A THEORY FOR PROTOCOL VALIDATION

Gerard J. Holzmann

Delft University of Technology
Delft, The Netherlands

ABSTRACT

This paper introduces a simple algebra for the validation of communication protocols in message passing systems. The behavior of each process participating in a communication is first modeled in a finite state machine. The symbol sequences that can be accepted by these machines are then expressed in 'protocol expressions,' which are defined as regular expressions extended with two new operators: division and multiplication. The interactions of the machines can be analyzed by combining protocol expressions via multiplication and algebraically manipulating the terms.
The method allows for an arbitrary number of processes to participate in an interaction. In many cases an analysis can be performed manually, in other cases the analysis can be automated. The method has been applied to a number of realistic protocols with up to seven interacting processes.

An automated analyzer was written in the language C. The execution time of the automated analysis is in most cases limited to a few minutes of CPU time on a PDP 11/70 computer.

Index Terms: Regular expressions, distributed systems, message passing, deadlock detection, verification, validation algebra, protocol analysis.

IEEE Trans. on Computers, Aug. 1982, Vol C-31, No 8, pp. 730-738.

1. Introduction

A protocol is a set of procedures that allows computers in a distributed system to establish, maintain, and complete communications through the network that connects them.

A well-designed protocol must meet a number of demands. It must, for instance, allow for efficient data transfer under normal circumstances. It should detect and respond adequately to occasional errors, like bit distortion or message loss. And, finally, it should be able to recover from more radical failures, like computer crashes and line disruptions [2].

Finite state automata models have been used frequently in protocol studies (for overviews see [4,6]). By defining the interaction of the finite state machines properly one can perform an exhaustive state space exploration for the combined machine, and thus trace its undesirable features. A standard problem with the analysis methods of this type is the inherent danger of a 'state space explosion.' To solve this problem various 'hybrid' techniques have been suggested, in which a partial state space exploration is combined with program proof methods based on Floyd-Hoare logic. A disadvantage of the latter, however, is that it cannot readily be automated.

Here another approach is examined. We will describe the behavior of the finite state machines with an extended type of regular expression, termed 'protocol expression.' We then define a cross product on protocol expressions and introduce some algebraic reduction and equivalence rules. In some cases a state space exploration can now be avoided completely by performing a simple manual algebraic analysis of the terms in the cross product. In other cases much of the costly state space construction can be avoided by

using an automated cross product evaluator that can establish the correctness of a protocol for a number of standard correctness criteria.

Unlike the earlier methods for protocol validation founded on finite state machine theory [4,13], we explicitly try to exploit the algebraic nature of the description we've chosen.

Several other forms of extended regular expressions have been used before in the study of systems allowing concurrency, see for instance Bochmann [2,3], Ogden & Riddle [8], Shaw [9], and Schindler [10]. Closely related are also Milner's calculus for communicating systems [7] and Teng & Liu's transmission grammar [12]. In their work, Bochmann [2,3] and Schindler [10] have already briefly sketched the expected merits of a validation [2,3] or specification method [10] based on regular expressions. Ogden & Riddle [8] and Shaw [9], and also Milner [7] and Teng & Liu [12], present well-elaborated models for the validation of concurrent systems. Their validation methods, though, do not apply to the type of message passing systems we will consider here (esp. with respect to message queueing, see below).

We will develop a concise validation algebra for message passing systems with fifo message queueing. We will not assume the existence of a 'shared memory' to which all processes have access, nor do we impose a 'rendez-vous' requirement on the synchronization (as in Hoare's CSP). Timeout, message loss, and message distortion can all be modeled straightforwardly within the algebra.
The analysis method introduced here was first described in [5].

In section 2 we make explicit some assumptions about the type of systems to which the algebra applies, and we discuss a simple example of such a system. In section 3 we discuss the interpretation of two special symbols in the algebra and indicate how they can be used to model deadlock and message loss. In sections 4 to 6 we develop the formalism for the extended expressions and the special operators we will use. Section 7 gives examples of the manual analysis of two moderately complicated protocols. In section 8 we discuss the possibilities for the automated analysis of more complex protocols. In section 9 we discuss notation, protocol structure, and the expected time complexity of the automated analysis. Section 10 summarizes the results. The appendix reviews the standard definition of regular expressions and summarizes the definition of protocol expressions.

2. Preliminaries

The systems we will study consist of an arbitrary number of concurrently operating sequential processes that interact via the exchange of messages. Every process owns precisely one 'mailbox' of a finite size. A process receives messages via its mailbox, and can send messages to any mailbox in the system, including its own. Only two types of operations may be performed on the mailboxes: send and receive. The actual syntax of the send and receive operations is of little interest to us here. The effect of their execution, however, is assumed to be as follows.

If the mailbox assigned to a process is not empty, the receive operation will return its oldest message. If, however, the mailbox is empty, the process executing the receive operation is delayed until the first message arrives. The process may specify a maximum waiting time. If the mailbox is still empty when that time limit expires, a system demon will place a 'timeout message' in the empty mailbox, thus forcing a restart of the waiting process.

