies on the fact that each output results from one of the last two inputs, it will not be satisfied with the buffered version. It is easy to create other occam programs which might run in parallel with this combination and whose function would be destroyed totally by the introduction of the buffer between P and Q.

Example 2. In the previous example the individual behaviours of P and Q were not really altered by the addition of the buffer. This is not always so: consider the pair below where there are two channels leading from P to Q and Q has external channel c. For dramatic effect let us suppose they are part of the program running inside a bomb.

```
P  =  WHILE TRUE
        SEQ
          a ! 0
          b ! 0
Q  =  VAR x, z:
      SEQ
        z := 0
        WHILE TRUE
          SEQ
            ALT
              a ? x
                z := z + 1
              b ? x
                z := z - 1
            IF
              z < 0
                EXPLODE
              z >= 0
                c ! 0
```

If a buffer is inserted only on the b channel then nothing changes, but if one is placed on the a channel (or both) disaster may strike at any time (non-deterministically). Notice also that if z rather than 0 were output in the last line then a b-buffer could change the values output—this could of course significantly affect the rest of the program.

On the other hand there are many widely varying cases where the buffers do not seem to matter. A general classification seems very difficult, but we would like to describe three broad categories with examples.

a) Strong buffer tolerance. Sometimes the introduction of a buffer does not change the semantics of the overall program at all. A channel where this is true can be termed *strongly buffer tolerant*. Indeed it may not be necessary to look beyond the pair of processes using that channel: if buffering does not affect the semantic value of those two running in parallel it cannot affect that of the whole network. One curious example of this was seen above (on the b channel), but far more common are cases where the double communication described above as a way of eliminating buffers is already present. This typically happens where one has a pair of opposite direction channels between two processes: a and b say. If these processes are P and Q, suppose that communications on a and b are paired into initiating communications and replies in the sense that any initiating communication on one is immediately followed by a reply on the other so that neither process can carry out any other communication until the reply has occurred. (It is easiest to imagine this when all initiating communications are one channel and all replies on the other, but it is quite possible to achieve a random mixture—see the crosslink communications of the program developed in (19) for an example of the latter.) In this case both channels can have a buffer added with no semantic effect.

It is also possible for larger scale effects to eliminate the semantic effect of buffers. If in the very first example described above the environment always prevents a difference of more than two between the inputs and outputs of this system, then the introduction of a buffer between the two processes has no effect. In other words rigidity elsewhere might compensate for the buffers.

b) Weak buffer tolerance. Sometimes, although the semantic value does change one does not mind. For example, in a system which takes in inputs at one end and delivers outputs at the other, one might not care about the exact phasing of these. In a broader sense, we might not be concerned with the exact set of traces possible for a network, only with the relationships between the complete traffic on all channels through a process' history. (Here complete might mean finite or infinite.) A good example of this is the Example 1 above, where the introduction of a buffer did not affect this whole-time behaviour, only the phasing of it. We might term such a system *weakly buffer tolerant.*

Formally speaking, we are observing that in certain circumstances the usual semantic model records too much detail and we wish a weaker one where a process is represented by a relation between the complete behaviours on all its channels. Unfortunately, as we have already essentially seen in the Example 2 and in our observations on possible environments for Example 1, this is not a proper semantic model for the whole of occam in that one cannot determine the value of a complete program from the values of its parts. If, however, there were a substantial class of programs for which it were a sensible model this would be very relevant to the study of buffer independence. For a buffer is characterised by the fact that its complete input sequence is always equal to its complete output sequence—if they are infinite, at least—and so the introduction of a buffer on a channel in such a network would have no effect on the semantic value.

This is a topic on which more formal work will have to be done before we can be completely confident. However there are two substantial classes of network which seem to fulfil these conditions and therefore be weakly buffer tolerant. In both cases we have assumed below that the network is nonterminating since this makes analysis easier: if the network were to terminate one would have to be careful about the contents of buffers at that time.

1. The first class of networks is that where (i) there are no edge-cycles in the undirected graph with an edge between two processes for each channel—so in particular there is at most one channel between any pair of processes (ii) the network is deadlock-free and nonterminating and (iii) in each infinite execution sequence every channel is used infinitely often. Notice that for (ii) and (iii) to hold it is sufficient that (i) holds, that (ii) and (iii) hold of each process individually and that the network is connected.
2. The second class is that of networks where (i) there are no `ALT` constructs, (ii) the network is deadlock free and nonterminating and (iii) in each infinite execution sequence every channel is used infinitely often. Here, (iii) holds if it holds of each process, (ii) holds and the network is connected. Note that a proof of deadlock freedom is required: see (21) for some suitable techniques. The intuition behind this example is that (i) (ii) and (iii) ensure that as soon as any communication becomes enabled it must eventually occur. The absence of `ALT` constructs (plus condition (iii)) also ensures that the pattern of communications of each component is determined totally by the component itself and is not influenced by the order in which its neighbours communicate. (This is not a trivial idea, since the reader will note that the components are entitled to include the parallel operator in their definitions, including terminating parallel constructs to communicate on two or more channels in parallel.) This class of networks also has the property of being deterministic.

