On Communicating Finite-State Machines

DANIEL BRAND AND PITRO ZAFIROPULO

IBM Zurich Research Laboratory, Rüschlikon, Switzerland

Abstract A model of communications protocols based on finite-state machines is investigated. The problem addressed is how to ensure certain generally desirable properties, which make protocols "well-formed," that is, specify a response to those and only those events that can actually occur. It is determined to what extent the problem is solvable, and one approach to solving it is described.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*protocol verification*; F.1.1 [**Computation by Abstract Devices**] Models of Computation—*automata*; G.2.2 [**Discrete Mathematics**] Graph Theory—*graph algorithms; trees*

General Terms: Reliability, Verification

Additional Key Words and Phrases: Communications

1. *Introduction*

The trend toward distributed computing and computer networks is increasing the complexity of communication protocols. Formal methods of specification and analysis are being gradually introduced to handle the complexity. We will briefly review the most common protocol representations and relate our representation to them. There are several excellent surveys [6, 11, 18, 23] on the subject of protocol specification and verification.

The most general model describes protocols as parallel programs [7, 14, 15, 22]. In this framework one can specify all protocols and most of their properties. The cost of this generality is the undecidability of most properties. Therefore, existing methods of analysis use human assistance or take advantage of the fact that many protocol features do not use all the generality available. The latter allows a protocol to be analyzed as if it were described in a less general formalism.

A Petri net [12, 17, 19, 24] is a less general model. In this formalism, protocols are more easily analyzable, and some properties undecidable for programs are decidable for Petri nets (e.g., boundedness [16]). The reduction in expressive power by comparison with programming languages is evident in some protocols that require a very large Petri-net description.

The least powerful model is that of a single finite-state machine describing the system including all the component processes and interconnecting channels [1, 3, 10, 13]. In this model only certain protocols can be described. (For example, a protocol allowing an arbitrary number of messages in transit cannot be described.) But describable protocols are relatively easy to analyze in the sense that all properties are decidable by exhaustive analysis.

Authors' present addresses: D. Brand, IBM Thomas J. Watson Research Center, P.O. Box 218, Yorktown Heights, NY 10598; P. Zafiropulo, IBM Zurich Research Laboratory, 8803 Rüschlikon, Switzerland.

In order to improve the expressive power of finite-state machines and Petri nets, as well as to reduce the size of their descriptions, a number of extensions have been proposed [4, 17, 21, 27]. These extensions usually employ some programming language capability, which in turn makes analysis harder.

Our model [25, 29] uses explicit finite-state machines to represent processes and implicit queues to represent channels (the details are given in Section 2). The processes communicate by sending messages to one another via the channels. The queues modeling the channels have unbounded capacity to represent protocols allowing an arbitrary number of messages in transit. In a physical implementation all channels must be bounded, but the bound may be too large to be of practical use. Moreover, since protocols are supposed to operate over different channels with different capacities, a channel of unbounded capacity is the proper abstraction.

This model is in a certain sense as powerful as programming languages, for the unbounded channels can in principle be used as memory devices (see [9]). However, this is not the intention; in modeling a given physical communication system, each process is supposed to be represented by only one finite-state machine, and the channels are to be used for communication only. With such a restriction the model is less powerful in modeling processes than Petri nets (e.g., an unbounded counter cannot be represented). But it is more powerful in modeling channels (e.g., the FIFO property is built in).

The suitability of the model for representing and analyzing communication protocols has been demonstrated [20, 21, 26] on existing protocol designs. It is particularly appropriate in situations where the propagation delay is not negligible (so that several messages can be in transit at one time) and in situations where it is natural to describe the protocol parties and the communication medium as separate entities.

Our protocol model is described in Section 2. Section 3 defines when a protocol is well formed, and Section 4 presents an approach to ensuring this property. Section 4 first characterizes all well-formed protocols if the processes are restricted to being trees, rather than general transition graphs. This characterization is then used in a (partial) solution for general protocols.

The approach has been developed for a protocol synthesizer described previously [8, 29] and is based on syntactical properties derived from notions of physical causality and completeness [28]. Because this paper is not directly concerned with the practical aspects of such a protocol synthesizer, we will describe only the essential properties of the model and the approach.

2. *Model of Protocols*

Our model of protocols employs a simple and commonly used representation of the communicating processes [1, 3], namely, each process is a finite-state machine. Each pair of communicating processes is connected by a full-duplex, error-free, FIFO channel. (A nonideal channel is represented using an additional finite-state machine corrupting messages passing through it; see [29].)

Our notation for expressing transmissions and receptions is illustrated in Figure 1. A minus sign identifies the transmission of a message, a plus sign its reception. When a process is in a state from which there is an arc labeled $-m$, then it can traverse the arc and enter a new state. (For example, process USER when in state READY can traverse the arc labeled −REQ and enter state WAIT.) By traversing such an arc the message m is transmitted to the destination process via the connecting channel. (The destination process of a message will not be indicated explicitly by its transmission;

this information will either be in an accompanying text or can be inferred by observing which process contains receptions of the message.)

There are no assumptions on the time that a process spends in a state before sending a message, and there are no assumptions on the time that a message spends in a channel before it is delivered to its destination. If and when a message m arrives at a process that is in a state s, then the process will enter a new state by traversing arc labeled $+m$. (If there is no arc $+m$ attached to state s, then the protocol is incorrect; see Section 3.)

The example of Figure 1 is a simple protocol between two processes, USER and SERVER. Initially, both are in their initial states—READY and IDLE. USER can send a request (message REQ) to SERVER. After receiving REQ, SERVER enters state SERVICE; when it is finished with the service, it goes back to state IDLE while sending the message DONE to USER. Between sending REQ and receiving DONE, USER stays in state WAIT.

While idle, SERVER can detect a fault in itself. If so, it informs USER about it by sending the message ALARM. When USER receives an ALARM, it registers it and directs SERVER back to its IDLE state by the message ACK.

Now we define the model formally.

Definition 2.1. A *protocol* is a quadruple

$$\langle \langle S_i \rangle_{i=1}^{N}, \langle o_i \rangle_{i=1}^{N}, \langle M_{ij} \rangle_{i,j=1}^{N}, \text{succ} \rangle, \tag{1}$$

where

N is a positive integer (representing the number of processes),

$\langle S_i \rangle_{i=1}^{N}$ are N disjoint finite sets (S_i represents the set of states of process i),

each o_i is an element of S_i (representing the initial state of process i),

$\langle M_{ij} \rangle_{i,j=1}^{N}$ are N^2 disjoint finite sets with M_{ii} empty for all i (M_{ij} represents the messages that can be sent from process i to process j),

succ is a partial function mapping for each i and j,

$$S_i \times M_{ij} \to S_i \qquad \text{and} \qquad S_i \times M_{ji} \to S_i.$$

(succ(s, x) is the state entered after a process transmits or receives message x in state s. It is a transmission if x is from M_{ij}, a reception if x is from M_{ji}.)

For example, in Figure 1,

$$\begin{aligned} N &= 2, \\ S_1 &= \{\text{READY, WAIT, REGISTER}\}, \\ S_2 &= \{\text{IDLE, FAULT, SERVICE}\}, \\ o_1 &= \text{READY}, \\ o_2 &= \text{IDLE}, \\ M_{12} &= \{\text{REQ, ACK}\}, \\ M_{21} &= \{\text{ALARM, DONE}\}, \\ \text{succ(READY, REQ)} &= \text{WAIT}, \\ \text{succ(WAIT, DONE)} &= \text{READY}, \end{aligned}$$

and so on according to Figure 1. The function succ is defined only for eight argument pairs, for there are eight arcs in Figure 1. An example of an undefined succ value is succ(READY, DONE) or succ(SERVICE, ALARM).

