17
Using dataflow algebra to analyse the alternating bit protocol

A. J. Cowling & M. C. Nike
Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello Street, Sheffield, S1 4DP, U.K.
Telephone: +44 114 282 5580; Fax: +44 114 278 0972;
Email: A.Cowling @ dcs.shef.uk.ac

Abstract

The alternating bit protocol is taken as a case study of a parallel distributed system, and it is shown how the dataflow algebra approach can be used to specify and then analyse the overall behaviour of a communications system that uses this protocol. The paper summarises the use of dataflow algebras for specifying such systems, and introduces the main features of the protocol that are relevant to the case study. Models are developed for two different cases of the behaviour of the system, distinguished by different conditions on the length of the timeout period that is integral to the operation of the protocol. It is shown that under one of these conditions the overall operation of the protocol is such that it can not be guaranteed to operate correctly, even though the individual processes may operate correctly. A brief comparison is made between the use of the dataflow algebra approach for carrying out such analyses and the use of process algebra models.

Keywords

Dataflow algebra, process algebra, formal specification, formal verification, alternating bit protocol.

1 INTRODUCTION

The fundamental characteristic of parallel or distributed systems is that they consist of a number of concurrent processes which interact in some fashion, so that their correct operation depends not only on the correctness of the individual processes, but also on these processes interacting correctly. In principle it ought to be possible to analyse these interactions in either a top-down or a bottom-up fashion, but in practice the analysis methods available are nearly all bottom-up: that is, they start from descriptions of the behaviour of the individual processes, and then analyse the way in which these are composed to produce the behaviour of the complete system. These descriptions of the individual processes have to contain some representations of their abstract state spaces, and as the descriptions are combined the size of the composite state space for the whole system explodes combinatorially with the number of processes.

I. Jelly et al. (eds.), *Software Engineering for Parallel and Distributed Systems*

Consequently, for these methods the complexity of the analysis process grows similarly, unless ways can be found for pruning it, and so the starting point for this paper is the hypothesis that some form of top-down description of the allowable behaviour of the overall system is needed to guide this pruning. To justify this, the paper presents a case study of an approach that we are developing, based on what we call the dataflow algebra model. This is intended to be complementary to process algebra models, but it emphasises system-wide patterns of communication rather than those for individual processes. The example used in the study is the alternating bit communications protocol, which we have chosen largely because process algebra models of different versions of it have already received a lot of attention, as in Milner (1983), although without producing the results concerning the overall correctness (or lack of it) for the version of the protocol that we present here.

In section 2 of the paper, therefore, we firstly describe the dataflow algebra approach, and in particular explain how it can be used to provide two different levels of detail in the specification of a system, which we term the syntactic and the semantic levels respectively. Section 3 then summarises the important features of the alternating bit protocol, indicating the different variants of it that can exist and identifying the particular variant that is used for this case study. Sections 4 and 5 present specifications of the protocol at these two levels of detail, and then sections 6 and 7 show how these are analysed for two cases, the first representing a situation where the protocol can be guaranteed to operate correctly and the second case one where incorrect operation is proved to be possible. Finally in section 8 we make a comparison between this approach and ones based on process algebra, and summarise the conclusions to be drawn from this comparison.

2 DATAFLOW ALGEBRA MODELS

The basic concepts of dataflow algebra models are described in more detail in Cowling (1995), and are derived from the data flow diagrams used in nearly all systems analysis methodologies, such as SSADM (CCTA, 1990) or Yourdon (1989) and its derivatives. The fundamental model underlying the algebra is that a system consists of processes which communicate via unidirectional channels, so that the basic element of the algebra is an action which consists of a single message moving from a source process s to a destination process d via some channel c: this action is denoted s ! c ? d. At the syntactic level we are then concerned with describing the behaviour of a system in terms of the allowable sequences of such actions, which in CSP terminology (Hoare, 1985) would be the allowable traces, so that the dataflow algebra is essentially an algebra of traces. The basic operations of this algebra are thus the concatenation of two sequences of actions (written s1 ; s2) and the choice between two sequences (written s1 | s2): for the purposes of specifying a system this choice is treated as nondeterministic.

From these basic operations, and the silent action ε, two other main operations can be derived. One of these is repetition, so that s^n for any natural number n denotes n repetitions of s; also s^* denotes zero or more repetitions of s and s^+ denotes one or more repetitions of s. The other operation is parallel composition, written s1 || s2, which denotes the choice between any of the set of sequences obtained by an arbitrary interleaving of the actions of s1 and s2. As will be seen later, in some cases we need to restrict the set of possible interleavings, but the issue of how best to model these still requires further investigation.

