On the Algebraic Synthesis of Protocols

Gerard J. Holzmann

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

A simple algebraic notation is used to derive a protocol specification from a formalized description of a data link. The goal of the protocol will be reliable data transfer by correcting for the potentially erroneous behavior of a channel.

1. Introduction

One of the toughest open problems in the area of protocol verification is to find analytical models with sufficient descriptive power to model the main characteristics of a protocol, and yet with sufficient analytical power to allow us to prove interesting properties about it. The requirements conflict. There are models with sufficient descriptive power (programming languages) that are hard to verify. There are also models that can be analyzed (e.g. Petri Nets), but cannot adequately model the protocol properties we are interested in. Adding descriptive power makes the analyses intractable; removing it makes them irrelevant.

Within the limits of the problem as stated here, progress has been made in the construction of tools that can offer practical help to a protocol designer in proving protocols correct [e.g. Zafi '80, Blum '82, Holz '85, Nounou '86]. Most of these tools, though, require us to construct the model of the protocol being validated as a separate entity: the model cannot usually be derived from the implementation, nor can an implementation easily be derived from a model that was proven correct. Unintended idealizations in the model may then mislead the programmer on the correctness of his design. Similarly, simplifications in the model itself can confuse the results with artificial errors.

Many of the above problems can be overcome if we would be able to prove a protocol correct by construction. We start out with a small, trivially correct, model, and step by step refine and expand it, while proving that each refinement preserves the correctness of the initial model. Though the technique is familiar in the domain of sequential programming little progress has been made applying it to distributed programs.

Methods have been studied for partitioning a sequential program into a distributed program, preserving functionality and correctness [e.g. Moitra '85, Prinoth '82]. Such algorithms, however, require us to solve the problem, through the derivation of a sequential program, before the synthesis method itself can be applied. [Bochmann '86] describes a method to derive parts of a lower level protocol from a higher level service specification. The method derives the message exhanges necessary to enforce time orderings, while assuming error free data transfers. More closely related to the work reported here, however, are methods that have been used to derive the description of a protocol entity from a specification of its communication partner [Brand '80, Gouda '83, Merlin '83, Zafi '80]. The method described here can be seen as an extension of this earlier work.

Overview

Section 2, introduces the model of distributed systems that forms the basis of our synthesis method. The model is simple, but flexible enough that more complicated systems can be built from its primitives. The next section provides an algebraic language that can formalize the behavior of a system. Then, using the model as a guideline, we define an algebra to manipulate the formulas and synthesize protocols.

2. The Model

Our model has three primitive elements: *processes*, *links*, and *messages*.

- A message is an abstract entity with a unique identity that can be communicated across links from process to process.
- A link too has a unique identity. Without loss of generality, we assume that a link can accept only one message at a time. Upon the acceptance of a message the link becomes *blocked*. The link only returns to its original state when the message it holds is claimed by a receiving process.
- A process is a finite state machine that can send and receive messages.

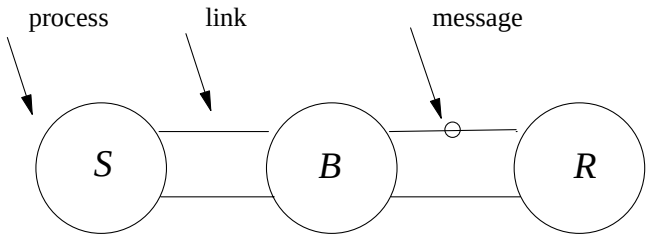

Figure 1

Processes may perform two types of operations on a link: send and receive. The *send* operation is executable only when it addresses a link that is not blocked. The receive operation can be used in two different ways: *universal* or *typed*. The *universal receive* operation is executable only when the link addressed is blocked. The *typed receive* operation is executable when the link addressed is blocked **and** holds a message of the type that is specified by the receiving process.

Note that *unexecutability* is not synonymous with *delay*. If an operation is unexecutable the executing process can avoid delay by selecting an alternative operation (see below).

We will now develop a notation, a language, to describe the behaviors of the processes in this model.

3. The Language

The notation used is based on the one introduced in [Holz '82]. We abstract from all actions that are not related to message passing. Messages are represented by symbols and process behaviors are expressed in formulas built from these symbols and a small set of operators. The formulas representing process behavior are called *protocol expressions*.