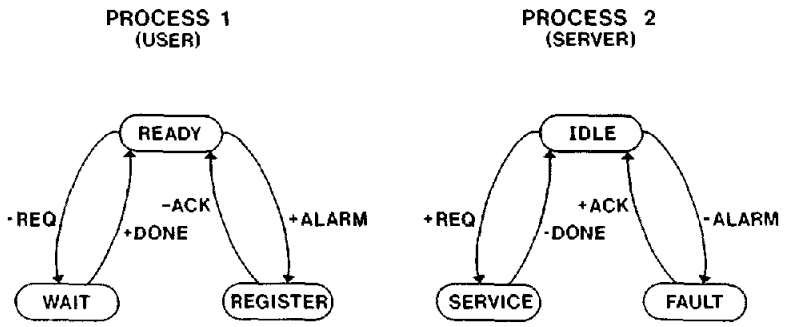


FIGURE 1

The requirement that all the sets M_{ij} be disjoint is merely a notational convenience and does not represent a restriction in practice. If two processes can send the same message, then one of the messages can be renamed to achieve distinctness.

There are extensions [8, 21, 25, 29] of the model, which are made for the handling of practical protocols. For example, a state can be marked as transient [25] so that it cannot receive messages, or the channels may not be FIFO and allow overtaking of messages. Such extensions have little impact on the results of this paper and will not be discussed.

Notation

(i) We use subscripts to identify processes. Thus, s_i always refers to a state in process i, that is, to a member of S_i. Similarly, x_{ij} refers to a member of M_{ij}, that is, a message that can be sent from process i to process j. If it is not necessary to indicate where a state s or a message x belongs, then indices are not used. Unsubscripted capital letters Q, R, S, T are used for sets of states, each from a different process. X, Y, Z are used for sequences of messages.

(ii) An N-tuple S always consists of N states $\langle s_1, \ldots, s_N \rangle$, one from each process. If a function f is defined for states, then we extend it to N-tuples componentwise:

$$f(\langle s_1, \ldots, s_N \rangle) = \langle f(s_1), \ldots, f(s_N) \rangle.$$

If a relation ρ is defined between states, then we extend it componentwise to N-tuples:

$$S \rho S' \quad \text{iff} \quad s_i \rho s_i' \quad \text{for all } i.$$

(iii) We use juxtaposition to express concatenation of message sequences. For example, if x and y are messages and X and Y are message sequences, then x, xy, xY, XY are examples of message sequences. The first one, x, is of length 1; the second, xy, is of length 2. We use the symbol ϵ to denote the empty sequence. If a relation ρ is defined between messages, then we extend it componentwise to sequences of equal length:

$$\epsilon \rho \epsilon' \quad \text{and} \quad xY \rho x'Y' \quad \text{iff} \quad x \rho x' \quad \text{and} \quad Y \rho Y'.$$

(iv) We extend the function succ to a sequence X of messages in the natural way:

$$\text{succ}(s, \epsilon) = s \quad \text{and} \quad \text{succ}(s, xY) = \text{succ}(\text{succ}(s, x), Y).$$

(v) For readability purposes we use a plus sign to identify a reception and a minus sign to identify a transmission. For example, we may write $\text{succ}(s_i, +x_{ji})$ or $\text{succ}(s_i, -y_{ij})$; here the plus and minus signs provide no new information because the

indices indicate what is a reception and what is a transmission. If we write succ(s, $+x$), then the plus sign does provide some information, namely, that there exist i and j so that s is in S_i and x is in M_{ji} (rather than M_{ij}). For a message sequence X, if we write succ(s, $+X$) = s', then X contains receptions only.

Properties of a protocol are defined in terms of its execution. An execution is a sequence of global states [25], where each global state gives the state of all the processes and contents of the channels. An execution must start from the initial global state, where each process i is in its initial state o_i and all the channels are empty. The execution can then proceed; at each step either

(i) a message is sent (process i sends a message x_{ik}, i.e., it is appended to the right of the channel c_{ik}), or
(ii) a message is received (process k receives a message x_{ik}, i.e., it is removed from the left of the channel c_{ik}).

Formally we define

Definition 2.2. A *global state* (for a protocol of the form (1)) is a pair $\langle S, C\rangle$, where S is an N-tuple of states $s_1, \ldots, s_N$ (s_i represents the current state of process i) and C is an N^2-tuple $\langle c_{11}, \ldots, c_{1N}, c_{21}, \ldots\ldots, c_{NN}\rangle$, where each c_{ij} is a sequence of messages from M_{ij}. (The message sequence c_{ij} represents the contents of the channel from process i to j. Note that every c_{ii} is empty, for M_{ii} is empty.)

Definition 2.3. We define a binary relation $\vdash$ on global states (meaning that one global state can produce the other in one step of execution): $\langle S, C\rangle \vdash \langle S', C'\rangle$ iff there exist i, k, and x_{ik} satisfying one of the two following conditions.

(i) All the elements of $\langle S, C\rangle$ and $\langle S', C'\rangle$ are equal except

$$s_i' = \text{succ}(s_i, -x_{ik}) \qquad \text{and} \qquad c_{ik}' = c_{ik}x_{ik}.$$

(ii) All the elements of $\langle S, C\rangle$ and $\langle S', C'\rangle$ are equal except

$$s_k' = \text{succ}(s_k, +x_{ik}) \qquad \text{and} \qquad c_{ik} = x_{ik}c_{ik}'.$$

Definition 2.4. Let $\vdash^*$ be the reflexive and transitive closure of $\vdash$. Then we say that a global state $\langle S, C\rangle$ is *reachable* iff $\langle S^0, C^0\rangle \vdash^* \langle S, C\rangle$, where $\langle S^0, C^0\rangle = \langle\langle o_i\rangle_{i=1}^N, \langle\epsilon\rangle_{i,j=1}^N\rangle$. ($\langle S^0, C^0\rangle$ is the initial global state with all channels empty and each process i in its initial state o_i.)

The definition of $\vdash$ does not restrict us to events (receptions and transmissions) happening one at a time rather than in parallel. If some events happen simultaneously in two or more different processes, then we can represent them as happening one after the other in any order. The definition of $\vdash$ implies only a delay between two transitions in the same process and a delay between the transmission of a message and its reception.

3. *The Problem*

We do not try to prove that a protocol performs its intended function as do, for example, [4, 7, 22]. Instead, we consider certain properties of interest to all protocols independent of their intended functions. This has two advantages: specification of the intended function is unnecessary (usually a difficult and error prone task), and the problem is computationally more tractable. At the same time we can detect a large number of common design errors such as missing or unexecutable receptions [20, 25, 29]. Avoiding errors of this kind is the main subject of this paper. Additionally,

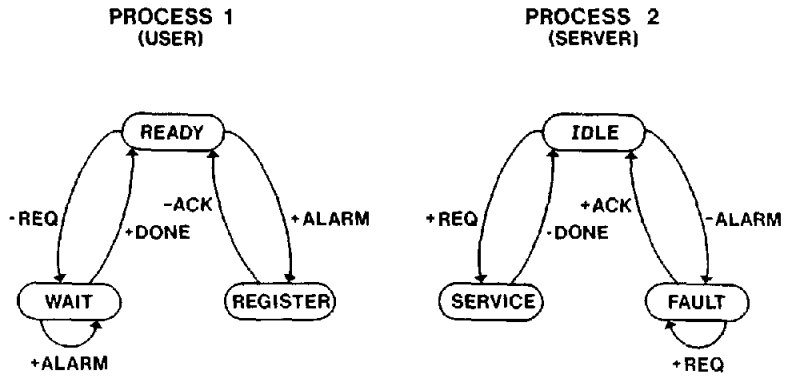


FIGURE 2

some interesting properties (e.g., deadlock) can also be discovered from so called stable N-tuples—reachable global states with all channels empty.

