Verification of flow control protocols.

Klaus Hansen

DIKU, University of Copenhagen
Sigurdsgade 41, DK-2200 Copenhagen N

ABSTRACT

An important issue in computer network design is end-to-end control, a term covering error control and flow control. The validity of schemes for these purposes may be demonstrated using simple formal models, in which assertions about central program variables are proven. S.Krogdahl [2] and D.E.Knuth [1] have shown the validity of error control for a class of data link protocols using such methods. In this paper their method is extended to cover flow control. The method is illustrated by proving assertions for a simple case, and it is shown how complex systems may be seen as compositions of the simple case.

1. Introduction.

End-to-end protocols in computer networks guarantee that a data path between a producer (or data source) and a consumer (or data sink) provides a reliable service. An end-to-end protocol incorporates mechanisms for error control and flow control.

Error control covers the aspects of preserving the sequence as well as detecting loss, duplication or corruption of data. This is important when data are transfered over a data path with a significant probability for errors.

Flow control covers resource management. Transmission of data involves a certain amount of copying, and flow control methods are employed to utilize limited buffer resources. A flow control scheme is valid, when it ensures that transmission is always possible in spite of limited resources. Other aspects of flow control, i.e. smooth flow and good utilization of resources, are not treated in this paper. They involve statistical behaviour of the components, and a treatment calls for e.g. queueing theory, which is outside the scope of this paper.

The method used to verify schemes is similar to that in [1]. A scheme is given by a set of algorithms working on a set of common variables. Each algorithms is given a name and is called an operation. The particular order in which the operations will be performed is immaterial for the purpose. To cite:

"Our goal is to derive facts about any scheme that is based on these operations; it is in this sense we are studying a "protocol skeleton" for a large class of conceivable protocols. The facts we shall derive are expressed in terms of relations that remain *invariant* under all operations."

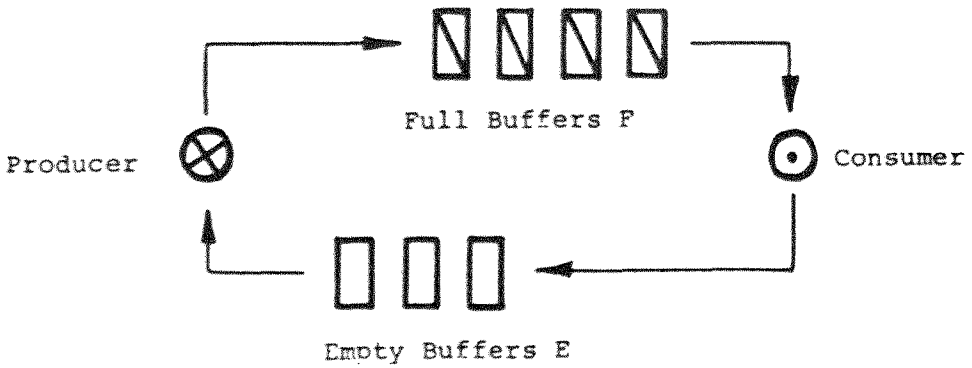
The scheme is thus completed by specifying a set of assertions about the variables.

The set of assertions consists of global and local assertions, the global assertions specifying the behaviour of combinations of operations, and the local assertions determining the applicability of each operation.

The method is applied to a simple case and two composite cases. The cases are simplified versions of protocols found in real networks; the simplification is done by ignoring the representation of various data, while keeping the essential parts of the protocols concerned with queues and feedback.

2. Flow control.

A constrained producer/consumer system (figure 1) consists of a producer of full buffers, a consumer that empties them, and two queues.