The send operation will append a message to the mailbox addressed. To simplify the analysis we will assume that the send operation will not terminate until the message has been delivered (i.e. appended to the mailbox). The capacity of the mailboxes is not bounded a priori, but is assumed to be finite. A send operation that addresses a full mailbox does terminate, but 'fails' (i.e. it returns an error status, and the message sent is lost).

All operations on the same mailbox exclude each other in time. Note that this implies that a timeout message, if present in a mailbox, is necessarily the oldest message. (It is only appended when the mailbox is empty. It can only be followed by, never preceded by, other messages.)

Although the receive operation allows us to specify explicit timing constraints, we will require the analysis to be independent of timing considerations. Only the fact that a timeout count has been specified, and thus a timeout made possible, makes a difference in the analysis. The value of the timeout count itself is irrelevant to the analysis.

The analysis method will allow us to model both ideal and non-ideal transmission channels. One can model the possibility that a message is lost on the channel, transformed into another message, or mutilated beyond recognition. But, here as well, we are only concerned with possibilities, not with probabilities.

We will abstract from all actions that are not related to message passing. Following Zafiropulo [13] we name the occurrence of such actions 'non-events.'

Figure 1 represents a simple connect protocol discussed in [13], for our purposes extended with a deliberate error. A receive operation is represented by a triangle. The lines branching out of the triangle are labeled with the names of the messages that the protocol designer expects to receive in that state. In the protocol of Figure 1 each receive operation is followed by a switch on the message received. The reception of an unexpected message is considered to be the result of a design error (incomplete specification). Each message is represented by a different symbol.

Fractions

To formalize the behaviors we will use fractions. Messages to be received are placed in the numerator of a fraction; messages to be sent are placed in the denominator. The timeout message is represented by the symbol T (tau). The plus, dot, and star operators have a special meaning, to be explained below. In this formalism we can represent the behavior of the processes in Figure 1 by the following two expressions:

```
((  1      1  ) 1)*
||a.- + T1.-.a|.-|                    (1)
{{  b      b  } c}

(  1        1  )*
|b.-.c + T2.-.b|                      (2)
{  a        a  }
```

In these expressions the dot operator represents a time ordering, the plus represents a choice between two possible (sub)behaviors, and the Kleene star indicates a repetition of zero or more times.

The star, the dot, and the plus are the usual operators in a regular expression; the division operator, however, is new.

The combined behavior of the two processes (fig. 1) is given by their cross product:

```
((  1      1  ) 1)*   (  1        1  )*
||a.- + T1.-.a|.-|  X |b.-.c + T2.-.b|          (3)
{{  b      b  } c}    {  a        a  }
```

The precedence of the operators, in descending order, is now: star, division, dot, plus, cross.

We have extended the regular expressions with fractions to capture the send operations, and with multiplication to capture the interactions. (For a discussion of this notation see section 9.) These extended expressions are named 'protocol expressions.'

3. Special Symbols

The symbols 1 and ∅ will be used with a special interpretation. To understand their meaning, it should be observed that every regular expression, and every protocol expression, represents a 'set' of possible execution histories.

In the behavior specified by expression (1) this set contains two elements: (a.(1/b).(1/c)) and (T1.(1/b).a.(1/c)). Every element of the set can be written in a canonical form as a string (i.e. a dot product) of symbols and fractions. The null string is represented by the special symbol 1. The null string gives us a way to model non-events, as defined above. Note also that the null string makes it possible to model the 'loss' of a message on the transmission channel.

An empty set of execution histories is represented by the special symbol ∅. That is, the ∅ indicates a true dead end in an execution sequence. It will give us a way to describe and detect deadlock.

Message Loss

To represent the loss of message m we can simply replace every occurrence of (1/m) in the protocol expressions by (1+(1/m)). Alternatively, we can model the transmission channel itself. Every message m is then addressed to the logical channel connecting the two processes instead of directly to the addressee. We indicate this by replacing m with m′ in the following expressions. The channel transfers the message m′ to its proper destination (using symbol m again).

An ideal channel transfers the messages without change. It can be modeled as:

```
(     1 )*
|m′.  - |    for every message m′ expected
{     m }
```

If we include deletion errors and mutation errors this becomes:

```
(   ( 1   1    ))*
|m′.| - + - + 1||    for every message m′ expected
{   { m   x    }}
```

where x represents a message mutilated beyond recognition. Below we will see that the phrase ‘for every message m′ expected’ can also be formalized with the aid of the concept of a ‘mailbox sort’ (section 5). Explicitly specifying the sets of messages expected by every process can help us to trace incompleteness and inconsistencies in protocol specifications.