A protocol has a reception missing if a message x can arrive at a process when it is in a state s, but the protocol does not specify what state should be entered upon receiving x in state s. For example, in Figure 1 consider the execution where USER sends REQ and SERVER sends ALARM before either message arrives at its destination. Message ALARM arrives when USER is in state WAIT, and the protocol does not specify what should happen in that situation. Similarly, there is a missing reception of message REQ in state FAULT.

Now consider Figure 2, which is obtained from Figure 1 by adding the two receptions discussed above. It has all executable receptions specified. (An example of an unexecutable reception would be obtained, e.g., by adding a reception +DONE to state READY.) We say that a protocol with all executable receptions specified and only executable receptions specified is well formed.

While the protocol of Figure 2 is well formed, it contains the possibility of a deadlock. In the execution considered above, the two processes first send the messages REQ and ALARM. They are both received, and then the two processes remain in states WAIT and FAULT forever, because no other message can arrive. We say that a protocol can be deadlocked if it is possible to reach a global state where all channels are empty and there is no transmission from any state. It has been shown [2, 21, 25] that N-tuples of states reachable with all channels empty (in this paper called stable N-tuples) are also useful for detecting losses of synchronization. Therefore, it is important to identify such stable N-tuples.

Definition 3.1. In a protocol of the form (1):

(i) A *reception* is a pair $\langle s, x \rangle$ for a state s and a message x. A reception $\langle s, x \rangle$ is *specified* iff succ$(s, +x)$ is defined.
(ii) A reception $\langle s, x \rangle$ is *executable* iff there exists a reachable global state $\langle S, C \rangle$, where for some i and k, $s = s_k$ and c_{ik} is of the form xY for some sequence Y.
(iii) A protocol is *well formed* provided each reception is specified iff it is executable.
(iv) We say that an N-tuple S of states is *stable* iff $\langle S, \langle \epsilon \rangle_{i,j=1}^{N} \rangle$ is reachable.

Experience [20, 21, 26] shows that it is important for a protocol to be well formed independently of its function, and also that the knowledge of stable N-tuples allows the detection of significant errors. There have been two strategies for assisting a protocol designer in these aspects. Protocol validation [25, 28] assumes a given

protocol and informs the designer of any missing or unexecutable receptions, deadlocks, etc. Protocol synthesis [8, 29] assumes input given interactively by the designer and constructs mechanically a protocol guaranteed to be well formed. From a theoretical point of view, both approaches have to solve the same problem:

Given a protocol, identify all executable receptions and all stable N-tuples. (2)

In the case of validation, the given protocol is the one constructed by a human designer; in the case of synthesis, it is the partially constructed protocol at a certain stage during the protocol's construction. Of course, finding a solution to (2) will not solve all problems with protocols. For example, Bochmann and Merlin [5] synthesize a whole process when given all the other processes. For them a solution to (2) would not determine the transmissions in the synthesized process.

We cannot expect a general solution for (2). Brand and Zafiropulo [9] present a proof that it is undecidable whether a given reception in a given protocol is executable. This implies undecidability of other questions of interest: Is a given protocol well-formed? Is a given N-tuple stable? Is deadlock possible? Therefore, we can expect a solution only for some classes of protocols; we can get an idea what classes might or might not be solvable by considering the proof of undecidability. It reduces the halting problem to problem (2) using the channels to represent the tape of a Turing machine. This can be done because in our model the channels have infinite capacity. Therefore, we can expect that solvability will depend on restricting what messages can be in transit at any one time.

Definition 3.2. We say that the channel from process i to j is *bounded* by a constant h iff for every reachable global state $\langle S, C\rangle$, $c_{i,j}$ is a sequence of length at most h. If no such constant h exists, then we say that the channel is *unbounded.*

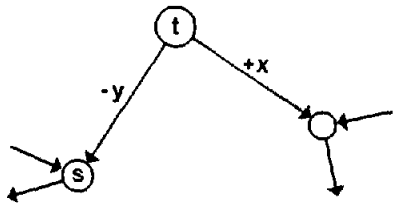
All the channels in the protocols of Figures 1 and 2 are bounded by one. A simple example of an unbounded channel is given by any protocol where $\mathrm{succ}(s, -x) = s$ (i.e., a state s has a loop transmitting x). Such a protocol can transmit x any number of times before the first x is received by the other party.

Boundedness of channels is again, in general, undecidable, but many practical protocols do have all their channels bounded. Protocols with unbounded channels usually use them in a simple manner, which makes them worth considering. For a given protocol the bound on its channels can be estimated with a method described in [9].

The problem (2) is solvable for the class of protocols with all channels bounded. As we show below, the problem is also solvable for the class of protocols with $N = 2$ processes and only one channel unbounded. In contrast, the problem is unsolvable for $N > 2$ when any one channel is unbounded. More generally, the problem is unsolvable for the class of protocols with an unbounded channel, say, from process i to j, if there exists another way of passing information from i to j. For example, if we have three processes, then we can use the third process just as a relaying station between processes i and j bypassing the unbounded channel. The same situation would exist for $N = 2$ if we extended our model by allowing more than one channel from process i to j.

For those protocols where a complete solution to problem (2) is not available, one can provide approximate solutions. For example, [25] uses a channel bound h as a parameter of validation. A validated protocol is guaranteed to have all those receptions specified that are executable using global states with channels shorter than h.

FIGURE 3



4. *A Solution*

Most existing approaches [15, 23, 25] to problem (2) are based on a search of reachable global states. We present an alternative approach; instead of considering an execution of all the processes as a whole, we consider N executions of the N processes separately. While the worst-case asymptotic behavior of both approaches is exponential, ours has a slightly smaller exponent, which indicates a potential for reducing the combinatorial explosion. However, worst-case asymptotic behavior is not necessarily a relevant measure here, because in both approaches a protocol can be analyzed successfully only if its behavior is far from the worst case, as is true for protocols designed in practice. Therefore, some experimentation will be necessary before the practical effectiveness of our approach can be determined.

There are reasons to expect the existence of a solution that does not search all reachable global states. Figure 3 contains a portion of a protocol where we wish to decide whether state s can receive message x. The reader can see that the answer is in the affirmative, provided state t can receive x, because an execution which delivers x before transmitting y implies another possible execution that differs only in that it is slower in delivering x than in transmitting y. Note that the answer to this question was independent of the rest of the protocol; we only needed the information that $s = \text{succ}(t, -y)$ and $\text{succ}(t, +x)$ is defined. Using this motivation, we aim for rules characterizing all the executable receptions with minimum information about the rest of the protocol.

The method proposed here builds N trees, one for each process, representing all the possible executions of each process. The N trees constitute a protocol according to Definition 2.1, but because of its restricted form we will call it a tree protocol. Figure 4 shows a tree protocol corresponding to the general protocol of Figure 2. Traversing a tree is like traversing the corresponding graph, but with two differences. First, one state s of a process in Figure 2 is represented in Figure 4 by several separate states $s.0$, $s.1$, $s.2$, ...; each corresponds to a different way of reaching state s from the initial state. (For example, in Figure 4 states READY.0, READY.1, READY.2 all represent the same state READY of Figure 2.) Second, messages are renamed so that no message is transmitted from two different states in the tree. (For example, −REQ.0, −REQ.1, and −REQ.2 represent three different transmissions of the same message REQ.) It should be stressed that the messages REQ.0 and REQ.1 (as an example) are completely different from the point of view of the tree protocol. They merely share their reasons for being in the tree protocol.