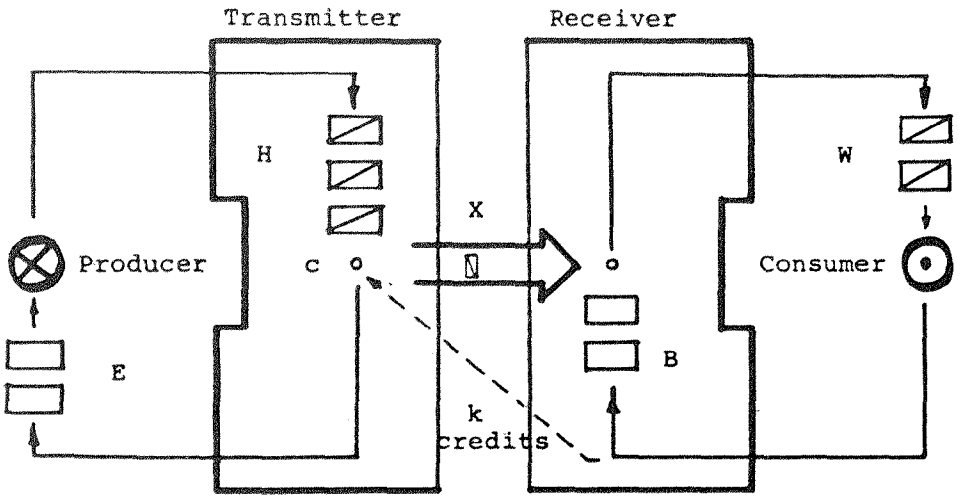


(F denotes queue of full buffers, E queue of empty)

Figure 1: Constrained producer/consumer system.

Flow control is implicit in the synchronization method used, as a process is blocked (waits) whenever a queue is empty. The system is stable irrespectively of the relative speeds; this is one of the purposes of synchronization methods.

However when the system is split into two and a transmission system is added (figure 2), it is necessary to provide a buffer pool at the consumer end, as the transmission path acts as a copying mechanism only, which cannot buffer data. We assume the transmission path to be error-free and have limited, but positive speed.



Note: lower case letters denote numbers.
The queues are: E empty buffers, H buffers held before transmission, B auxiliary pool of empty, W full buffers waiting to be consumed. X is transmission path, k credits in transit, c available credits at T.

Figure 2: Producer/consumer with transmission system.

The queue of full buffers (F) has been split in three: buffers waiting for transmission (H), buffers in transit (X), and buffers delivered to the consumer, but waiting to be emptied (W). The producer fills an empty buffer from E and delivers it to the transmitter, where it is put into the H queue. At an appropriate time, it is put into the transmission path, and the buffer is returned empty to E. When data arrives at the receiver, it is put into a buffer taken from B and delivered to the consumer queue W. After emptying it, the consumer returns it to B. X may contain any number of buffers, depending on propagation time and speed of the data path.

The feed-back needed is provided by a stream of credits that conveys information about empty buffers, one credit representing one empty buffer.

The purpose of flow control is now to avoid the situation where B is empty when data are received, an unrecoverable situation. If the number of buffers in X and B are called x and b respectively, this may be expressed as

$$x \leq b \qquad (1.1)$$

i.e. the number of buffers in transit must not exceed the current size of B.

The full description of the variables is:

e	size of empty queue (E)	initially $\tilde{e}$
h	size of hold queue (H)	initially 0
x	number of buffers in transit	initially 0
b	size of free queue (B)	initially $\tilde{b}$
w	size of queue of waiting (W)	initially 0
k	credits in transit	initially 0
c	current credits at transmitter	initially $\tilde{b}$