A syntactic level specification of a system then consists of the set of expressions that define its permitted traces, and this can also be thought of as forming the production rules for a grammar which will generate the allowable set of traces. Then, what we call a semantic level specification (which is not the same thing as the sets of traces, even though these would often be understood as

constituting the semantics of the grammar) can be developed by incorporating into this description specifications of the data that is contained in the messages. These specifications will define both the types of the data that can be communicated along each channel, and the processing which must be carried out in order to produce the data items that are output onto a channel and to handle the data items received from a channel. Components of these are embedded in the basic grammar to form an attribute grammar, in the same way as attribute grammars for programming languages are used to embed semantic definitions into their syntactic descriptions.

One effect of this embedding is that the sequences of actions define orderings on the execution of the various functions that specify the processing, and these orderings can be used in constructing functions to represent the overall processing of the system. Defining and using such orderings seems to us to be one of the fundamental problems in reasoning about the behaviour of parallel systems, and the approach that we are adopting here is intended to be applicable to any formal specification methodology that could be used for defining the individual components of the processing. In the example presented here, the attributes will be used to embed specifications that have been written in OBJ (Goguen and Winkler, 1988).

3 THE ALTERNATING BIT PROTOCOL

The alternating bit protocol (ABP) was introduced by Bartlett, Scantlebury and Wilkinson (1969) as a means of transmitting data reliably across an unreliable medium. It is an automatic repeat request protocol, in which both the packets of data and the replies carry a single-bit sequence number (the alternation bit) which is used to encode whether or not data has been correctly received on the other side of the medium. In the original description it is assumed that the data contains check bits from which the receiver determines whether to request a retransmission, but in fact that original description would be equally valid if the receiver is merely expected to echo the data back, so that the transmitter checks whether or not it matches what was originally sent.

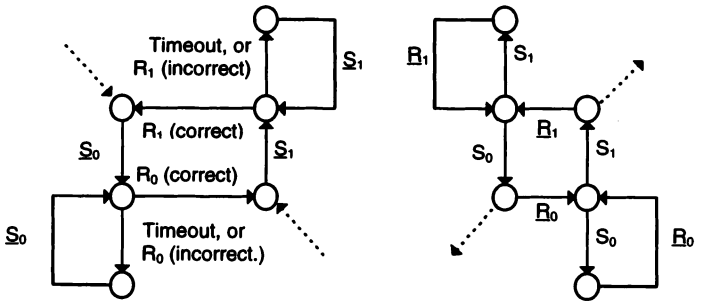

Figure 1 Finite State Automata for the Alternating Bit Protocol.

In Figure 1 the finite state automata for the sender (a) and receiver (b) processes are shown. These are similar to the diagrams in the original, except that here data is only being passed in one direction, and the timeout has been included on the relevant arcs. The labels on the arcs represent values being received or sent by each process, with S denoting data packets and R denoting replies.

The underlined labels represent transmissions, and the subscripts represent the values of the alternation bit for the messages. The dotted arrows in (a) denote the acceptance of a new packet for transmission, after a reply has been received that indicates that the packet had been received correctly. Delivery of the item currently being sent occurs when the receiver accepts a packet with the alternation bit "flipped", and the dotted arrows in (b) show where this happens.

4 A SYNTACTIC SPECIFICATION

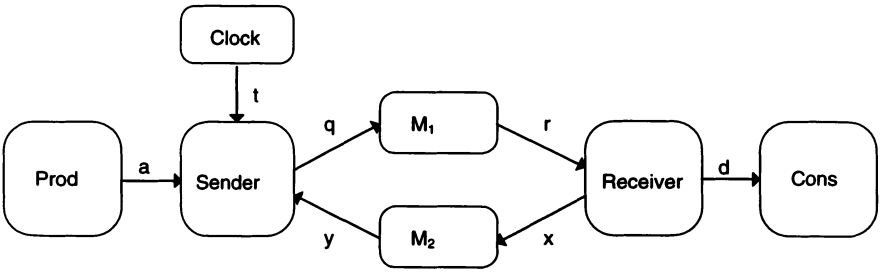

Figure 2 Data Flow diagram for the Alternating Bit Protocol system.