It should be clear that given a general protocol as in Figure 2, we can construct tree protocols (like Figure 4) by tracing all possible executions. In general, there is an infinite number of such tree protocols, depending on how long we trace the executions. Conversely, given a tree protocol, we can construct the corresponding general protocol by merging equivalent states and messages. This is the basis of our approach; we grow the N trees and interpret any findings about the trees (executable receptions, stable N-tuples) in terms of the general protocol.

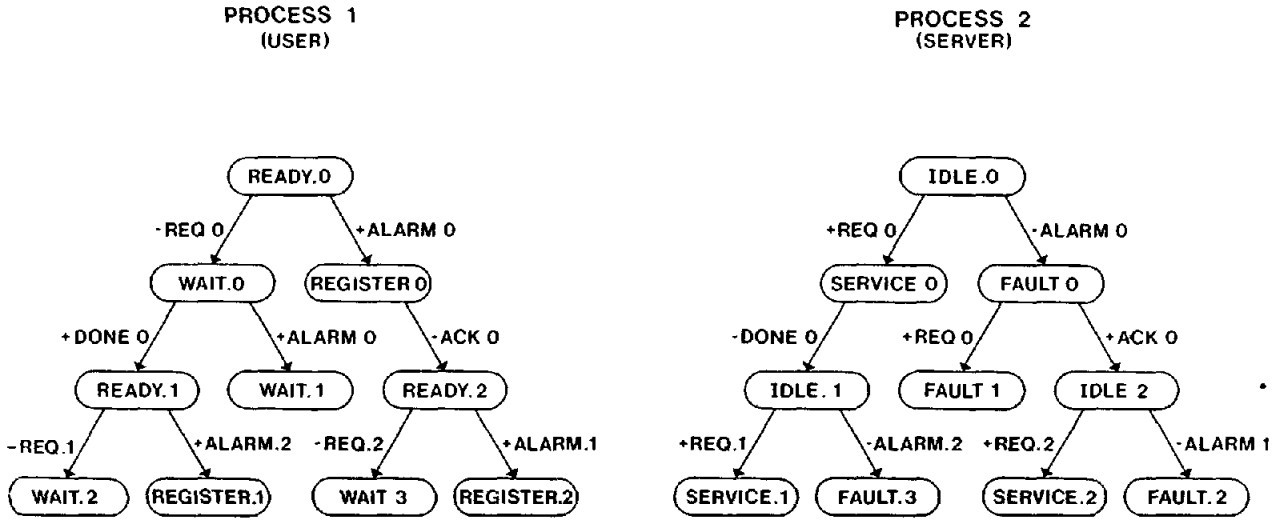

PROCESS 1
(USER)
READY.0
-REQ 0
+ALARM 0
WAIT.0
REGISTER 0
+DONE 0
+ALARM 0
-ACK 0
READY.1
WAIT.1
READY.2
-REQ.1
+ALARM.2
-REQ.2
+ALARM.1
WAIT.2
REGISTER.1
WAIT 3
REGISTER.2
PROCESS 2
(SERVER)
IDLE.0
+REQ 0
-ALARM 0
SERVICE 0
FAULT 0
-DONE 0
+REQ 0
+ACK 0
IDLE. 1
FAULT 1
IDLE 2
+REQ.1
-ALARM.2
+REQ.2
-ALARM 1
SERVICE.1
FAULT.3
SERVICE.2
FAULT.2

FIGURE 4

Thus we reduce the general problem (2) into two subproblems: first, how to solve problem (2) for tree protocols; and second, when to terminate growing of the trees without missing any receptions or stable N-tuples in the corresponding general protocol. The first problem is solved in Section 4.1. The second problem is, in general, unsolvable, and a limited solution is presented in Section 4.2. But first we define a tree protocol.

Definition 4.1. We call a protocol of the form (1) a *tree protocol* iff the following two conditions are satisfied:

(i) For every state s_i there is a unique sequence X of messages so that $s_i = \text{succ}(o_i, X)$.
(ii) For every message x there is a unique state t so that $\text{succ}(t, -x)$ is defined.

Condition (i) is the usual definition of a tree. The unique sequence X is called the *branch* leading from the *root* o_i to state s_i. Condition (ii) is independent of any treelike qualities; we introduce it because it will be convenient to have a unique state from which a message can be transmitted. If $\text{succ}(t, -x)$ is defined, then we call t the *departure* state of the transmission of x and $\text{succ}(t, -x)$ the *entry* state of the transmission.

Condition (ii) does not constitute a restriction on the protocols that the approach can handle. It merely implies that in constructing a tree protocol from a general protocol, every new transmission must be renamed.

Notation. In a tree protocol:

(i) $s \leq s'$ iff there exists a (possibly empty) sequence X so that $\text{succ}(s, X) = s'$. We call X the *path* from s to s'. If X is not empty, then $s < s'$. For example, in Figure 4 READY.0 $\leq$ WAIT.1, but neither REGISTER.0 $\leq$ WAIT.1 nor WAIT.1 $\leq$ REGISTER.0. In particular, note that $s \leq s'$ only if s and s' are in the same process.
(ii) $s \lesseqgtr s'$ iff $s \leq s'$ or $s' \leq s$ (i.e., s and s' are on the same branch).
(iii)

$$\begin{array}{ll} \max(s, s') = s' & \text{if } s \leq s', \\ \max(s, s') = s & \text{if } s' \leq s, \\ \max(s, s') \text{ is undefined} & \text{if it is not the case that } s \lesseqgtr s'. \end{array}$$

4.1 Arc Specification for Tree Protocols. This section shows how to construct well-formed tree protocols, independent of any general protocol that might correspond to the tree protocol. It gives necessary and sufficient conditions for a reception to be executable. These conditions are expressed in terms of three functions: From, To, and L. All three take two arguments—a process identifier i and a state s.

$\text{From}_i(s)$ is the last message received from process i on the branch to state s. If no such message exists, then $\text{From}_i(s) = *$, where $*$ is a special symbol used for this purpose. For example, in Figure 4, From_2(REGISTER.2) = ALARM.1, From_2(WAIT.0) = $*$. In general, $\text{From}_i(s_i) = *$ for any s_i.

$\text{To}_i(s)$ is the last message transmitted to process i by the branch to state s. For example, To_2(READY.1) = REQ.0, To_1(SERVICE.0) = $*$. In general, $\text{To}_i(s_i) = *$ for any s_i.

$L_i(s)$ is the last state in process i that *must* have been reached before state s can be reached. (L is precisely defined in Definition 4.3.) For illustration, consider the protocol of Figure 5. Process 2 sends message 1 to process 1 followed by message 2 to process 3. Upon receiving message 2, process 3 sends message 3 to process 1. Process 1 follows two different paths, depending on whether message 1 or 3 arrives

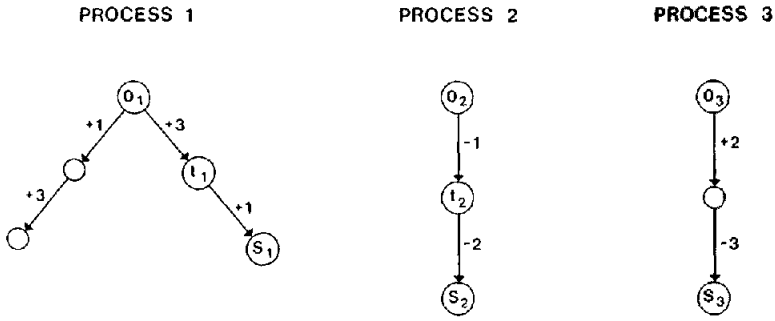


FIGURE 5

first. In determining the value of $L_2(s_1)$ we take the position that we can observe the state of process 1 only and try to say as much as possible about the state of process 2. Since +1 has been received, process 2 cannot be in state o_2; it must be in state t_2 or further. But we know even more. Since message 3 cannot be sent without message 2 being sent first, we know that process 2 must have entered state s_2. In fact, in this example $L_2(s_1) = s_2$ and $L_3(s_1) = s_3$.