Protocol expressions are an extension of regular expressions. They use five different operators. In descending order of precedence they are: a star, a slash, a dot, a plus and a cross. Parentheses are used for grouping and the dot may omitted where this causes no confusion.

A dot indicates a time ordering, a plus indicates a choice and a star indicates a repetition zero or more times. For instance, if a and b are two protocol expressions, then $a + b$ is another protocol expression that specifies a choice between the two possible executions a and b. The choice is determined by the leading symbols in subexpressions a and b, and the state of the links that they refer to. If the leading symbols happen to be the same in both a and b a nondeterministic choice is made. Similarly, $a^*.b$ defines an arbitrary number of executions of a followed by a single execution of b. The choice between a repetition and a continuation with b is again determined by the leading symbols in a and b.

If a is a regular expression of the symbols used in the protocol and the operators dot, plus and star, then a and $1/a$ are also protocol expressions. Furthermore, if a and b are protocol expressions, then so are $a + b$, $a.b$, $a \times b$, (a), and a^*. Note that the denominator of a "fraction" must be a regular expression. Expressions such as

$$\frac{1}{p + (1/q)}$$

are therefore undefined.

Division

If p is a message symbol then expression p will be used to represent a typed[2] reception of message p, and $1/p$ will represent the sending of p. A few rewriting rules can help to determine the meaning of expressions such as:

$$\frac{r}{p+q^*} \tag{a}$$

where p, q and r are symbols. First of all, the fractional notation a/b is merely used as shorthand

$$\frac{a}{b} = a.\frac{1}{b} \tag{1}$$

so we already have:

$$\frac{r}{p+q^*} = r.\frac{1}{p+q^*}$$

Secondly, the slash is defined to be left and right distributive over the plus.

$$\frac{a+b}{c} = \frac{a}{c} + \frac{b}{c} \tag{2}$$

$$\frac{a}{b+c} = \frac{a}{b} + \frac{a}{c} \tag{3}$$

So we can rewrite the expression (a) as

$$r \cdot \left(\frac{1}{p} + \frac{1}{q^*} \right)$$

Although the star does not in general distribute over the slash, we can use the fact that $1^* = 1$ (see below), to write the last expression as

$$r \cdot \left(\frac{1}{p} + \left(\frac{1}{q} \right)^* \right)$$

Now we can read the expression as specifying the reception of message r followed by either the sending of a single message p or the sending of zero or more messages q.

There a two special symbols in this algebra: 1 and 0. Note that a protocol expression really defines a *set*: the set of possible executions. As a protocol expression by itself, 1 defines a set of executions with one element: the null execution or no-op. The symbol 0 on the other hand defines the empty set; it models a dead end, a hang state. It is easy to see that we must have

$$1 = \frac{1}{1} = 1^* = 1+1 = 1.1 \tag{4}$$

and

$$1.a = a = a.1 \tag{5}$$

Equality $1 + 1 = 1$ is in fact a special case of the more general:

$$a + a = a \tag{6}$$

The semantics of the symbol 0 prompts the following two rules

$$a + 0 = a \tag{7}$$

since a choice between executable behavior and unexecutable behavior is forced, and

$$0.a = 0 \tag{8}$$

---

2) The notation for universal receives is given below, in the section on "Variables and Links".

since nothing can follow something that cannot happen. Furthermore, we can write

$$1 = 0^* \tag{9}$$

since zero or more times something that cannot happen is a no-op.

Note that the slash does not distribute over the dot. For a numerator of 1, however, we define the rewriting rule:

$$\frac{1}{a.b} = \frac{1}{a} \cdot \frac{1}{b} \tag{10}$$

Multiplication

If a and b are protocol expressions then $a \times b$ specifies the concurrent execution of a and b. More precisely, the cross operator derives all executable combined behaviors implied by the two protocol expressions. If the two expressions contain no common symbols, the executions they specify are disjoint and the cross product is the full shuffle of all combinations of all executions defined by the two expressions [Ogden '78]. If the two expressions have symbols in common the cross product is a subset of the full shuffle consisting of only those combined executions that are feasible within the model we specified earlier.