Figure 2 shows the data flow diagram for a system using this protocol, with a producer process Prod using the intermediate processes to transmit packets to a consumer process Cons. For simplicity, it is assumed that the medium can be modelled as two separate processes, M_1 and M_2, representing respectively the medium for the journey from the sender to receiver and vice-versa. The medium can lose or corrupt packets, but it is assumed that it can not corrupt the value of the alternation bit in a packet. Also for simplicity, only a single channel is shown from the Clock, on which timeouts are signalled to the Sender: in practice there would need to be a reverse channel as well, on which messages were sent to start the clock timing. The topology of this system can then be defined by listing the basic actions that can occur, as follows: these will be the terminal symbols for any syntactic specification of the system.

ToS = Prod ! a ? Sender
S_1 = Sender ! q ? M_1
R_1 = Receiver ! x ? M_2
Deliver = Receiver ! d ? Cons

timeout = Clock ! t ? Sender
S_2 = M_1 ! r ? Receiver
R_2 = M_2 ! y ? Sender

4.1 Simple Cyclic Operation

If no corruption or loss occurs, then the operation of the protocol will be cyclic, so that in terms of these actions the basic loop for sending one packet would be denoted S_1 ; S_2 ; R_1 ; R_2. This simple operation does, however, depend on the value of the timeout being greater than the maximum time that can be taken for the whole loop, so that no parallel actions can take place. Under this assumption, the full protocol ABP can be described as follows:

Start = S_1 ; (timeout ; S1)* ; S_2
Reply = R_1 ; (timeout | R_2)
OneCyle = MStart ; Reply ; Loop1*

MStart = ToS ; Start ; Deliver
Loop1 = Start ; Reply
ABP = $OneCyle^n$

The basic operation of this is that the ToS action represents a new value being accepted for transmission, and the flipping of the alternation bit. Any packet loss results in a timeout, as in both Start (where the subsequent restarting of the cycle is shown explicitly) and Reply (where the restarting is implicit in the definition of OneCycle). When a packet with this new bit is received across the medium, the Receiver delivers the currently held message and holds the new one until the bit flips again. The Loop1 component describes the fact that after the Deliver there may be multiple retransmissions of the same piece of data: once they finish OneCycle is over and a new one can begin. Therefore, ABP is simply an arbitrary number of executions of this cycle.

4.2 Parallel Operation

The time taken for the basic cycle (ie S_1 ; S_2 ; R_1 ; R_2) is not usually deterministic, and so if the timeout period were to be gradually reduced there would come a point in the operation where a timeout would sometimes occur part way through a cycle that had not yet finished, and so would trigger the start of a new cycle in parallel with the existing one. The significance of this case is that it could give rise to unreliable operation, as a consequence of the system-wide behaviour being incorrect rather than because of any failure of an individual process. To see how this could occur requires a more elaborate model of the overall behaviour, to describe the possible parallelism: a time sequence diagram for the actions involved is shown in Figure 3.

If we assume that the timeout period is larger than the time for a "half loop", but smaller than the time for a whole loop, then essentially the parallel operation can start once an outward message has reached Receiver, as represented in the diagram by the first part of the main sequence, labelled Start 1. At that point a timeout could occur, indicated by the dotted line, and this would cause a duplicate sequence to start up, labelled Duplicate 1: this operates in parallel with the reply being transmitted back as part of the main sequence (labelled Rest 1). Any further timeout, however, could not occur as part of this duplicate sequence, as by then the original cycle would have finished (indicated by the dotted line after Rest 1) and a new main loop would have started, labelled Start 2: consequently any such timeout would form part of that, indicated by the dotted line after Start 2. Under these assumptions, therefore, there can at most be two activities occurring in parallel. (Further shortening the timeout period could give rise to the possibility of more parallel activities, but we shall not consider that case in this paper.)

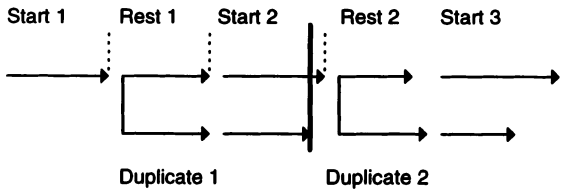

Figure 3 Time sequence diagram for parallel operation.