Using the same reasoning, $L_2(t_1) = s_2$. Thus, when process 1 reaches state t_1, it knows that process 2 must have already sent message 1. This ability of a process to find out that a message has been sent to it before receiving the message is not present for protocols with $N = 2$ processes only. Therefore, for two processes the solution [29] is different from the general solution presented here.

When a process i reaches a state s_i, then of course it knows that process i is in state s_i; therefore always $L_i(s_i) = s_i$. Thus the function L assigns an N-tuple to every state in the protocol: $L(s) = \langle L_1(s), \ldots, L_N(s)\rangle$.

Using the function L, we express three conditions, which we will later prove to be necessary and sufficient in order that a state r_k can receive a message x_{ik} (see Figure 6):

(i) Process k can receive x_{ik} only after it has received all messages previously sent to it by process i. This first condition is expressed by

$$\text{From}_i(r_k) = \text{To}_k(t_i). \tag{3}$$

(ii) If r_k is to receive x_{ik}, then the knowledge possessed by processes i and k about any third process j must be consistent in the following sense. When process k is in state r_k, then process j must have reached state $L_j(r_k)$. After process i transmits x_{ik} entering state t_i', then process j must have reached state $L_j(t_i')$. These two states of process j must lie on the same branch, or, using the notation introduced before Section 4.1,

$$L_j(t_i') \lessgtr L_j(r_k) \qquad \text{for all} \quad j \neq k. \tag{4}$$

(iii) In order that r_k can receive x_{ik}, condition (4) must also be satisfied for $j = k$, in which case it can be simplified to $L_k(t_i) \lessgtr r_k$. But this is not sufficient, because it allows $r_k < L_k(t_i)$. If that were the case, then process k must have passed state r_k by the time x_{ik} has been transmitted, and therefore r_k could not receive x_{ik}. Therefore, we require

$$L_k(t_i) \leq r_k. \tag{5}$$

Using these three conditions, we now define a constructible reception and a constructible protocol. Later we will prove that "constructible" is equivalent to

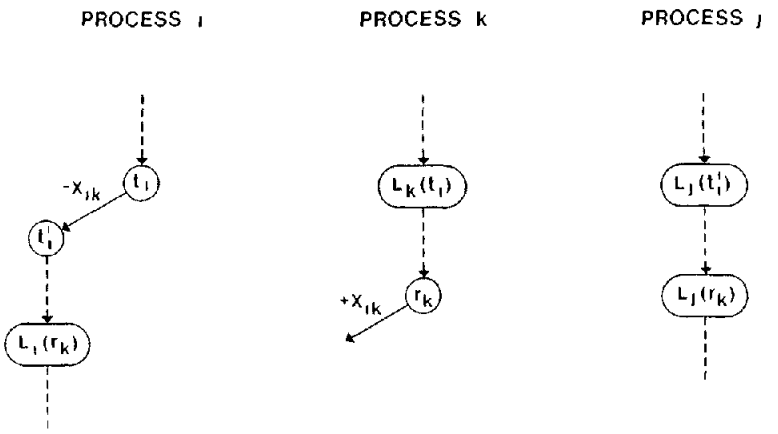


FIGURE 6

"executable." The function L is defined in Definition 4.3 together with a constructible tree protocol because they are interdependent—constructible protocols are characterized in terms of the function L, and the function L exists only for constructible tree protocols.

Definition 4.2. Assume a tree protocol of the form (1) in Definition 2.1 and a function

$$L\colon S_1 \cup \cdots \cup S_N \to S_1 \times \cdots \times S_N.$$

A reception $\langle r_k, x_{ik}\rangle$ is *constructible* iff conditions (3)–(5) are satisfied, where t_i and t_i' are the departure and entry states of $-x_{ik}$, respectively.

The next definition states that a tree protocol is constructible iff it is obtainable in any number of steps:

(i) starting from the roots,
(ii) by adding a new transmission $-x$ to any state t_i, or
(iii) by adding any constructible reception.

Definition 4.3. A tree protocol P of the form (1) is *constructible*, and

$$L\colon S_1 \cup \cdots \cup S_N \to S_1 \times \cdots \times S_N$$

is a *function associated* with P iff P and L are obtainable by the following inductive definition.

(i) P is constructible if $S_i = \{o_i\}$ for all i and M_{ij} is empty for all i, j. The following function L is associated with P: $L(o_i) = \langle o_1, \ldots, o_N\rangle$ for all i.
(ii) Let P be a constructible tree protocol of the form (1), $i \neq k$ two fixed process indices, t_i a state in S_i, and x and s new symbols. Let P' be the tree protocol obtained from P by replacing S_i with $S_i \cup \{s\}$ and M_{ik} with $M_{ik} \cup \{x\}$ and extending succ to $\mathrm{succ}(t_i, -x) = s$. Then P' is a constructible tree protocol.
If L is associated with P, then the following extension L' is associated with P':

$$L_i'(s) = s, \qquad L_j'(s) = L_s(t_i), \qquad \text{for all } j \neq i.$$

(iii) Let P be a constructible tree protocol of the form (1), L a function associated with P, $\langle r_k, x_{ik}\rangle$ an unspecified constructible reception (with respect to L), and s a new symbol. Let P' be the tree protocol obtained from P by replacing S_k

with $S_k \cup \{s\}$ and extending succ to $\text{succ}(r_k, +x_{ik}) = s$. Then P' is a constructible tree protocol.

If L is associated with P, then the following extension L' is associated with P':

$$L'_k(s) = s, \qquad L'_j(s) = \max(L_j(r_k), L_j(t'_i)), \qquad \text{for all } j \neq k,$$

where t'_i is the entry state of $-x_{ik}$. (Note that $L'_j(s)$ is well defined because (4) is satisfied.)

Condition (i) of Definition 4.3 is the basis of the inductive definition. It defines the initial protocol, where each process has only one state (the root o_i) and there are no messages. The associated function L has the same value for all the roots o_i, namely, the N-tuple of roots.

Condition (ii) says that to any state t_i of a constructible tree protocol P we can append the transmission $-x$ of a new message x and again obtain a constructible tree protocol. In order that the result of the addition is a tree again, we have to create a new state s for the entry state of $-x$. The function L' will have the same value on s as on t_i, because by sending a message x we have learned nothing new about the other processes. For process i itself we know that after sending x it is in state s. Therefore $L_i(s) = s$.

Condition (iii) describes the expansion of a constructible tree protocol by the addition of a reception $\langle r_k, +x_{ik}\rangle$. As in (ii), we have to create a new state s as the entry state of $+x_{ik}$. In order that the new reception is executable, $+x_{ik}$ can be attached only where the conditions (3)–(5) are satisfied. The extension of L to s expresses that after a reception new information about the other processes can be gained. When process k receives x_{ik}, then the transmitting process i must have reached state t'_i. And for that to happen, any process j must have reached $L_j(t'_i)$. However, it is possible that r_k has more information about a process j than t'_i has; therefore we take the maximum of the two.

Definition 4.3 allows many ways of obtaining the same constructible tree protocol. It is conceivable that different ways might lead to different associated functions. The next Lemma 4.1 states that this is not so, and therefore from now on L will always be *the* function associated with a constructible tree protocol. Proofs for the following lemma and theorem are outlined in Appendix A and carried out in [9].