The expansion of a cross product is in fact no more than the derivation of the behavior of the two concurrent processes.

$$(p.a) \times (q.b) = p.(a \times (q.b)) + q.(b \times (p.a)) \tag{11}$$

Operations on symbols that two processes have in common (and do not share with any other process) become internal to the combined process and can be removed from the cross products. Terms that specify unexecutable behavior are again replaced by a single symbol 0. To hide successful interactions we can replace matching send/receive pairs in cross product terms by 1 symbols. For instance, if p is a symbol, and no symbol in expression a addresses the same link (for the proper notation see below) we can reduce:

$$\frac{1}{p}.a.p = 1.a.1 \tag{12}$$

More Rewriting Rules

Here are some other useful rewriting rules. We will make no attempt here to derive a complete set. The consistency of these and other rules can be evaluated by refering to the model from section 2. Note that the behaviors on both sides of an equals sign must be indistinguishable when interpreted in the model of section 2. Plus, dot and cross are associative operators.

$$a + (b + c) = (a + b) + c \tag{13}$$

$$a.(b.c) = (a.b).c \tag{14}$$

$$a \times (b \times c) = (a \times b) \times c \tag{15}$$

Dot is right distributive over plus.

$$(a + b).c = (a.c) + (b.c) \tag{16}$$

Unlike dot, plus is a commutative operator.

$$a + b = b + a \tag{17}$$

Some rules for rewriting expressions with stars:

$$a^* = 1 + a^*.a \tag{18}$$

$$a^*.a = a.a^* \tag{19}$$

$$a^* = (1 + a)^* \tag{20}$$

Not all rewriting rules that hold for regular expressions hold also for protocol expressions. Specifically

$$(a.b)+(a.c) \neq a.(b+c)$$

In the expression on the left, the choice between the two behaviors $a.b$ and $a.c$ is non-deterministic. In the expression on the right the decision is determined by the availability of message b or c. For the same reason the distributive law is also invalid in Milner's CCS [Milner '80].

Variables and Links

There are just a few more things missing from our language: the specification of the transfer links and the usage of variables. We will use the following conventions.

A message p that is sent on a link named i is written $i{:}\,a$. The link prefix can be omitted wherever this causes no confusion.

We will reserve Greek symbols for names of variables. Initially, a variable is equivalent to the symbol 1. But the variable can be assigned a new identity in a receive operation. The usage of a variable in a receive operation makes it a *universal* instead of a *typed* operation. It means that the identity of the next message received is to be assigned to the variable. The usage of a variable in a send operation will mean that the last symbol that was assigned to the variable is to be transferred.

The working of a 1-slot buffer can now simply be modeled with one variable α.

$$\left(\frac{in{:}\,\alpha}{out{:}\,\alpha}\right)^*$$

where *in* and *out* are message links.

Canonical Forms, Inverse Formulas

Any protocol expression can be rewritten in canonical form as a sum of dot products (cf. for instance [Cooper '67]). To account for potentially infinite behavior we allow the use of parentheses and Kleene stars within the dot products. Each behavior can now be expressed as a finite sum:

$$\sum_i^N p_i$$

where p_i defines a dot product of simple fractions such as $1/a$ and $a/1$, with a a message symbol. The only operators allowed in p_i are dot, star and slash.

The dot products formalizes all possible execution sequences of the process specified. The *inverse* $\bar{P}$ of a protocol expression P is defined as the expression that is obtained by replacing every occurrence of $1/a$ with a and every occurrence of $a/1$ with $1/a$ in the canonical form, for every a in the alphabet of the expression (i.e. for every message symbol used).

Restriction

A restriction operation is used to hide a given class of messages from an expression [Milner '80]. The following definition will suffice here. If P and Q are expressions defining the behavior of two communicating processes, then $P\backslash Q$ is the expression P restricted to the interaction with Q. That is: all message symbols that are outside the set of messages exchanged between processes P and Q are replaced by 1 symbols.

4. Synthesis

The notation we have introduced so far gives the syntax of a terse, algebraic language for specifying protocol behaviors. Every operation in the language defines a set of possible events in the model that underlies it. The model defines the semantics of the language, and determines which rewriting rules will apply. The rewriting rules themselves allow for straightforward algebraic manipulation of the formulas that can be exploited in protocol synthesis.