Thus, while the parallel activity of the duplicate sequence can continue beyond the start of the next main loop, it can only overlap with it so far, indicated by the thick vertical line. In principle, therefore, it must have finished before the point in the main loop at which another duplicate could start (ie the start of Rest 2). In practice, even though the sequence must have finished by then, its effects may not have, in that the duplicate may have returned a message to the Sender which effectively fools it into acting as though the new main sequence has already finished, and so causes it to send another message out. This is, however, modelled satisfactorily by the possibility of a new duplicate sequence starting up, as indicated by the start of Duplicate 2, except that in this case there will be no timeout involved, either in the duplicate sequence or the main one (ie in this case the dotted line before Rest 2 would not correspond to a timeout). Thus, although some duplicate sequences will start with a timeout, not all of them will. The duplicate sequence can therefore be defined as follows.

Duplicate = (timeout | ε) ; S_1 ; DupRest DupRest = ε | S_2 ; R_1 | S_2 ; R_1 ; R_2

For the main loop there are four different cases that need to be considered, although not all are shown in detail in Figure 3. The cases depend on whether or not the alternation bit has flipped at the start of the loop, and whether or not the medium M_1 loses the message on the first time round the loop. With Start and Reply defined as in the simple cyclic case above, these four cases can in principle be expressed as

```
Loop2 = ((Reply ; S1 ; S2) || Duplicate)                          -- bit not flipped, no loss
      | (((Reply ; S1 ; timeout) || Duplicate) ; Start)           -- bit not flipped, loss
      | ((Reply ; ToS ; S1 ; S2 ; Deliver) || Duplicate)          -- bit flipped, no loss
      | (((Reply ; ToS ; S1 ; timeout) || Duplicate) ; Start ; Deliver)-- bit flipped, loss
ABP = MStart ; Loop2^n
```

In practice, though, there are two problems with this specification, both arising from the fact that the parallel composition operation s1 || s2 is defined in terms of arbitrary interleavings of the sequences s1 and s2. Here, since both Reply and Duplicate can contain the actions R_1 and R_2, an arbitrary interleaving of them could contain either R_1 ; R_1 ; R_2 ; R_2 or R_1 ; R_2 ; R_1 ; R_2. One problem is therefore that the first of these would only be valid if the medium M_2 had some internal buffering capacity, and while this might be the case in a complex communications system, it would not be true of the simplest form of medium. The other problem is that the second interleaving is valid, but only makes sense if the first R_1 were associated with Reply rather than Duplicate. We are still exploring various ways of expressing the sort of constraints that are required in these two cases, but space does not permit discussion of them here. We are, however, satisfied that they can be expressed, and that for this example it is legitimate to assume where necessary that the messages pass through the medium M_2 in the order that one would expect.

5 A SEMANTIC SPECIFICATION

In the grammar, an action describes the output of a value from one process and the input of this value into another process. The semantics of an action can therefore be described by defining functions which model the input/output behaviours of the component processes. If each process has an output and input function, then the behaviour of an action will consist of the input function of the

second process applied to the result produced by the output function of the first process. Then, from these expressions describing the input and output of each action, expressions can be built up to describe the behaviour of sequences of actions, and so finally an expression can be built up describing the behaviour of the whole system. In this particular example, we want to use this expression to show that the stream of values input into the system by Prod are all output at the other end to Cons, without any corruption. This can be done in the simple cyclic case, but not in the case of parallel operation, where we shall show that the protocol can be unstable.

Attribute grammars (Knuth, 1968) can be used to describe this level of operation of the system, in similar fashion to their use for specifying compilation of programming languages and other computations that involve tree structures (Kastens, 1991). Thus, each basic action is required to have attributes defined for it that we shall call Src and Dest, whose values will be the appropriate functions. In addition, there will have to be global variables representing the effects of these on the states of the different processes, and these will need to be passed around as an attribute of the actions, and used to construct the corresponding attributes for complete sequences. The semantic specification will therefore be built up by constructing specifications for the individual processes, and then attaching the appropriate functions as attributes to the actions. Attributes for the sequences can then be synthesised from these.

5.1 Process Specifications

For each process, the specification is defined in terms of some suitable abstraction of its state, and so could in principle be treated as a hidden state algebra (Goguen, 1991), although in presenting this example it is simpler to have the state models explicit. For each specification, therefore, the state of the process will be denoted simply by State, and then input operations defined to correspond to the generators for this abstract state, and output operations defined as observers of the state. The equations of the specification will then define in abstract terms how the outputs are computed from the inputs, and so can be used to reduce expressions involving these outputs.