Interaction

The cross operator, introduced in equation (3), is derived from the basic ‘shuffle’ operator [e.g.8,9,12]. The shuffle of two dot products will give us all possible interleavings of the symbols in each product:

```
(a.b) D (x.y) = a.b.x.y + a.x.b.y + a.x.y.b +
                x.a.b.y + x.a.y.b + x.y.a.b
```

where D represents the shuffle operator. Clearly, not all of these interleavings really represent feasible interactions in a real message passing system. The cross product differs from the shuffle operator in precisely this way: it will generate only feasible interleavings (i.e. possible interactions). In the present case, where all symbols represent ‘received’ messages, the cross product will yield:

```
(a.b) X (x.y) = ∅
```

that is, ‘none’ of the interleavings can be realized because both dot products start by claiming a message that was never sent. (Note that this behavior specifies no send actions at all.)

As a slightly more complicated example, consider the following cross product:

```
( 1 )n  (    ( 1 ))*   (    ( 1 ))*
| - | . |p . | - ||  X |c . | - || (4)
{ c }   {    { c }}    {    { p }}
```

It formalizes the interaction between a producer and a consumer process, each with a buffer of size n. The producer (the right-hand side of the cross product) sends messages of type p to the consumer. In order not to overflow the consumer’s buffer the producer delays the transmissions until an appropriate ‘go-ahead’ message c is received. The consumer triggers the producer’s output by sending a sequence of n initial c messages before engaging itself in the conversation. We would, of course, like to be able to prove that the buffers cannot overflow when this scheme is enforced, just as we would like to be able to find the design error in the example protocol of Figure 1. We will return to these problems in section 7.

4. Rules

Protocol expressions have some interesting properties. Here are some rewriting rules for formalized behaviors. Each lower-case character represents a non-extended regular expression (see appendix).

$$\text{I.01} \qquad a = \frac{a}{1}$$

$$\text{I.02} \qquad a \cdot \frac{1}{b} = \frac{a}{b}$$

$$\text{I.03} \qquad \frac{1}{a.b} = \frac{1}{a} \cdot \frac{1}{b}$$

$$\text{I.04} \qquad \frac{a+b}{c} = \frac{a}{c} + \frac{b}{c}$$

$$\text{I.05} \qquad \frac{a}{b+c} = \frac{a}{b} + \frac{a}{c}$$

The rules are adopted here as axioms. Note that the resemblence between this algebra and the more familiar version taught in high school is only partial (cf. I.05). Note also that the interpretation of the dot as a time ordering implies (with I.02) that

$$\frac{1}{b} \cdot a \neq \frac{a}{b}$$

For an analysis, the simplest case to deal with is clearly the behavior of a single process that appends messages to and deletes them from its own mailbox, without interacting with any other process. In that case one can conceive of the following reduction rule:

$$\frac{1}{a} \cdot a = 1 \qquad (5)$$

which states that the net result to a mailbox of appending and deleting a message a is nil. However, the ordering property of the mailbox complicates a generalization of this rule. In general, if a and b are addressed to the same mailbox (cf. section 4):

$$\frac{1}{a} \cdot \frac{1}{b} \cdot b \neq \frac{1}{a} \cdot 1$$

simply because the process claims message b when only a message a is available. This implies:

$$\frac{1}{a} \cdot \frac{1}{b} \cdot b = \frac{1}{a} \cdot \frac{1}{b} \cdot \varnothing$$

which means that only the first two operations in the dot product to the left of the equal sign can actually be executed. After the execution of these first two operations the system will come to a dead stop (formalized by the ∅ symbol) because the only option for a continuation would violate a ‘soundness rule.’

In a later section we will see that both a soundness and a reduction rule can be formulated concisely with a few extra definitions (section 6).

5. The Cross Operator

The cross operator can obviously be defined to take any positive number of operands. We can therefore also write a cross product as a function that takes individual process behaviors as arguments:

X(P1,P2, ... ,Pn)

where P1, ..., Pn are protocol expressions.

In the simplest case the cross can be applied to the single process that appends messages to and deletes them from its own mailbox.

For instance:

```
 (1 1   1          )     1 1   1
X|-.-.a.-.b.(d + e)|  =  -.-.a.-.b.∅
 {a b   c          }     a b   c
```

Neither the d message nor the e message can be received, because the first message in the mailbox at that moment is a c. The cross operator thus implements the generalized reduction rule discussed above. The generalization is fairly simple when we use the formalized notion of soundness, defined in the next section.

If more than one process is considered the cross evaluator must distinguish between the mailboxes to verify that messages are received in the order in which they were sent. (A fifo order is to be maintained for all messages addressed to the same mailbox. Two messages addressed to two distinct mailboxes, however, need not be received in the order in which they were sent.)

The cross operator then has, as parameters, the sets of messages that can be appended to each mailbox in the system: one per process. We call these sets the 'mailbox sorts.'

To simplify the discussion we will assume that every message used below uniquely identifies both its sender and its receiver. That is: if we say that process A sends message m to B it is implied that no other process in the system considered will use the same message name m. This assumption guarantees that mailbox sorts are always disjoint.