Hoare [Hoare '85] observed that a data transfer protocol really tries to reproduce the behavior of a buffer process that delivers the messages it receives without distortion, deletion, or reordering. The protocol must reproduce this ideal behavior even when the data transfer path itself is not ideal.

Recall the specification of a buffer process B, modeling an ideal channel:

$$B = \left(\frac{in:\alpha}{out:\alpha} \right)^* \tag{b}$$

The protocol we set out to derive will describe the behavior of three processes: a sender S, a receiver R, and the data link B:

$$S \times B \times R$$

To simplify the following discussion somewhat we assume the task of the protocol to be the reliable transfer of precisely one message named m. A generalization will be relatively easy. We can now write the buffer expression without variables.

$$B = \frac{in:m}{out:m}$$

If the channel is to communicate successfully with its environment, the the environment will have to behave as the inverse of B. The environment of B will consist of two processes, S and R. We derive two restricted behaviors from B: one restricted to the message set that is intended to be shared by B and S, and one restricted to the message set shared by B and R:

$$B\backslash S = in:m$$

$$B\backslash R = \frac{1}{out:m}$$

The inverse expressions give us appropriate behaviors for S and R:

$$S = \overline{B\backslash S} = \frac{1}{in:m}$$

and

$$R = \overline{B\backslash R} = out:m$$

The derivation is trivial in this simple case, but will be useful as a guideline for the treatment that follows. The method for deriving a process description from the inverse of a given formula is similar to [Merlin '83]. Below we use the derived formulas as a starting point for further refinements.

Let us assume that the sender obtains a new message to transfer via an external link I. The receiver deposits the symbols it receives through another external link that we shall name O. The full protocol then becomes

$$\frac{I:m}{in:m} \times \frac{in:m}{out:m} \times \frac{out:m}{O:m}$$

which, using for instance rewriting rules (15) and (11), can be rewritten as

$$\frac{I:m}{in:m} \cdot \frac{in:m}{out:m} \cdot \frac{out:m}{O:m}$$

and reduces with (12) to

$$\frac{I:m}{O:m} \tag{c}$$

As intended, the external, combined behavior of sender, receiver and channel is equivalent to that of the ideal buffer process.

The synthesis problem for this data transfer protocol can now be stated more precisely as the problem of finding S and R for a given expression B such that the cross product does indeed reduce to c. To find the expressions for S and R in a stepwise manner, we can gradually expand the expression for B from the simple buffer expression into a more complex channel behavior.

Assume, for example, that the channel can distort data. The expression for B can then be written:

$$B = \frac{in:m}{out:x + out:m}$$

where x is a special symbol that is used to indicate a message that can be recognized by the receiver as being faulty, for instance by calculating a checksum. The synthesis problem is to find the expressions for S and R that compensate for the erroneous channel behavior.

We can again derive the inverses of the restricted channel behaviors. $\overline{B\backslash S}$ is unchanged. $\overline{B\backslash R}$ becomes:

$$\overline{B\backslash R} \;=\; out{:}\,x + out{:}\,m$$

Adding the communication with the external link O, we can then write in a first approximation:

$$R \;=\; \frac{out{:}\,x + out{:}\,m}{O{:}\,m}$$

which preserves the correctness of the composite $S \times B \times R$ (equivalence with c), but at the price of accepting distorted data as correct. To ignore a distorted message we can try:

$$R \;=\; out{:}\,x + \frac{out{:}\,m}{O{:}\,m}$$

but now $S \times B \times R$ reduces to

$$\frac{I{:}\,m}{1 + O{:}\,m}$$

instead of c. The only way to recover from a distorted message is to have the sender retransmit it. To make this possible the receiver can signal back to the sender whether the message was rejected or accepted with two control messages, say r and a. This refinement requires an extension of all three processes. Let us first extend B and try to derive an initial version of the expanded R. We will use a new link j to transfer message r from R to B.