In what follows, the basic type Message is assumed to be defined as Bit × Value, with observer operations bit and value, and with an operation flip defined on the type Bit. In principle Value could be any suitable type, which in practice would probably be a string of characters: it is assumed here to include a null element ε which is different to any meaningful value that can be transmitted. Models are given for each of the four processes in the system.

The Media

These two processes are identical, and each requires an input function, which accepts a message and stores it in some internal "register" (ie its state), and an output function which delivers the contents of that register. Each also requires a "corruption" function, which randomly corrupts a message's value, but not its alternation bit, and which does so in such a way that corruption in one medium can not reverse the effects of corruption in the other. The signatures of and equations for these functions are as follows.

$M_1.in : Message \times State \rightarrow State$
$M_1.out : State \rightarrow Message$
$c : Message \rightarrow Message$

$M_1.out(M_1.in(a,b)) = c(a)$
$bit(c(x)) = bit(x)$

$M_2.in : Message \times State \rightarrow State$
$M_2.out : State \rightarrow message$
$d : Message \rightarrow Message$

$M_2.out(M_2.in(a,b)) = d(a)$
$bit(d(x)) = bit(x)$

value(c(x)) = value(x), or value(d(x)) = value(x), or
value(c(x)) ≠ value(x), randomly value(d(x)) ≠ value(x), randomly
value(c(x)) ≠ value(x) ⇒ value(d(c(x))) ≠ value(x)

The Sender

The state of this process consists of a pair of registers: the first one to hold the data that is currently being sent across the network, and the second one for the currently returned value. The process then requires two input functions and one output function, with the following signatures:

$Sender.in_1$: Value × State → State -- input of a new item of data from Prod
$Sender.in_2$: Message × State → State -- input of an item of data from R_2
Sender.out : State → Message

Note that there does not need to be an input function to correspond to the arrival of a timeout, because this is a control flow rather than a data flow, and so its arrival does not alter the data state of the process. In terms of the state these three functions can be defined as follows, and equations derived from the definitions for reducing expressions involving these functions.

$Sender.in_1(x, (v_1,v_2)) = ((flip(bit(v_1)),x) , (bit(v_2), \varepsilon))$
$Sender.in_2(y, (v_1,v_2)) =$ **if** $bit(y) = bit(v_1)$ **then** (v_1,y) **else** (v_1, v_2) **endif**
$Sender.out((v_1,v_2)) = v_1$

$value(Sender.out(Sender.in_1(x, (v_1,v_2)))) = x$ (1)
$bit(Sender.out(Sender.in_1(x, (v_1,v_2)))) = flip(bit(v_1))$ (2)
$Sender.out(Sender.in_2(x, (v_1,v_2))) = v_1$ (3)

Here, rules 1 and 2 represent the activity of a newly accepted value x being ouput as a message, while rule 3 represents the retransmission of an already held value (ie the one in the first register) after a comparison has proved incorrect, or after a "rogue" reply has been discarded.

The Receiver

This process also needs a pair of registers: the first one to hold the data that is currently being received, and the second one forming an "output buffer" into which it moves the current message that it is holding (ready for delivery) when it receives a message with the bit flipped. A full analysis of whether the system could be made to operate correctly would require complete histories to be stored, but that is not necessary here.

In terms of the state changes, there need to be two basic input functions, one (in_1) for the case where the bit has not changed and the other (in_2) for the case where it has: the function in can then be synthesised from these. There will also be two output functions, one corresponding to a Deliver action and the other to the transmission of a reply. The signatures of these operations will then be:

$Receiver.in_1$, $Receiver.in_2$, Receiver.in : Message × State → State
$Receiver.out_1$: State → Value -- output for "deliver" port
$Receiver.out_2$: State → Message -- output for "retransmit" port

The equations defining these, and the reductions that can be made, are:

$Receiver.in_1(x, (v_1,v_2)) = (x,v_2)$ -- new message replaces old in "current" buffer
$Receiver.in_2(x, (v_1,v_2)) = (x,v_1)$ -- also old message (v_1) placed in "output" buffer
$Receiver.in(x, (v_1,v_2)) =$ **if** $bit(x) = bit(v_1)$
then $Receiver.in_1(x, (v_1,v_2))$ **else** $Receiver.in_2(x, (v_1,v_2))$ **endif**
$Receiver.out_1((v_1,v_2)) = value(v_2)$ -- deliver "output" buffer value
$Receiver.out_2((v_1,v_2)) = v_1$ -- current message output for retransmission