In practice, this 'disambiguation' of message names is easily accomplished by extending every message name with the names of the sending and receiving processes.

All events in the process behaviors concerning messages that are not in any mailbox sort specified are treated as non-events. These messages are not subject to the soundness rules that apply to the others. This convention is useful when we study message exchanges between processes that are part of a larger system. We can then concentrate on messages exchanged between a subset of the processes in the system, and abstract from the interactions with the environment. Messages that are outside the parameter sets of the cross product (outside the mailbox sorts) are called 'free' messages.

The following example will illustrate the use of mailbox sorts and free messages. Consider a system of three processes named P, Q, and R. The three mailbox sorts are specified as:

S(P) = {a, c}, S(Q) = {b}, and S(R) = {d}.

Let the behavior of the processes P, Q, and R be given by:

```
a          1              d
- . c ,   --- . b ,  and  - , respectively.
b         d.a             c
```

Instead of considering all possible interactions in one single cross product, we can single out a subsystem of two processes and abstract from the third process. Let us, for instance, take the cross product of P and Q, with c and d as free messages. The mailbox sorts S(P) and S(Q) for the subsystem, with a set of free messages F, becomes:

S(P) = {a}, S(Q) = {b}, F = {c, d}

For (P X Q) we find:

```
( a    )   (  1     )      1    a             1
| - . c| X | --- . b|  =  --- . - . b . c  =  - . c
{ b    }   { d.a    }     d.a   b             d
```

The validity of the reduction implied by the second equal sign can be argued with the aid of the 'soundness' concept to be discussed in the next section. In this case, we can cross out the a's and the b's only because of the fact that symbol d is neither in S(P) nor in S(Q), and therefore the simple reduction rule (5) from section 4 applies.

It is now easy to calculate (P X Q X R) as ((P X Q) X R):

```
( 1    )   ( d )     1   1
| - . c| X | - |  =  -.d.-.c  =  1
{ d    }   { c }     d   c
```

The fact that the cross product reduces to 1 proves that every message sent is eventually received. It also trivially shows that the system P, Q, R is completely specified (there are no 'unexpected' receptions) and cannot deadlock (there are no dead ends).

Reversing the order of the symbols in the first term of the product above gives us the trivial deadlock:

```
( c )   ( d )     (    1 )   (    1 )
| - | X | - |  =  |c . - | X |d . - |  =  ∅
{ d }   { c }     {    d }   {    c }
```

We will refrain from specifying the mailbox sorts for every cross product that occurs below. In most cases they can trivially be derived from the context. (The mailbox sort of a process is implicitly defined to be the set of all messages that occur in the numerators of the corresponding protocol expression.)

Here are some useful identities for the cross operator, again adopted as axioms:

```
I.06     X(A, 1)  =  X(A)
I.07     X(A, B)  =  X(B, A)
I.08     X(A, B, C)  =  X( X(A, B), C)
I.09     X(A, (B+C)) =  X(A, B) + X(A, C)
```

6. Soundness

We have used the term 'soundness' informally until now. When do we call a message exchange sound? Before we answer that, let us introduce two new symbols that will prove especially helpful here.

Let M_i be the 'mailbox history' of process i. M_i is a string of symbols, where each symbol represents a message that was sent to the mailbox of process i, and where the order of the symbols corresponds to the order in which the messages were appended to the mailbox.

Secondly, let C_i be the 'claiming record' of process i. C_i is also a string of symbols, but in this string each symbol represents a message that was claimed to have been received by process i. The order of the symbols again indicates the order in which the corresponding messages were claimed.

Now, for every execution sequence E that is in the shuffle of the behaviors of a set of interacting processes we can find the corresponding mailbox histories and claiming records for each process. Call p(S) the set of all prefixes of string S, including ∅ and S itself. (A prefix of a string is obtained by deleting its last n elements, where n is any number between ∅ and the lenght of the string.)

A message exchange is called 'sound' if for every prefix E_j of execution sequence E every claiming record $C^j{}_i$ is a prefix of the corresponding mailbox history $M^j{}_i$:

$$A(i,j)\ \{\ E_j\ E\ p(E) \rightarrow C^j{}_i\ E^j{}_i\ p(M)\ \}$$

In the absence of timeouts this definition would suffice. For timeout messages, though, the situation is different, as these messages are not sent by a user-defined process as part of the user-defined protocol, but by an unknown system demon (section 2). We do not want to be bothered with a formal specification of the demon's behavior, for obvious reasons. The system demon will have to remain anonymous, and as a

result we disallow send operations on timeout messages in the protocol specifications. Instead, we will define a special rule for the soundness of a sequence with timeout claims (i.e. receptions of timeout messages). The rule is quite simple: a timeout message can only be received if it was specified as an option in the pending receive operation, and if the mailbox of the waiting process is empty when the timeout message is claimed.

To formalize this, let us define a 'hiding' operation "\" (the terminology stems from [7]). With the symbolism "S\T" we then mean: the string "S" from which all the "T" messages (timeouts) have been deleted.