3. Operations and assertions for the simple system.

Using the variables described above, we can give a scheme for the simple system in figure 2 that has five operations, three for the transmitter (T1, T2, T3) and two for the receiver (R1, R2). The operations are given below. The comments enclosed in (* and *) shows the semantics of the statement, which is otherwise only specified by the effect on the common variables. Local assertions are shown for each operation, and have to be true for the operation to be applicable, e.g. h>0 and c>0 for a buffer to be transmitted by operation T2. A protocol using flow control will consist of some sequence of these operations; a sequence is valid only if the local assertions are true when an operation is invoked.

```
T1: PUT FULL
(* assert e>0 *)
    e:= e-1;
    h:= h+1;      (* enqueue *)

T2: DELIVER FULL
(* assert h>0 *)
    h:= h-1;      (* dequeue *)
    c:= c-1;
    x:= x+1;      (* SEND DATA *)
    e:= e+1;      (* deliver empty*)

T3: CREDIT ARRIVED
(* assert k>0 *)
    k:= k-1;
    c:= c+1;

R1: FULL ARRIVED
(* assert x>0 *)
    x:= x-1;
    b:= b-1;      (* dequeue *)
    w:= w+1;      (* deliver full *)

R2: PUT EMPTY
(* assert w>0 *)
    w:= w-1;
    b:= b+1;      (* enqueue *)
    k:= k+1;      (* SEND CREDIT *)
```

The global assertions are as follows. All variables except c are non-negative; for c a weaker assumption $c+k \geq 0$ may be used (the meaning of this is not elaborated here).First, the purpose of flow control is reformulated as

$$c+k=b-x \tag{1.1a}$$

Second, no buffers may be lost or misplaced at T or R:

$$e+h=\tilde{e} \tag{1.2}$$

$$w+b=\tilde{b} \tag{1.3}$$

It is easily seen that the assertions are true for the initial values and invariant under the operations. To prove $x \leq b$ observe that

$$b-x = c+k \quad \text{and} \quad c+k \geq 0$$

which implies that

$$x-b \leq 0$$

The global assertions will then be true, and the total scheme be valid.

A simplified version of the operations is worth considering. When the variables c and k, and operation T3 are omitted, there is no feed-back. If the consumer and transmission path are faster than the consumer, it is easy to construct a valid sequence which violates equation (1.1); an example is (T1 T2 R1) repeated a sufficient number of times. This scheme is invalid for flow control; but it may of course be used if one dares to use timing considerations in validity arguments.

4. Composite systems.

Two common ways of obtaining more complex schemes are to combine simple schemes by concatenation or nesting, in the following called serial and multiplexed flow control.

Serial flow control is found in gateways between different nets and in front-ends for host computers.

Multiplexing and the resulting nesting of flow control is common, examples are the X.25 datalink and DCE-DTE levels, or the datalink, network and transport layers in the Open Systems Interconnection model [3].

5. Serial flow control.

Figure 3 shows the concatenation of two schemes. The arguments above are easily generalized to a concatenation of simple schemes joined in the obvious way. The five operations are still valid, with a change of variables to indexed variables, to distinguish between the component schemes. As a consequence, some variable names are synonyms:

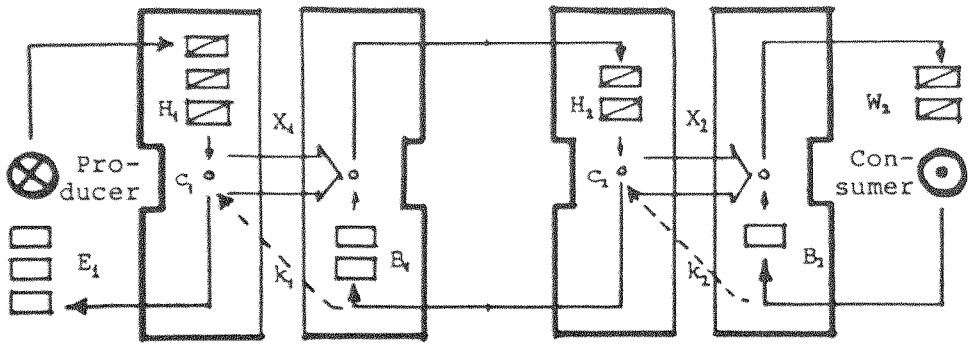


Figure 3: Serial flow control.

$$b_1=e_2 \quad \text{and} \quad w_1=h_2$$

The operations are derived from those of the simple system by indexing of variables. The assertions are (i having values 1 and 2):

$$c_i+k_i=b_i-x_i \tag{2.1}$$

$$e_i+h_i=\tilde{e}_i \tag{2.2}$$

$$w_i+b_i=\tilde{b}_i \tag{2.3}$$

It is easily verified that the assertions are valid. The proof of $x_i \leq b_i$ is similar to that of the simple scheme. The conclusion is that concatenation of flow control is a meaningful operation, which preserve the validity.