$Receiver.out_2(\ Receiver.in(x_i, (v_1,v_2))\ ) = x_i$ (4)
$Receiver.out_1(\ Receiver.in_2(x_i, (x_{i-1},v_2))\ ) = value(x_{i-1})$ (5)

Here, rule 4 represents the retransmission of a received message back to the sender, where the same message is sent back irrespective of which of the two input functions was invoked. Rule 5 represents the delivery of the value of the old message held (viz x_{i-1}) in the case where a value has just been accepted with a different bit.

5.2 Actions And Their Attributes

A global attribute States contains the array of process states, where the elements will be denoted as though indexed by the process names. Components of the state of a process p will be denoted as States[p,1] and States[p,2], as we have not formally defined projection functions for the pairs. Then, for each action the attributes Src and Dest can be defined in terms of the operations introduced above, and hence its effect can be synthesised. Most of the actions will have the same basic form, and this is illustrated for R_2, which uses M_2.out and $Sender.in_2$:

Action R_2 **is** $R_2.Src = M_2.out(States[M_2])$
$R_2.Dest = Sender.in_2(R_2.Src, States[Sender])$
$States[Sender]=R_2.Dest$
EndAction

Similarly, the S_1 action uses Sender.out and M_1.in; the action for S_2 uses M_1.out and Receiver.in; and the action for R_1 uses $Receiver.out_2$ and M_2.in. The only exceptions to this form are ToS and Deliver, since we have not specified any abstract operations for Prod or Cons, and so have to define directly the values accepted or delivered. These actions will therefore be written as:

Action ToS **is** $ToS.Src = x_i$
$ToS.Dest = Sender.in_1.(ToS.Src, States[Sender])$
$States[Sender] = ToS.Dest$
EndAction

Action Deliver **is** $Deliver.Src = Receiver.out_1(States[Receiver])$ **EndAction**

6 ANALYSIS OF RELIABLE OPERATION

To show that a value will be delivered correctly, a sequence has to be "translated" into the expressions it forms, which can be illustrated with the following example:

ToS ; S_1 ; timeout ; S_1 ; S_2 ; Deliver ; R_1 ; R_2 ; ToS ; S_1 ; S_2 ; Deliver

Each ToS action marks the start of the loop OneCycle, and associated with it will be an invariant that for States[Sender,1], States[Sender,2] and States[Receiver,1] the bits must all be the same. Proving maintenance of this invariant will be an important part of proving correct operation for the overall system. We assume that Sender is initially in state (v,v), where v = $(0,x_{i-1})$, meaning that a succesful loop has been completed; and that Receiver is in the state (v,y), where bit(y) = 0. Using the attributes we can determine the state change resulting from each individual action, and the equations can then be used to reduce these expressions and the ones that are built up from a sequence of actions. Table 1 sets out the results for the above sequence, showing the state changes that result from each action, to establish that the input value x_i is finally delivered correctly.

Table 1 State changes for a sequence of actions that operates correctly

	States				
Action	*[Sender]*	*[M1]*	*[Receiver]*	*[M2]*	*Notes*
ToS	$(1,x_i),(0,\varepsilon)$	-	-	-	new bit = 1
S_1	-	$(1,x_i)$	-	-	but is lost
timeout	-	-	-	-	no effect on data state
S_1	-	$(1,x_i)$	-	-	
S_2	-	-	$c_k(1,x_i),v$	-	new bit = 1
Deliver	-	-	-	-	output previous value of v
R_1	-	-	-	$c_k(1,x_i)$	
R_2	$(1,x_i),d_j(c_k(1,x_i))$	-	-	-	new bit = 1
ToS	$(0,x_{i+1}),(1,\varepsilon)$	-	-	-	
S_1	-	$(0,x_{i+1})$	-	-	
S_2	-	-	$c_{k+1}(0,x_{i+1}),c_k(1,x_i)$	-	
Deliver	-	-	-	-	output value$(c_k(1,x_i))$

Here, the significance of the bit changes is that the first S_2 action reestablishes one part of the invariant, and then the R_2 action reestablishes the second part. As the sequence continues after this R_2 action with a ToS (ie another loop of OneCycle), we must have value(States[Sender,2]) = value(States[Sender,1]): otherwise further iterations of Loop1 would occur until this condition was satisfied. The condition itself reduces to value$(d_j(c_k(1,x_i)))$ = x_i, and as d_j can not reverse a corruption produced by c_k we must therefore have value$(c_k(1,x_i))$ = x_i. This guarantees that x_i is output unchanged from the Receiver, which is the property that we wished to show.