LEMMA 4.1. *If L and L' are two functions associated with a constructible tree protocol, then $L = L'$.*

THEOREM 4.1. *A tree protocol is constructible iff each of its specified receptions is executable.*

COROLLARY 4.1. *A tree protocol is well formed iff it is constructible and all constructible receptions are specified.*

Theorem 4.1 says that (3)–(5) are necessary and sufficient conditions for receptions to be executable. Corollary 4.1 implies that well-formed tree protocols can be constructed by applying Definition 4.3, where step (iii) is used to specify all the receptions allowed by conditions (3)–(5).

Such a construction can be made more efficiently than suggested by Definition 4.3. In the rest of this section we outline an algorithm for finding all the receptions that must be specified as a result of a transmission. This is followed by an algorithm for finding all the stable N-tuples. The correctness of the algorithms is established in [9].

Assume a well-formed tree protocol that has been expanded by adding the transmission $\text{succ}(t_1, -x) = t'_1$ (see Figure 7). We will identify all the new receptions

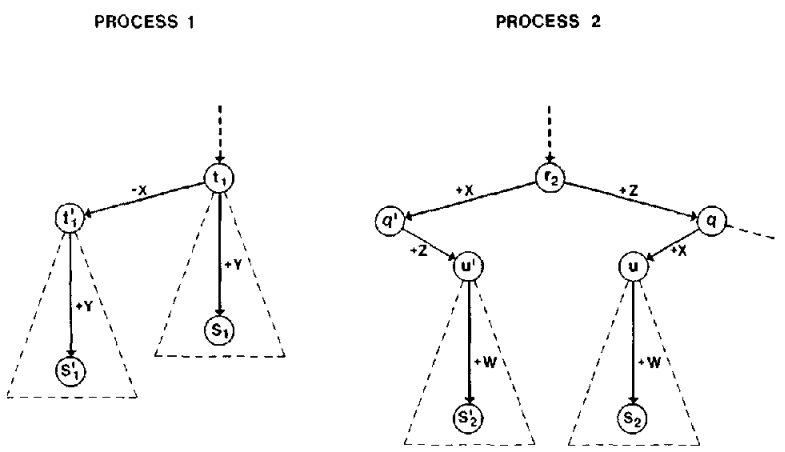


FIGURE 7

that must be specified in order that the resulting tree protocol is again well formed. Part of the construction is also the calculation of the function L for newly introduced states. The algorithm is described for the case of the new message x being sent from process 1 to process 2; the general case is obtained by renumbering the processes.

The first step of the algorithm, called propagation along negative arcs, copies the positive subtree of t_1 below t_1'. That is, for every $s_1 = \text{succ}(t_1, +Y)$ for some sequence Y, a new state $s_1' = \text{succ}(t_1', +Y)$ is generated:

$$L_1(s_1') = s_1' \qquad \text{and} \qquad L_j(s_1') = L_j(s_1) \qquad \text{for all} \quad j \neq 1.$$

The second step finds all the executable receptions of x in process 2 by testing conditions (3)–(5). (This can be done relatively efficiently by taking advantage of the tree structure.)

The last step is called propagation along positive arcs. It is done for every state of process 2 bottom up; that is, it is applied to a state r_2 only after it is applied to the whole subtree of r_2. If there are z, q, q' so that $\text{succ}(r_2, +z) = q$, $\text{succ}(r_2, +x) = q'$, and $\text{succ}(q, +x) = u$, then we create a new state $u' = \text{succ}(q', +z)$ and copy the subtree of u below u'. (Note that the subtree of u contains only receptions, for the subtree was created in previous iterations of this step, which generates only receptions.) The copying is done as in the propagation along negative arcs: a new state $s_2' = \text{succ}(u', +W)$ is created for every $s_2 = \text{succ}(u, +W)$:

$$L_2(s_2') = s_2' \qquad \text{and} \qquad L_j(s_2') = L_j(s_2) \qquad \text{for all} \quad j \neq 2.$$

At the same time as all the new receptions are being added we can also identify all the new stable N-tuples. Assume a list of all the stable N-tuples in the original protocol before the addition of $-x$. We will expand the list by all the new stable N-tuples in the new protocol.

During the second step (finding all the receptions of x) identify all the stable N-tuples $\langle s_1, q, s_3, \ldots, s_N \rangle$, where $s_1 = \text{succ}(t_1, +Y)$ for some sequence Y and $\text{succ}(q, +x) = u$. Add $\langle s_1', u, s_3, \ldots, s_N \rangle$ to the list of stable N-tuples, where $s_1' = \text{succ}(t_1', +Y)$.

During the third step (propagation along positive arcs) identify all the stable N-tuples $\langle s_1', s_2, s_3, \ldots, s_N \rangle$, where $s_1' = \text{succ}(t_1', +Y)$ for some sequence Y and $s_2 = \text{succ}(u, +W)$ for some sequence W. Add $\langle s_1', s_2', s_3, \ldots, s_N \rangle$ to the list of stable N-tuples, where $s_2' = \text{succ}(u', +W)$.

4.2 TERMINATION. In this section we assume a solution to problem (2) for tree protocols and therefore are dealing with well-formed tree protocols only. In contrast to the previous section we assume the existence of a general protocol, for which problem (2) is supposed to be solved. As explained at the beginning of Section 4, the solution is obtained by expanding a tree protocol looking for missing receptions and stable N-tuples. The purpose of this section is to determine when to stop this expansion. Appendix B outlines proofs of validity, namely, that stopping the expansion of the trees will not cause any receptions or stable N-tuples in the general protocol to be missed. This appendix also outlines proofs of termination, namely, for what classes of general protocols the method guarantees that the tree expansion be stopped (and hence for what classes it solves problem (2)).

More specifically, this section assumes a given well-formed tree protocol and a given equivalence relation $\sim$ on states and messages of that tree protocol. This equivalence relation represents all the information we need about the general protocol: Two states of the tree protocol are equivalent if they respect the same state of the general protocol, and similarly for equivalent messages. Therefore, we assume that $\sim$ relates states from the same process S_i and messages from the same set M_{ij}. Moreover, $\sim$ is a congruence relation with respect to succ:

If $s \sim s'$ and $x \sim x'$, then $\text{succ}(s, x) \sim \text{succ}(s', x')$ whenever both sides are defined.

In Figure 4, naming is used to indicate the equivalence; two states or messages are equivalent if they have the same name except for the number following the period. For example, READY.0, READY.1, READY.2 are equivalent (and no other states belong to this equivalence class).

This section gives a partial solution to the problem of deciding whether further expansion of the tree protocol can uncover new properties of the general graph—a new reception or stable N-tuple. It operates by marking states; we call such a marked state, as well as all states in its subtree, dead. The criteria for marking a state guarantee that the subtree of a dead state cannot contribute any new information to the given general graph. Thus the expansion of the tree protocol terminates when new transmissions can be appended below dead states only.

For example, suppose that in Figure 4 our criteria allow states READY.1, READY.2, IDLE.1, and IDLE.2 to be marked dead (as is actually the case). Then we do not need to generate the transmissions REQ.1, REQ.2, ALARM.1, and ALARM.2. Thus the tree construction can stop after building three levels only, and state WAIT.2 as well as all the other states on the fourth level are not needed.

There are two criteria under which a state can be marked dead, yielding two types of dead states—types 0 and 1. The criterion for type 0 is analogous to the termination mechanism of the perturbation method [25]. That method generates reachable global states; a newly generated global state can be ignored if it is equal to a previously generated global state. Our criterion for marking a state s dead of type 0 considers only that global state containing s which is reached in the least number of steps. These steps do not have to be carried out, because this minimum global state is given by $\langle L(s), C\rangle$, where $C = \text{Channels}(L(s))$ (see Definition 4.4). The state s can be marked dead if this minimal global state is equivalent to the minimal global state for some previous state s'.