Also, let us define that 'tail (C_i)' represents the last element in string C_i. We can now improve our definition of soundness as follows.

A message exchange is called 'sound' if and only if:

$$A(i,j)\ \{\ E_j\ E\ p(E)\ \rightarrow\ C_i \backslash T\ E\ p(M_i^j)$$

$$\text{and}\quad tail\ (C_i)\ = T\ \rightarrow\ C_i \backslash T = M_i^j\ \}$$

The cross operator enforces the soundness rule on the full shuffle of the individual process behaviors defined in protocol expressions.

Reductions

The reduction of a cross product, intuitively applied in sections 4 and 5, can now be formalized by adopting the convention that, in an analysis, any sound message exchange may be replaced by the nul string 1. Note that the soundness rule defined above applies equally well to (partially) reduced message exchanges.

Equivalence

Even with the soundness rule the number of terms generated by the cross product tends to be extremely large for systems of a practical size. We therefore need some notion of equivalence that will allow us to describe large classes of behavior by a single abstraction. Fortunately, we can give an adequate definition of equivalence with the same tools as used for the definition of soundness.

Two execution sequences E_1 and E_2 are called 'equivalent' if and only if the mailbox histories and the claiming records for the two sequences are identical for every process considered:

$$A(i)(M^1{}_i = M^2{}_i \text{ and } C^1{}_i = C^2{}_i)$$

Residuals

It is often useful to study also the messages left behind in the mailboxes after a series of message exchanges. In general, the completion of a 'conversation' between two processes (often after a single cycle through the protocol; cf. section 8) is expected to return the system to a state where all the mailboxes are empty. Residual messages may well, and generally do, create problems in subsequent execution sequences. To be able to study this aspect of the interactions we define the notion of the residual.

The 'residual' for process i of a message exchange is the string R_i which is defined as the difference between M_i and C_i (the string that is obtained by deleting prefix C_i from string M_i, or in formal jargon: the derivative of M_i with respect to C_i).

7. Manual Analysis

In section 3 we discussed the following formalization of a producer-consumer interaction (expression (4)):

```
( c )*   ( 1 )n  ( p )*
| - |  X | - | . | - |                    (4′)
{ p }    { c }   { c }
```

We would like to be able to prove that the two finite mailboxes in this system cannot overflow. The proof given below shows that some protocols can be analyzed by hand, without elaborating the full cross product at all. We merely assume that the cross operator enforces the soundness rule.

We can represent an arbitrary partial behavior of the system by an expression with some variables:

```
     s           u
( c )   ( 1   p )
| - | X | - . - |                          (6)
  t       n   v
{ p }   { c   c }
```

Clearly, the variables s, t, u, v, and the constant n in expression (6) are related. We have, for instance:

$$s \geq t \tag{7}$$

simply because the producer is assumed to cycle through its protocol starting with a receive operation. Similarly:

$$u \geq v \tag{8}$$

Also, as the consumer cannot consume more p messages than the producer sends, we have:

$$t \geq u \tag{9}$$

and finally:

$$n + v \geq s \tag{10}$$

as the producer can never claim more c messages than the consumer sends.

These four relations are sufficient to prove the observance of the invariant condition:

$$t - u \leq n \tag{11}$$

which implies that the consumer's mailbox cannot overflow, and

$$n + v - s \leq n \text{ or } v \leq s \tag{12}$$

implying that the producer's mailbox cannot overflow.

From equations (7) and (10) it follows that

$$n + v \geq t \tag{13}$$

and from (8) and (13) we can derive

$$n + u \geq t \tag{14}$$

or

$$n \geq t - u \tag{15}$$

which proves condition (11). Similarly, from (7) and (9) it follows that

$$s \geq u \tag{16}$$

and from (8) and (16) we can derive

$$s \geq v \tag{17}$$

which proves also condition (12) (Q.E.D.).

The example protocol illustrated in figure 1 can also readily be analyzed by hand. We will illustrate a procedure that is based on a series of 'single cycle analyses.' Each process has two options for its behavior,

which gives us four possible combinations in the cross product.

```
(   1       1   1 )   (  1        1  )
|a.--- + T1.-.a.- | X |b.-.c + T2.-.b|     (18)
{  b.c      b   c }   {  a        a  }

 = ∅                                  ...term 1.1 x term 2.1

       1   1 (  1   1  )
  + T2.-.a.-.|b.- + -.b|               ...term 1.1 x term 2.2
       a   b {  c   c  }

       1   1   1
  + T1.-.b.-.a.-.c                     ...term 1.2 x term 2.1
       b   a   c

  ((   1   1)   (   1  ))
+ ||T1.-.a.-| X |T2.-.b||              ...term 1.2 x term 2.2
  {{   b   c}   {   a  }}

     1         1
 = ∅ + - + 1 + -
     c         c
```