6. Multiplexing.

When several data streams share a channel, but are otherwise independent, the data streams are said to be multiplexed (see also [3]). The outer transmission systems (for the streams) use that of the inner (the channel) for the actual transfer (see figure 4). As both the channel and the streams need flow control, this leads to nested flow control. The combined producers and stream transmitters act as one producer for the channel, and similarily the combined consumers and receivers act as one consumer.

In the following the index zero signifies the channel variables, and a positive index the corresponding stream. The notation is otherwise unchanged.

The scheme has been simplified somewhat with respect to the queues. It is assumed that no copying is needed at the stream/channel interface. As a consequence E_0 and B_i are omitted. Similarily the buffers in B_i are non-existent, but b_i is the number of buffers in B_0 that stream i considers its

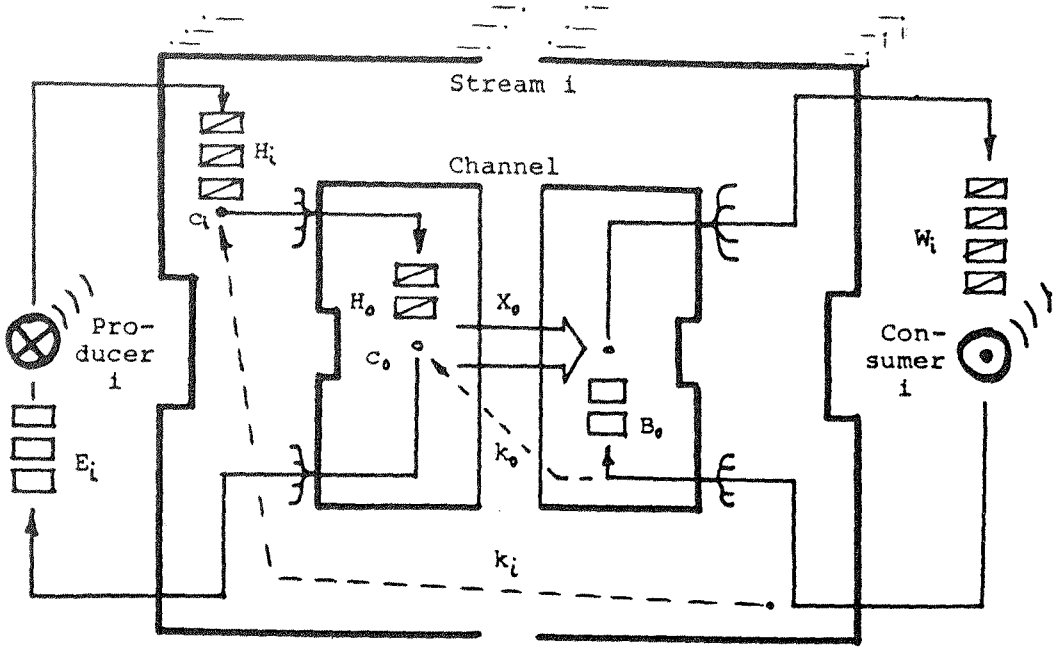


Figure 4: Multiplexing, nested flow control.

share. Allocation of buffers is a separate issue.

The operations of the channel and the streams are as follows (note that some actions are coupled together):

Operations of the channel.