Definition 4.4. For an N-tuple S in a tree protocol we define *Channels*(S), provided there exists a reachable global state $\langle S, C\rangle$. In that case $\text{Channels}(S) = C$.

Note that it is a proper definition; given S, there is at most one C so that the global state $\langle S, C\rangle$ is reachable. For if s_i and s_j are in S, then the channel from i to j consists

of all the messages on the branch to s_i that have been sent to process j but not received on the branch to s_j. Thus, calculating Channels(S) is a purely syntactic process, which does not involve searching for a reachable global state $\langle S, C\rangle$. The condition under which Channels(S) are defined can also be stated without reference to execution (Appendix A). For the method, it is necessary to determine Channels only for N-tuples of the form $L(s)$, and Lemma B7 guarantees that Channels($L(s)$) are always defined.

Definition 4.5. We say that a state s' is a *precursor* of a state s, provided

(i) $s' < s$,
(ii) $L(s') \sim L(s)$, and
(iii) Channels($L(s')$) $\sim$ Channels($L(s)$).

As an example we show why in Figure 4 READY.0 is a precursor of READY.1:

$$L(\text{READY.0}) = \langle \text{READY.0, IDLE.0}\rangle,$$
$$\text{Channels}(L(\text{READY.0})) = \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle,$$
$$L(\text{READY.1}) = \langle \text{READY.1, IDLE.1}\rangle,$$
$$\text{Channels}(L(\text{READY.1})) = \langle \epsilon, \epsilon, \epsilon, \epsilon\rangle.$$

Hence all three conditions for being a precursor are satisfied.

Having a precursor is the condition for a state s to be marked dead of type 0. The criterion for a state s to be marked dead of type 1 is motivated by the following natural expectation: If all receptions of a message can be ignored (because they are below dead states), then the transmission can be ignored, too. This condition is not valid for protocols with more than two processes, but a stronger condition expressed in Definition 4.6 is valid.

According to this stronger condition, the subtree of a state t_i can be ignored if t_i "can send" to dead states only, that is, if any message sent from t_i can be received at dead states only. The relation "can send" is obtained from Definition 4.2 of a constructible reception by substituting the condition (6) for (4). Condition (6) is implied by (4) and differs from (4) only for $j = i$, ((6) refers to t_i rather than t_i'). It does not require that the knowledge of state r_k be consistent with the transmission of any specific message. Thus it allows the possibility that r_k will be consistent with any message sent form t_i. This could be a message only later introduced into the protocol.

Definition 4.6. A state t_i *can send* to a state r_k provided the three conditions (3), (5), and

$$L_j(t_i) \lneqq L_j(r_k) \qquad \text{for all} \quad j \neq k. \tag{6}$$

Definition 4.7. A tree protocol is *well marked* iff it is a well-formed tree protocol with two distinguished subsets of all the states—states marked dead of type 0 and states marked dead of type 1—satisfying the three conditions below. If $s \geq s'$ for some s' marked dead of type n, then we will call s dead of type n.

(i) If s and t are two marked states, then it is not the case that $s \lneqq t$.
(ii) If s is marked dead of type 0, then s has a precursor.
(iii) If t is marked dead of type 1 and t can send to r, then r must be dead of type 0.

Please note the difference between a state s' *marked* dead and a state s dead just by virtue of being in the subtree of s'. Note that condition (i) guarantees that there is just one type associated with every dead state.

For example, in the protocol in Figure 4 we can mark states READY.1 and IDLE.2 as dead of type 0 because they have precursors READY.0 and IDLE.0,

respectively. This also makes states WAIT.2, REGISTER.1, SERVICE.2, and FAULT.2 dead of type 0. State IDLE.1 can send to state READY.1 and WAIT.2 only, both of which are dead of type 0, so IDLE.1 can be marked dead of type 1. Similarly, READY.2 can be marked dead of type 1. If all this marking is done, then the states READY.1, WAIT.2, REGISTER.1, IDLE.2, SERVICE.2, FAULT.2 are dead of type 0, and the states READY.2, WAIT.3, REGISTER.2, IDLE.1, SERVICE.1, FAULT.3 are dead of type 1. But only states READY.1 and IDLE.2 are marked dead of type 0, and only states READY.2 and IDLE.1 are marked dead of type 1.

Appendix B shows that this approach guarantees tree-growth termination for protocols with all channels bounded, even if we restrict ourselves to type-0 dead states. If we allow dead states of type 1, then we can terminate the growth of some protocols with unbounded channels; in particular, we prove that termination is guaranteed for protocols with only two processes and one channel unbounded.

For an implementation the following consideration must be taken into account. A state dead of type 0 can remain dead even when new transmissions are introduced into the protocol. This is not necessarily the case for a state t dead of type 1. A new transmission may cause new states to be created to which t can send. If one of such new states is not dead of type 0, then t must be "revived." Therefore states dead of type 1 should be used only in a final check on a completed protocol.

5. *Conclusions*

The purpose of this paper was to investigate a model of communications protocols from the point of view of certain properties (executable receptions and stable N-tuples). As we documented in the introduction, this model and the properties considered provide useful information for protocol designers—first, the class of errors prevented often have a crippling effect in an implementation; and second, experience has shown that designers are prone to making errors of this class.

After showing that the properties of interest are undecidable in general, we concentrated on one approach to establishing those properties for some classes of protocols. This approach has its goal in considering the execution of each process separately, as compared with considering executions of the global system of all the processes.

The whole problem was divided into two subproblems—first, growing of trees in the search for all executable receptions and stable N-tuples, and second, determining when the tree growth can be stopped. We feel that the first problem was solved successfully. We found necessary and sufficient conditions characterizing all executable receptions. In contrast, terminating the tree growth is much more difficult. First of all, it is undecidable when we can stop growing the trees. Therefore, we have to expect only partial solutions. We described a procedure that is guaranteed to terminate the tree growth if all channel capacities are finite (without any restriction on the number of processes). Termination of tree growth is also guaranteed if there are only two processes and one channel in unbounded. This is as far as we can relax the need for bounding the channels and keep the problem solvable.

The limitations of the approach lie in the termination. Since a complete solution cannot be expected, the user must be able to specify a parameter limiting the search for a solution. Such a limiting parameter should be natural to the user, easy to guess for common protocols, and easy to incorporate into a termination procedure. This area requires more research.

Appendix A. Proof Outlines for Section 4.1

(For details of these proofs see [9, App. B].)

Lemma 4.1 is proved by induction on the construction of the protocol essentially following Definition 4.3. However, one tree protocol may be constructed following Definition 4.3 in many different ways. Therefore we order all the states of a constructible tree protocol by a relation $\ll$ and its transitive closure $\ll^+$. Intuitively, "$t \ll^+ s$" means that the state t must be constructed before s can be constructed, or in other words, in any execution t must be reached before s can be reached. The ordering $\ll^+$ is well founded; thus Lemma 4.1 can be proved by induction on $\ll^+$.

Theorem 4.1 has to be proved in two directions.

(1) First we assume a tree protocol with all specified receptions executable and prove that P is constructible. Imagine generating all the reachable global states of P: start with the initial global state with empty channels, and proceed by sending and receiving messages in all possible combinations according to Definition 2.3. In parallel we can construct the protocol P: obtaining a new global state by a transmission (or reception) corresponds to adding a transmission (or reception) arc to the protocol. Note that it is necessary to prove that any such added reception arc is constructible. This process will result in constructing the whole protocol, because every reception in P is executable and thus will be exercised during generation of all the reachable global states.