Each term on the right-hand side of the equality can be linked to one particular combination of the behavior options specified on the left side. For brevity we didn't elaborate all possible interleavings of the last term, but instead used the abbreviation allowed by the cross operator. The first term cancels against the other three (appendix, identity C.2). There are no deadlock terms in the product, but the second and the fourth term indicate that a residual message `c` may be left in the mailbox of the second process when this first cycle is completed. The design error in the protocol is found if we repeat the test while placing this residual message as an initial message in that mailbox. The new cross product then yields a term:

```
1   (   1   1 )   (   1 )
- . |T1.-.a.- | X |T2.- | . ∅
c   {   b   c }   {   a }
```

After consuming the `T2` message and sending `a`, the second process expects to find message `b`, but instead receives residual message `c` and blocks. The claim for a `b` message is replaced by the symbol ∅ above, to indicate that this continuation violates the soundness rule. The first process, meanwhile, completes its cycle, but cannot fail to block in a subsequent cycle for lack of new `a` messages from the second process.

Comments on the use of the single cycle analysis are included in section 9.

8. Automated Analysis

A cross evaluator has been implemented for process behaviors that can be expressed in the language to which the algebra is restricted. The language of the protocol expressions can be considered an indeterministic language, much like Hoare's CSP, but without the rendez-vous requirement, and without variables. Many protocols can be expressed in this restricted language. (Exceptions are, of course, protocols that rely heavily on the use of variables.)

The cross evaluator extracts all 'sound' message exchanges from the shuffle of the protocol expressions. It exploits the identities I.06 to I.09, and uses the notion of equivalence to minimize its work.

The most interesting results of the evaluator are of course the execution sequences which end in a mailbox deadlock, as formalized by the symbol ∅. These 'deadlocks' really represent cases where the reception of a message violates the specification. A milder term for this type of error is therefore 'incomplete specification.'

A direct consequence of the abstraction we make from the absolute values of the timeout counts in receive

operations is, of course, that we also trace errors that can only occur under the most unfavorable choice one could possibly make for the timeout counts (the 'micro-second timeout'). Therefore, the majority of the deadlocking execution sequences found by the cross evaluator usually represent sequences that cannot occur in a real system if the timeouts are chosen properly.

To avoid generating infinite execution sequences, the cross evaluator starts by considering only a single cycle through a protocol for all participating processes. From the results we then examine the deadlocking sequences, and eliminate those caused by 'micro-second timeouts' (by hand). Then we extract the residuals of the execution sequences of the first cycle (by program), and repeat the analysis while placing the residual messages as 'initial' messages in the mailboxes. In a well-defined protocol either there are no residuals, or the residuals are cleared before a second cycle is initiated. In many real protocols, however, this rule is not observed. The errors that may result are traced by the analyzer.

An analysis is typically done in three steps. First, the formalized protocol, specified in an intermediate language that matches the descriptive power of the protocol expressions, is checked on syntax and then translated into protocol expressions by a program named 'prc' (for 'protocol compiler'). Then, the cross evaluator is used to trace deadlocking execution sequences implied by the protocol specification. This program was named 'pan' ('protocol analyzer'). Thirdly, a program named 'orphans' is used to find the residuals of the message exchanges. If residuals are found the analysis can be repeated, starting at step 2.

Two of the protocols to which the method has been applied were fairly complex data-switch protocols that specify interactions among respectively five and seven different processes and use more than 30 different types of messages. The execution times for analyzing the larger protocols is still quite reasonable: ranging from mere seconds to several minutes of CPU-time. Still, potentially infinite cycles in a message exchange are problematic. In its present implementation the cross evaluator cannot recognize infinite cycles and represent them in a symbolic way in the output. This, however, seems to be more a limitation of the implementation than of the method. Note, for instance, that simple cycles like:

```
( 1 )*      *
| - |  X |a|
{ a }
```

may, by applying the equivalence rules, be represented as:

```
( 1     )*  ( 1 )*
| - . a | . | - |
{ a     }   { a }
```

The disadvantage of working with manual formalizations of an implemented protocol is of course that an unintended idealization may cause the evaluator to miss real design errors in the implemented version of the protocol. To avoid this we plan to automate also the formalization process.

9. Discussion

In this section we will discuss some remaining issues of the validation algebra introduced here. First, we will consider the advantages and disadvantages of the notation we've chosen for the protocol expressions. We will then consider some potential limitations of the automated analysis process (single cycle analysis, and time complexity).

Notation

The choice for a fractional notation in the protocol expressions was made deliberately, but is of course by no means an obvious one. (Bochmann [2] uses an underscore for send actions; Ogden and Riddle [8] use symbols with overscore; Shaw [9] uses the Greek symbols omega and sigma for send and receive, and Schindler [10] uses the prefixes r_ and s_ for message names.)