Assume the channel simply forwards the new messages from the receiver to the sender via a link k, error-free. We will allow for multiple rejects and retransmissions and have the channel process terminate when an acceptance message has passed.

$$B \;=\; \left(\frac{in{:}\,m}{out{:}\,x + out{:}\,m} \;+\; \frac{j{:}\,r}{k{:}\,r}\right)^{*} \cdot \frac{j{:}\,a}{k{:}\,a}$$

The inverse of $B\backslash R$ is now:

$$\overline{B\backslash R} \;=\; \left(out{:}\,x + out{:}\,m + \frac{1}{j{:}\,r}\right)^{*} \cdot \frac{1}{j{:}\,a} \qquad \text{(d)}$$

To derive the new expression for R from this expression we will have to include some requirements for the behavior of the receiver that are independent of the channel. As far as the channel is concerned, for instance, there is no relation between control messages and data messages. The receiver will have to be more specific. We rewrite expression d as:

$$\left(out{:}\,m \;+\; \frac{out{:}\,x}{j{:}\,r}\right)^{*} \cdot \frac{1}{j{:}\,a}$$

We also know that the first correctly received m should terminate R:

$$\left(\frac{out{:}\,x}{j{:}\,r}\right)^{*} \cdot \frac{out{:}\,m}{j{:}\,a}$$

Note that these two revisions did not add execution sequences to expression d: they removed a few that were in conflict with the requirements for the protocol. The derivation of d gave us the preconditions for R's behavior given the erroneous channel behavior. The two revisions restricted the derived behavior to the protocol desired. Adding external message $O{:}\,m$ completes the refinement:

$$R \;=\; \left(\frac{out{:}\,x}{j{:}\,r}\right)^{*} \cdot \frac{out{:}\,m}{j{:}\,a} \cdot \frac{1}{O{:}\,m}$$

Now it is the sender's task to process the control messages r and a. We have:

$$\overline{B \backslash S} = \left(\frac{1}{in:m} + k:r \right)^* \cdot k:a$$

Again, as far as the channel is concerned there is no relation between r and m messages. The sender has to establish the relation. We rewrite the derived expression as:

$$\left(\frac{k:r}{in:m} \right)^* \cdot k:a$$

Next, the communication with external link I is added:

$$S = \frac{I:m}{in:m} \cdot \left(\frac{k:r}{in:m} \right)^* \cdot k:a$$

Less formally, this says that a message m from I is retransmitted once for every reject message r received, and the send action completes upon the reception of an accept symbol a.

Reduction

Expanding the cross product of S, B and R as derived above, gives us the following expression

$$\frac{I:m}{i:m} \cdot \left(\frac{i:m}{o:x} \cdot \frac{o:x}{j:r} \cdot \frac{j:r}{k:r} \cdot \frac{k:r}{i:m} \right)^* \cdot \frac{i:m}{o:m} \cdot \frac{o:m}{j:a} \cdot \left(\left(\frac{j:a}{k:a} \cdot k:a \right) \times \frac{1}{O:m} \right)$$

which again reduces to c.

5. An Implementation

It is relatively straightforward to translate the algebraic formulae we have derived into a programming language. We will use the language C as an example. The example uses two library routines. The first, $rcv(N)$, returns the first available message from link N. The second, $snd(N, M)$, submits message M to link N. Both procedures delay until they are executable. Procedures S() and R() are to be called once per transaction (message transfer).

```
S()
{       switch (rcv(I)) {
                case  m: break;
                default: protocol_error();
        }
        snd(in, m);

        for (;;)
        switch (rcv(k)) {
                case  r: snd(in, m); break;
                case  a: return;
                default: protocol_error();
        }
}
```

```
R()
{
        for (;;)
        switch (rcv(out)) {
                case x: snd(j, r); break;
                case m: snd(j, a); snd(O, m); return;
                default: protocol_error();
        }
}
```

For comparison, in CSP [Hoare '78], with *C* the name of the channel process, the implementation would be:

S :: I?m; C!m; *[C?r → C!m :: C?a]
R :: *[C?x → C!r :: C?m → C!a] O!m

6. Message Loss, Timeouts

To mimic the behavior of a channel that occasionally loses messages we can change the last channel behavior formula as follows, using the symbol 1:

$$B = \left(\frac{in{:}\, m}{1 + out{:}\, x + out{:}\, m} + \frac{j{:}\, r}{k{:}\, r} \right)^{*} \cdot \frac{j{:}\, a}{k{:}\, a}$$

Now, the receiver no longer knows whether a message was lost or just delayed. The sender, however, can timeout on the reception of an *r* or *a* message to recover from a loss. The symbol for timeouts in the algebra is tau (τ) [Holz '82]. We can refine the sender's behavior as follows

$$S = \frac{I{:}\, m}{in{:}\, m} \cdot \left(\frac{k{:}\, r + \tau}{in{:}\, m} \right)^{*} \cdot k{:}\, a \qquad \text{(e)}$$

and leave the receiver's behavior as it was before. The expansion of the cross product is again straightforward. The timeout is defined to be executable whenever the link it addresses is not blocked (there is no message pending).