As we said above, this is an area where further work is required to analyse this weak form of equivalence and to provide formal justifications for these and similar claims. Perhaps the most important thing to bear in mind about weak buffer tolerance is that it seems to be a global property of a whole network: adding an extra process on the edge of a weakly buffer tolerant network (for example one with an `ALT` in the second case above) can destroy the tolerance throughout the whole network.

c) Buffer tolerant specifications. This final class is less well-defined than the previous two, but seems to be important nonetheless. Sometimes a subnetwork (which may vary in size from a single process upwards) is constructed to satisfy some specification: typical examples are the specification of a buffer above and the much more complex requirement for a message routing system set out in (19). A specification can be buffer tolerant in that, if a process satisfies it, then so does the same process with an extra buffer placed on one or more channels. Both of those mentioned above are buffer tolerant on all their external channels.

If the only property which is required of the subnetwork (for the correct functioning of the whole with respect to some overall specification) is that it satisfies this local specification then clearly all of the local specification's buffer tolerant channels may have buffers placed on them. (Note that this may change the overall semantic value of the network, however.)

These observations only allow us to place buffers on the external channels of the subnetwork (for these are the only ones which will be visible to its specification). It is sometimes possible to prove that the network one creates continues to satisfy this specification if a buffer were placed on some internal channel. Of course this would be true if the internal channel were strongly buffer tolerant, but there are certainly other cases. For example, if a buffer is implemented as a chain of smaller buffers then each of the internal channels is buffer tolerant in this sense. Less trivially, it was shown in (19) that the ring channels of the system developed there are buffer tolerant in this sense. Unfortunately this sort of analysis is likely to remain *ad hoc*.

ACKNOWLEDGEMENTS

We are grateful to Geoff Barrett and Geraint Jones of the Programming Research Group for many conversations which have influenced the ideas contained in this paper.

The initial development of the Transformation System was largely funded by project CAD-036 of the UK Alvey Initiative, and owes much to the efforts of A.D.B. Cox, then of the Rutherford Appleton Laboratories. Subsequent developments have been supported by a contract from the Science and Engineering Research Council/Department of Trade and Industry Transputer Initiative.

REFERENCES

1. Hoare, C.A.R., 1978, CACM, 21(8), pp 666–677

2. Hoare, C.A.R., 1985, "Communicating Sequential Processes", Prentice/Hall International, Hemel Hempstead, England

3. Milner, R., 1980, "A Calculus of Communicating Systems", Lecture Notes in Computer Science 92, Springer-Verlag, Berlin

4. INMOS Ltd, 1984, "occam Programming Manual", Prentice/Hall International, Hemel Hempstead, England

5. INMOS Ltd, 1988, "occam 2 Reference Manual", Prentice/Hall International, Hemel Hempstead, England

6. Shepherd, D., 1988, ACM SIGMICRO, (*forthcoming*)

7. Roscoe, A.W., "A Denotational Semantics for occam", *in* Brookes, S.D., Roscoe, A.W. and Winskel, G. (*edd.*), 1985, "Seminar on concurrency, Carnegie-Mellon University, Pittsburgh, PA., July 9-11, 1984", Lecture Notes in Computer Science 197, Springer-Verlag, Berlin, pp 306–329

8. Brookes, S.D. and Roscoe, A.W., "An Improved Failures Model for Communicating Processes", *ibid.*, pp 281–305

9. Milner, R., "Operational and Algebraic Semantics of Concurrent Processes", *in* van Leeuwen, J. (*ed.*), 1989, "Handbook of Theoretical Computer Science", North Holland, Amsterdam (*forthcoming*)

10. Brookes, S.D., Roscoe, A.W. and Walker, D.J., 1988, "An Operational Semantics for CSP", *to appear*

11. Barrett, G., 1988(?), "Semantics and Implementation of occam", DPHIL Thesis, Oxford University (*forthcoming*)

12. Roscoe, A.W. and Hoare, C.A.R., 1986, "Laws of occam Programming", Technical Monograph PRG–53, Programming Research Group, Oxford University

13. Hennessy, M., 1985–6, Lecture Notes, University of Århus

14. Harper, R., MacQueen, D. and Milner, R., 1986, "Standard ML", LFCS Report Series ECS-LFCS-86-2, Laboratory for the Foundations of Computer Science, Edinburgh University

15. Wikstrom, A., 1987, "Functional Programming in Standard ML", Prentice/Hall International, Hemel Hempstead, England

16. Goldsmith, M.H., 1987, "occam Transformation at Oxford", *in* Muntean, T. (*ed.*), Proceedings of 7th occam Users Group & International Workshop on Parallel Programming of Transputer based Machines, Laboratoire de Génie Informatique–Informatique et Mathématiques Appliquées de Grenoble, Grenoble

17. Goldsmith, M.H., 1988, "The Oxford occam Transformation System", documentation, Programming Research Group, Oxford University

18. Jones, G. and Goldsmith, M.H., 1988, "Programming in occam 2", Prentice/Hall International, Hemel Hempstead, England

19. Roscoe, A.W., "Routing messages through networks: an exercise in deadlock avoidance", *in* Muntean, T. (*ed.*) *loc. cit.* (16)

20. Jesshope, C. and Yantchev, J., 1988, "Deadlock free packet routing with bounded buffering for asynchronous regular arrays", Technical Report, Southampton University

21. Roscoe, A.W. and Dathi, N., 1986, "The Pursuit of Deadlock Freedom", Technical Monograph PRG–57, Programming Research Group, Oxford University