Our choice is motivated by the intuition that sending and receiving are each others' inverses. There is a pleasing, though only a partial, duality between the division operator and the cross. Note, for instance, that:

```
1          1            1
-  X  a =  - . a  +  a . -  =  1 + 0  = 1
a          a            a
```

Another advantage of the fractional notation is that it makes very concise, yet unambiguous, protocol specifications possible, when the rewriting rules (I.01 to I.05) are applied. As an example of this conciseness, consider the following specification of the NPL alternating bit protocol [1]. This protocol, partially specified with regular expressions in [2], can be specified concisely and completely with the fractional notation as follows:

```
           ( 1   ( n+x )*  m   ( m+x )*   )*
A process: | - . | --- | . - . | --- | . n|
           { a   {  a  }   b   {  b  }    }

           (( b+x )*  a   ( a+x )*  b )*
B process: || --- | . - . | --- | . - |
           {{  n  }   m   {  m  }   n }
```

where process A alternates between an a and a b message (frame), and similarly, process B alternates between m and n. An x denotes a message (frame) mutilated beyond recognition.

It is trivial to verify the correct working of the protocol for an ideal channel and for a totally unreliable channel (a channel that changes every message into an x).

For instance, for the ideal channel we find:

```
           ( 1   a   m   b     )*
(A X B) = | - . - . - . - . n |
           { a   m   b   n     }
```

which symbolizes effective data transfer. Similarly, for the worst case unreliable channel we find:

```
           1   ( a   x   n )*
(A X B) = - . | - . - . - |
           a   { x   n   a }
```

where the unreliable channel contributes the term a/x.

On the other hand, a disadvantage is that the fractional notation forces us to represent send actions with a binary operator, where a unary operator would suffice.

Block Structure

For the moment we have limited ourselves to single cycle analysis, notably without firm proof that a well-chosen series of such tests (section 7) would prove the absence of errors for ‘any’ number of cycles. As noted, the message passing sequences may leave residuals in the mailboxes that can lead to problems in subsequent cycles. There are still many reasons to prefer a single cycle analysis over an exhaustive analysis. Clearly, a single cycle analysis is simpler and faster than an exhaustive one, and when some precautions are taken in the protocol itself a full analysis may even be unnecessary.

For most interactions where protocols are required, one can structure the message exchanges into logical blocks which can be labeled ‘conversations.’ The possibility of an overlap of two conversations is the main reason for the insufficiency of the single cycle analysis. Now, either one can design the protocol in such a way that two conversations never overlap [13], or one can tag a ‘conversation number’ to every message exchanged (named ‘incarnation number’ in [11]). The numbers will ensure that two distinct conversations can never interfere, as long as the numbers are (sufficiently) unique. Any message received that has a number differing from the current conversation number is then simply ignored.

Let A and B be protocol expressions that specify the behavior of two participants in a protocol. Protocols that guarantee ‘conversational independence’ (we call them ‘block structured protocols’) have the following property:

$$A^* \times B^* = (A \times B)^*$$

The problem of establishing and maintaining agreement about the legality of conversation numbers now becomes a new problem to be solved by the protocol, and the verification of its solution a new problem for the protocol analysis technique.

Complexity

The worst case time complexity of the cross evaluator is clearly exponential in the number of interacting processes N. The worst case to analyze would be a system in which all processes only perform send operations, all directed to the same mailbox. Every possible shuffle of the message sequences then has to be taken into account due to the fifo ordering discipline imposed by the mailbox.

Fortunately, though, for realistic protocols the cross evaluator only has to consider a very small subset of the full shuffle of process behaviors. Firstly, it should be noted that the purpose of the protocol itself is to limit the amount of indeterminism in the combined behavior of the participating processes. Secondly, the equivalence rules allow us to ignore the larger part of the remaining theoretically possible shufflings of the behaviors. The expected time complexity of the cross evaluator, therefore, is not exponential in N, but only exponential in a factor close to 1.

10. Conclusion

We have formulated a concise verification algebra for message passing systems which can be used for both manual and automated analysis. Verification algebras based on regular expressions have, to our knowledge, not been applied to mailbox systems before.

With the algebra one can effectively analyze complex message passing protocols for mailbox systems. The method discussed is quite attractive from a theoretical point of view (it has an appealing simplicity) but, more importantly perhaps, was also shown to be of practical value by the implementation of the cross evaluator program.

With the automated analyzer we can check a protocol on the observance of a small set of fairly standard correctness criteria (no deadlocks, no residual messages, no potential timeout problems, completeness). It is expected that several other properties (reachability, connection establishment) can be automated along the same lines. More specific properties, though, (proper data transfer, flow control, message sequencing) still have to be analyzed by hand (cf. section 6) guided by human intuition.

Acknowledgement

I am grateful to Al Aho, Greg Chesson, Doug McIlroy, Lee McMahon, Dave McQueen, and Ravi Sethi for the considerable time they spent discussing this theory with me. Rob Pike never stopped surprising me with his rapid solutions to subtle implementation problems. I thank him for his help in the writing of the analysis programs. Special thanks are also due to the referees, who's elaborate comments led to many improvements in the presentation. Finally, I wish to thank Susan Massotty for correcting my English.