Making the channel behavior independent of specific message types, such as *m*, *r* and *a*, takes two variables, α and β. If we also introduce errors and message loss on the return channel, and make the channel behavior truly cyclic, the expression becomes.

$$B = \left(\frac{in{:}\, \alpha}{1 + out{:}\, x + out{:}\, \alpha} + \frac{j{:}\, \beta}{1 + k{:}\, x + k{:}\, \beta} \right)^{*}$$

If more than just one message is transfered expressions *S* and *R* will have to be extended. Note for instance that expression *e* does not guard against unintended generation of duplicate messages. We can again extend the specification of either process and make the matching upgrades in the peer processes. Proceeding in this way we can derive a complete data transfer protocol.

7. Conclusion

With a small example we have illustrated an algebraic method to derive a data transfer protocol by stepwise refinement. The initial solution is derived from the problem: the potentially erroneous behavior of a data link and a given specification of the required external behavior of the composite of sender, data link, and receiver.

8. References

[Blum ’82], T.P. Blumer, and R.L. Tenney, "A formal specification technique and implementation method for protocols," Computer Networks, Vol. 6 , No. 3, pp. 201-219.

[Bochmann ’86], G. von Bochmann and R. Gotzheim, "Deriving protocol specifications from service specifications," University of Montreal, Report #562, March 1986.

[Cooper ’67], "Bohm and Jacopini’s reduction of flow charts," Comm. of the ACM, 1967, Vol. 10, No. 8, pp. 463.

[Gouda ’83], M.G. Gouda, "An example for constructing communicating machines by stepwise refinement," Protocol Specification, Testing and verification, IFIP, H. Rudin and C.H. West (Eds.), North-Holland Publ., 1983, pp. 63-74.

[Hoare ’78], C.A.R. Hoare, "Communicating Sequential Processes," Comm. ACM, Vol. 21, No. 8, August 1978, pp 666-677.

[Hoare ’85], C.A.R. Hoare, "Communicating Sequential Processes," Prentice Hall, 1985, p 158.

[Holz ’82], G.J. Holzmann, "A Theory for Protocol Validation," IEEE Transactions on Computers, August 1982, Vol. C-31, No. 8, pp. 730-738

[Holz ’85], G.J. Holzmann, "Tracing Protocols," AT&T Technical Journal, December 1985, Vol. 64, No. 10, pp 2413-2435.

[Merlin ’83], P. Merlin, and G. von Bochmann, "On the construction of submodule specifications and communication protocols," ACM Trans. on Programming Languages and Systems, Jan 1983, pp 1-25.

[Moitra ’85], A. Moitra, "Automatic construction of CSP programs from sequential non-deterministic programs," Science of Computer Programming, Vol. 5 (1985), pp. 277-307.

[Milner ’80], R.A. Milner, "Calculus for Communicating Systems," Lecture Notes in Computer Science 92, Springer Verlag, New York, 1980.

[Nounou ’86], N.M. Nounou, "A Methodology for Specification-based Performance Analysis of Protocols," Ph.D. Thesis, Comp. Sci, Columbia University, June 1986.

[Ogden ’78], W.F. Ogen, W.E. Riddle, and W.C. Rounds, "Complexity of expressions allowing concurrency," Proc. 5th Ann. ACM Symposium on Principles of Programming Languages, Tucson, AZ, Jan. 1978, pp. 185-194.

[Prinoth ’82], R. Prinoth, "An algorithm to construct distributed systems from state machines," Protocol Specification, Testing and verification, IFIP, C. Sunshine (Ed.), North-Holland Publ., 1982, pp. 261-282.

[Zafi ’80], P. Zafiropulo, C.H. West, H. Rudin, D.D. Cowan and D. Brand, "Toward analyzing and synthesizing protocols," IEEE Trans. Commun., Vol. COM-28, No. 4, (1980), pp. 651-661.