7 ANALYSIS OF FAULTY OPERATION

The faulty operation can only occur in the case where a timeout causes parallel activity to occur, and this is illustrated by the sequence of actions shown in Figure 4, where the values of the bit associated with the message are shown above each action, along with an M or D to indicate whether the action is part of the Main or Duplicate sub-sequence respectively.

The basic operation of this sequence can be described informally as follows. It begins with x_i being accepted by the Sender and sent, not being corrupted by c_1, and being received, resulting in a Deliver that outputs x_{i-1}. A timeout then occurs, causing another copy of x_i to be sent. This is

corrupted by c_2, but when it reaches the Receiver it will be stored, replacing the previous copy. Meanwhile, the correct copy will have been received back at the Sender, but has not been corrupted by d_1. The Sender will therefore accept a new value x_{i+1}, will flip the bit, and so will ignore the corrupted version (because it has the wrong bit value) when it is returned. Thus, when the message containing x_{i+1} is received, with its bit different to the one sent with x_i, the currently stored value of x_i will be delivered, even though it is corrupt.

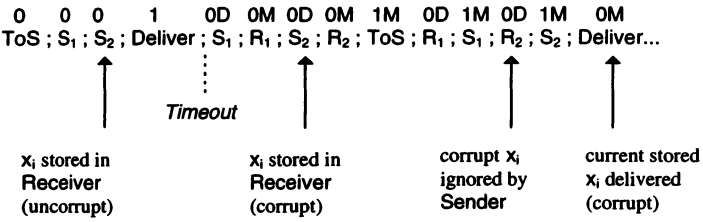

Figure 4 A sequence of actions that operates incorrectly

This can be analysed formally by working through the sequence of attributes to the actions in the same way as was done above for the simple cyclic operation. The results of this are presented in Table 2.

Table 2 State changes for a sequence of actions that operates incorrectly

	States				
Action	*[Sender]*	*[M1]*	*[Receiver]*	*[M2]*	*Notes*
ToS	$(1,x_i),(0,\varepsilon)$	-	-	-	
S_1	-	$(1,x_i)$	-	-	
S_2	-	-	$c_1(1,x_i),(0,x_{i-1})$	-	c_1 does not corrupt
Deliver	-	-	-	-	x_{i-1} is output
S_1	-	$(1,x_i)$	-	-	
R_1	-	-	-	$c_1(1,x_i)$	
S_2	-	-	$c_2(1,x_i),(0,x_{i-1})$	-	c_2 does corrupt
R_2	$(1,x_i),d_1(c_1(1,x_i))$	-	-	-	d_1 does not corrupt
ToS	$(0,x_{i+1}),(1,\varepsilon)$	-	-	-	
R_1	-	-	-	$c_2(1,x_i)$	
S_1	-	$(0,x_{i+1})$	-	-	
R_2	unchanged	-	-	-	input has wrong bit
S_2	-	-	$c_3(0,x_{i+1}),c_2(1,x_i)$	-	
Deliver	-	-	-	-	corrupt x_i output

8 SUMMARY AND CONCLUSIONS

We have therefore been able to demonstrate that the dataflow algebra model of this version of the alternating bit protocol is sufficiently powerful to enable the overall correctness of the system to be analysed, both in cases where it is guaranteed to function correctly and cases where it will fail.

These results are not particularly significant for the protocol (where they merely illustrate why setting timeout periods in protocols is a tricky problem for communications engineers), as most practical protocols now use some form of check bits rather than echo checking, and a version of this protocol that used check bits would not fail in the way that has been analysed here. What is important about these results is that, once the syntactic specification has been built for the whole system and the functional specifications constructed for the individual processes, then not only is their synthesis into the semantic specification almost mechanical, but so too is the analysis of it.