References

[1] K.A. Bartlett, R.A. Scantlebury & P.T. Wilkinson, A note on reliable full-duplex transmission over half-duplex links. Comm. ACM, Vol. 12, No. 5, pp. 260-265, May 1969.

[2] G.V. Bochmann, Communication protocols and error recovery procedures. ACM SIGOPS Operating Systems Review, Vol. 9, No. 3, pp. 45-50, March 1975.

[3] G.V. Bochmann, Finite state description of communication protocols. Publ. No. 236, Dept. d'Informatique, University of Montreal, 55 pp., July 1976, (see page 35-37).

[4] IEEE Transactions on Communications, Special Issue on Computer Network Architectures and Protocols. Vol. COM-28, No. 4, April 1980. See especially:
G.V. Bochmann & C.A. Sunshine, Formal methods in communication protocol design, pp. 624-631, and
P. Zafiropulo, C.H. West, H. Rudin, et al., Towards analyzing and synthesizing protocols, pp. 651-661, (cf. [13]).

[5] G.J. Holzmann, The analysis of message passing systems, Bell Laboratories Computing Science Technical Report, CSTR No. 93, March 1981.

[6] P.M. Merlin, Specification and validation of protocols, IEEE Transactions on Communic., Vol. COM-27, pp. 1761-1680.

[7] R. Milner, A calculus for communicating systems. Lecture Notes in Computer Science, Vol. 92, 1980, (170 pp).

[8] W.F. Ogden, W.E. Riddle & W.C. Rounds, Complexity of expressions allowing concurrency. Conference Record of the 5th Annual ACM Symposium on Principles of Programming Languages, Tucson, Arizona, pp. 185-194, Jan. 1978.

[9] A.C. Shaw, Software descriptions with flow expressions. IEEE Transactions on SE, Vol. SE-4, No. 3, pp. 242-254, May 1978.

[10] S. Schindler, Algebraic and model specification techniques. Proc. 13th Hawaii Intern. Conference on System Sciences, pp. 20-34, Jan. 1980.

[11] C.A. Sunshine & Y.K. Dalal, Connection management in transport protocols. Computer Networks, Vol. 2, No. 6, pp. 454-473, Dec. 1978.

[12] A.Y. Teng & M.T. Liu, A formal approach to the design and implementation of network communication protocols. Proc. COMPSAC 78, IEEE, pp. 722-727.

[13] P. Zafiropulo, Protocol validation by duologue-matrix analysis. IEEE Transactions on Communications, Vol. COM-26, No. 8, pp. 1187-1194, Aug. 1978 (see also [4]).

APPENDIX

Regular Expressions

Define a set of symbols and call it V. The following rules determine whether an expression constructed from the symbols in V and the operators plus, dot, and star is a regular expression. We use parentheses for grouping.

1. Every single symbol from set V is a regular expression.
2. If a and b are regular expressions, then so are a+b, a.b, (a), and a^* . The dot operator is left and right distributive over the plus.

Any string of symbols and operators which forms a regular expression represents a set of input symbol sequences that is accepted by a state machine. In some cases we need a special symbol to represent an empty (nil) string. The empty string represents a set with precisely one element: a string of zero length. We will use the symbol 1 to represent this.

We also occasionaly need a special symbol to represent the empty set of input sequences. This particular symbol cannot be replaced by any sequence of input symbols, not even by the empty string. For this we use the symbol ∅. Below we give a complete set of identities that hold for all regular expressions. Lower case characters represent arbitrary regular expressions.

R.01 $a + (b + c) = (a + b) + c$

R.02 $a . (b . c) = (a . b) . c$

R.03 $a . (b + c) = (a . b) + (a . c)$

R.04 $(a + b) . c = (a . c) + (b . c)$

R.05 $a + b = b + a$

R.06 $1 . a = a$ and $a . 1 = a$

R.07 $\varnothing . a = \varnothing$

R.08 $1 = \varnothing^*$

R.09 $a^* = 1 + a^* . a$ and $a . a^* = a^* . a$

R.10 $a^* = (1 + a)^*$

The precedence of the operators is: star, dot, and plus (in decreasing order).

Especially helpful in an analysis of protocols are identities of the type:

C.1 $a + a = a$

C.2 $a + \varnothing = a$

which can be reduced to identities R.01 to R.10.

Note also that the loss of a message on the transmission channel can be modeled straightforwardly with the aid of the symbol 1. (See section 2.)

Protocol Expressions

The following rules determine wether an expression constructed from the symbols in set V, and the operators star, dot, plus, division, and cross is a legal protocol expression.

1. If a is a regular expression then a and 1/a are protocol expressions.
2. If a and b are protocol expressions then so are a+b, a.b, aXb, (a), and a^*.

Note that this definition will rule out expressions with fractions in the denominators of other fractions.

Identities I.01 to I.05 (section 3), and I.06 to I.09 (section 4) apply to protocol expressions.