T1: PUT FULL
(* assert nil *)
h_0:= h_0+1; (* enqueue *)

T2: DELIVER FULL
(* assert h_0>0 *)
h_0:= h_0-1; (* dequeue *)
c_0:= c_0-1;
x_0:= x_0+1; (* SEND DATA *)
call $STREAM_i$.T4; (* deliver empty *)

T3: CREDIT ARRIVED
(* assert k_0>0 *)
k_0:= k_0-1;
c_0:= c_0+1;

R1: FULL ARRIVED
(* assert x_0>0 *)
x_0:= x_0-1;
b_0:= b_0-1; (* dequeue *)
(* call $STREAM_1$.R1 *)

R2: PUT EMPTY
(* assert nil *)
b_0:= b_0+1; (* enqueue *)
k_0:= k_0+1; (* SEND CREDIT *)

Operations for stream i.

```
T1: PUT FULL
(* assert e_i>0 *)
    e_i := e_i-1;
    h_i := h_i+1;          (* enqueue *)

T2: DELIVER FULL
(* assert h_i>0 *)
    h_i := h_i-1;          (* dequeue *)
    c_i := c_i-1;
    x_i := x_i+1;          (* call CHANNEL.T1 *)

T3: CREDIT ARRIVED
(* assert k_i>0 *)
    k_i := k_i-1;
    c_i := c_i+1;

T4: EMPTY ARRIVED
(* assert nil *)
    e_i := e_i+1;       (* deliver *)

R1: FULL ARRIVED
(* assert x_i>0 *)
    x_i := x_i-1;
    b_i := b_i-1;
    w_i := w_i+1;       (* deliver full *)

R2: PUT EMPTY
(* assert w_i>0 *)
    w_i := w_i-1;
    f_i := f_i+1;
    call CHANNEL.R2;
    k_i := k_i+1;       (* SEND CREDIT *)
```

The invariants are (note i>0 unless otherwise stated):

$$c_i+k_i = b_i-x_i \quad \text{(all } i\text{)} \tag{3.1}$$

$$\Sigma(e_i+h_i)+h_0 = \Sigma\tilde{e}_i \tag{3.2}$$

$$\Sigma w_i+b_0 = \tilde{b}_0 \tag{3.3}$$

Multiplexing is consistent

$$\Sigma x_i=h_0+x_0 \tag{3.4}$$

It is easily verified that the assertions are true under the operations. If $\Sigma\tilde{b}_i \leq \tilde{b}_0$ then follows (as $\Sigma b_i \leq b_0$ is invariant):

$$b_i - x_i = c_i + k_i \geq 0$$

which implies $b_i \geq x_i$

which implies $b_0 \geq \Sigma b_i \geq \Sigma x_i = h_0 + x_0 \geq x_0$

As this does not involve the assertions for c_0 and k_0, it shows that independent of the channel flow control mechanism, the channel will never overflow if the stream flow control is conservative enough. Thus interesting enough, we have shown that the channel flow control can be omitted. This may be taken as an argument against indiscriminate use of nested or layered systems.

7. Conclusion.

It has been shown that a simple scheme consisting of a few time-independent operations and their associated variables and assertions may be used to validate the essential part of flow control protocols. Furthermore, it is possible to obtain results for composite systems using the same method.

The method combines well with the error control validation of [1], to give a total method for verifying end-to-end protocols.

Other methods for specification and verification [4] like finite-state automata or Petri nets are used for specific protocols to prove e.g. freedom from deadlock, completeness, or stability, or to verify that the design meets its specification. The approach in this paper is not quite the same, as it is more limted with respect to properties, and more general in scope, as the properties are shown for classes of conceivable protocols.

8. Acknowledgements.

I wish to thank Jens Hammerum for the original idea, which appeared in his Master's thesis in disguise, and A.P.Ravn for much advice on the presentation of the ideas.

9. Litterature.

[1] D.E.Knuth: Verification of link-level protocols.
BIT 21,1 (1981)

[2] S.Krogdahl: Verification of a class of link-level protocols.
BIT 18,4 (1978)

[3] ISO/IS 7498
Open Systems Interconnection
(1983)

[4] C.A.Sunshine: Formal Modelling of Communication Protocols.
USC/Information Sciences Institute, RR-81-89 (March 1981)