This simplicity is not because of any particular properties of the underlying computational model, as it appears that this should be equivalent to that of a value-passing process algebra such as the one used in (Bezem and Groote, 1994), although there is still much more work needed to explore the relationship between the dataflow algebra and process algebra approaches. Rather, the dataflow algebra approach gains its simplicity by making explicit the underlying chains of cause and effect within the processing that are implemented by the passing of messages between processes, whereas in a process algebra approach these chains need to be inferred (if possible) from patterns of behaviour that are implicit within the action trees generated by the algebra. In the dataflow algebra approach these explicit chains of causality are reflected in the essentially linear nature of the sequences, which then makes straightforward the derivation and reduction of the expressions for the state changes. Also, reasoning about the properties of the system is guided by the occurrence of loop structures in the sequences, as these highlight points where suitable invariants and stopping conditions need to be defined over the states of the processes.

Making explicit these chains of causality should also have a practical benefit, in that experience of the process of designing parallel and distributed systems suggests that designers will usually have at least an intuitive idea of the pattern of causality that they want to create. Indeed, some design approaches place considerable emphasis on describing the input-output behaviour of systems in terms of such patterns: for example the "use cases" of the ObjectOry method (Jacobson *et al*, 1992). Providing a specification methodology which allows designers to capture this intuition within a formal notation, so that it can be reasoned about, should therefore assist in formalising the development of such systems.

To be of maximum benefit, of course, this approach will need to be related more firmly to the models used to capture the behaviour of individual processes, such as process algebra models, but this still requires further investigation, along with the issue of how best to express the sort of relationships between the semantic and syntactic levels of specification that are important to constraining the interleavings generated by parallel composition of sequences of actions. Despite this, though, we believe that this example has demonstrated the validity of the dataflow algebra approach, and its potential value as a model for supporting the process of reasoning about the correctness of the overall behaviour of parallel and distributed systems.

9 ACKNOWLEDGEMENTS

The germ of the original idea behind this work was provided by George Wilson, and its development was greatly assisted by discussions with colleagues in the parallel processing research group, and notably Jon Kerridge, about their approaches to designing concurrent systems. The work done by Dave Cash in developing an early version of the notation used here for the syntactic level of the specification is also acknowledged.

10 REFERENCES

Bartlett, K., Scantlebury, R. and Wilkinson, W. (1969) A Note on Reliable Full-duplex Transmission over Half-duplex Links. *Communications of the ACM*, **12**, 260-261.

Bezem, M.A. and Groote, J.F. (1994) A Correctness Proof of a One-bit Sliding Window Protocol in μCRL. *Computer Journal*, **37**, 289-307.

C.C.T.A. (1990) *SSADM version 4 reference manual*. NCC Blackwell, Oxford.

Cowling, A.J. (1995) *Dataflow Algebras as Formal Specifications of Data Flows*. University of Sheffield Department of Computer Science Research Report **CS-95-18**.

Goguen, J.A. and Winkler, T (1988) *Introducing OBJ3*. Technical Report **SRI-CSL-88-9**, SRI International, Menlo Park, CA.

Goguen, J.A. (1991) Types as Theories, in *Topology and Category Theory in Computer Science* (eds. G.M. Reed, A.W. Roscoe and R.F. Wachter), Oxford University Press, Oxford.

Hoare, C.A.R. (1985) *Communicating Sequential Processes*. Prentice Hall, Englewood Cliffs, NJ.

Jacobson, I., Christerson, M., Jonsson, P. and Övergaard, G. (1992) *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Wokingham.

Kastens, U. (1991) Attribute Grammars as a Specification Method, in *Attribute Grammars, Applications and Systems* (eds. H. Alblas and B. Melichar), Lecture Notes in Computer Science **545**, Springer-Verlag, Berlin.

Knuth, D.E. (1968) Semantics of Context-Free languages. *Mathematical Systems Theory*, **2**, 127-145.

Milner, R. (1983) *Communication and Concurrency*. Prentice Hall, Englewood Cliffs, NJ.

Yourdon, E. (1989) *Modern Structured Analysis*. Prentice Hall, Englewood Cliffs, NJ.

11 BIOGRAPHIES

Tony Cowling obtained a BSc with Honours in Computational Science and Mathematics in 1970 from the University of Leeds, followed by a PhD in 1977. Since 1973 he has been a lecturer in the Department of Computer Science at the University of Sheffield, with research interests in the formal modelling of software systems and in the teaching of software engineering.

Martin Nike obtained a BSc with Honours in Physics and Computing from Coventry University in 1992. He then gained an MSc in Parallel Computing and Computation from the University of Warwick in 1994. He is currently studying for a PhD at the University of Sheffield.