(2) We assume a constructible tree protocol P and show that any specified reception $+x$ must be executable. Let $s = \text{succ}(t, +x)$; to show that x is executable, it is enough to show that there is reachable global state containing s. The global state $(L(s), \text{Channels}(L(s))$ contains s and is reachable because it satisfies the following necessary and sufficient condition for reachability:

$$(S, \text{Channels}(S)) \text{ is reachable} \quad \text{iff} \quad \text{for all } i, j, \quad L_i(s_j) \leq s_i.$$

Appendix B. Proof Outlines for Section 4.2

(For details of these proofs see [9, App. D].)

We have to prove (1) validity and (2) termination.

(1) For validity assume that P is a tree protocol obtained by tracing all possible executions of a general protocol. This determines the equivalence relation $\sim$. Assume that P has been expanded very far, even far beyond the point where our criteria would allow the expansion to be stopped. Let P'' be the subprotocol of P obtained if the expansion had been stopped as soon as our criteria allowed that. We have to prove two properties.

(a) By restricting ourselves to P'' we will not miss any receptions. That means that whenever $\text{succ}(r, +y)$ is defined in P, then in P'' there exist $r'' \sim r$ and $y'' \sim y$ so that $\text{succ}(r'', +y'')$ is defined.
(b) By restricting ourselves to P'' we will not miss any stable N-tuples. That is, whenever P has a stable N-tuple S, then P'' has a stable N-tuple $S'' \sim S$.

The proof is based on a number of lemmas, which are generally proved by induction on the tree structure. Some examples follow.

COROLLARY D.1. *Every state dead of type* 0 *has a precursor* (*not just states* marked *dead of type* 0).

LEMMA D.2. *If a state s has a precursor s' and Q is a reachable global state with $Q \geq L(s)$, then there exists an equivalent reachable global state $Q' \geq L(s')$. Moreover, the paths from $L(s')$ to Q' are equivalent to the paths from $L(s)$ to Q.*

COROLLARY D.2. *Every state outside P'' either has a precursor or is dead of type* 1.

LEMMA D.11. *If a state t dead of type* 1 *can send to a state r, then r has a precursor.*

Part (a) is proved by induction on the tree structure. The induction step has to show that there are $r' \sim r$ and $y' \sim y$ with succ(r', $+y'$) defined and with r' being closer to the root than r. Let t be the state from which the message y is being sent. Let us restrict ourselves to the interesting situation when t is outside P''.

If t happens to have a precursor, then we can apply Lemma D.2 to the global state reached just before y is received at r; Lemma D.2 gives us an equivalent global state closer to the roots. If t does not have a precursor, then by Corollary D.2 t is dead of type 1. By Lemma D.11, r has a precursor. Therefore we can apply Lemma D.2 again.

The proof of (b) uses Corollary D.2 and Lemma D.11 to show that at least one state of S must have a precursor. Then Lemma D.2 can be applied to S to get an equivalent stable N-tuple closer to the roots.

(2) For termination assume a general protocol expanded into an infinite tree protocol. We have to show that our criteria for termination allow us to ignore all but a finite portion of that infinite tree protocol. For that it is enough to show that every infinite branch will have a state with a precursor. Consider ($L(s)$, Channels($L(s)$)) for every state s along an infinite branch. If all channels are bounded, then there are only a finite number of nonequivalent global states ($L(s)$, Channels($L(s)$)). And therefore one of the states along the infinite branch will have to have a precursor.

The assumption that all channels are bounded can be relaxed in the case of two processes only; in this case it is enough when just one channels is bounded, because every infinite branch in the protocol sending over the finite channel will have a state with a precursor (i.e., a state dead of type 0), and every infinite branch of the other protocol must have a state that can send to states dead of type 0 only (i.e., a state dead of type 1).

ACKNOWLEDGMENTS. We are grateful to D.D. Cowan, R. Hauser, H. Rudin, M. Sherman, and C.H. West for valuable discussions and careful reading of the manuscript.

REFERENCES

1. BARTLETT, K A., SCANTLEBURY, R A., AND WILKINSON, P T. A note on reliable full-duplex transmission over half-duplex links. *Commun. ACM 12*, 5 (May 1969), 260–261.
2. BOCHMANN, G V. Communication protocols and error recovery procedures, *ACM Oper. Syst. Rev.* 9, 3 (July 1975), pp. 45–50.
3. BOCHMANN, G V. Finite state description of communications protocols. Proc. Computer Network Protocols Symp., Liege, Belgium, Feb. 1978, pp. F3-1–F3-11.
4. BOCHMANN, G.V., AND GESCEI, J. A unified method for specification and verification of protocols. *Proc. IFIP 77*, AFIPS Press, Arlington, Va., 1977, pp. 229–234.
5. BOCHMANN, G.V., AND MERLIN, P. On the construction of communication protocols. Proc. Int. Conf. on Computer Communications, Oct. 1980, pp. 371–378.
6. BOCHMANN, G.V., AND SUNSHINE, C. Use of formal methods in communication protocol design. *IEEE Trans. Commun. COM-28*, 4 (Apr. 1980), 624–631.

7. BRAND, D., AND JOYNER, W.H., JR. Verification of protocols using symbolic execution. *Comput. Networks 2*, 4/5 (Sept./Oct. 1978), 351–360.
8. BRAND, D., AND ZAFIROPULO, P. Synthesis of protocols for an unlimited number of processes. In *Proc. of Trends and Applications 1980: Computer Network Protocols*, National Bureau of Standards, Gaithersburg, Md., May 1980, pp. 29–40.
9. BRAND, D., AND ZAFIROPULO, P. On communicating finite-state machines. Tech Rep. RZ 1053, IBM Zurich Research Lab., Rüschlikon, Switzerland, Jan. 1981.
10. CCITT. Recommendation X 21 (revised), AP VI, No. 55-E, Geneve, Switzerland, 1976.
11 DANTHINE, A., ED Proc Computer Network Protocols Symp., Liege, Belgium, 1978; see also special issue on computer network protocols, *Comput Networks 2*, 4/5 (Sept /Oct. 1978)
12 DEVY, M., AND DIAZ, M Multilevel specification and validation of the control in communication systems. Conf. on Distributed Comput. Syst., Oct 1979, pp. 43–50
13. GOUDA, M.G., AND MANNING, E.G On the modeling, analysis and design of protocols—A special class of software structures Proc. 2nd Int Conf on Software Engineering, San Francisco, Calif., Oct. 1976, pp. 256–262
14 HAILPERN, B., AND OWICKI, S. Verifying network protocols using temporal logic. In *Proc Trends and Applications 1980: Computer Network Protocols*, National Bureau of Standards, Gaithersburg, Md., May 1980, pp 18–28.
15. HAJEK, J Automatically verified data transfer protocols Proc Int Conf on Computer Communications, Kyoto, Japan, Sept 1978, pp. 749–756
16. KARP, R M., AND MILLER, R E Parallel program schemata: A mathematical model for parallel computation *Conf Rec 8th Ann. IEEE Symp on Switching and Automata Theory* (Oct 1967), IEEE, New York, pp 55–61.
17. MERLIN, P.M. A methodology for the design and implementation of communication protocols *IEEE Trans Commun. COM-24*, 6 (June 1976), 614–621.
18. MERLIN, P.M. Specification and validation of protocols. *IEEE Trans Commun. COM-27*, 11 (Nov 1979), 1671–1680
19. PETERSON, J L Petri nets *Comput. Surv. 9*, 3 (Sept. 1977), 223–251
20 RUDIN, H., WEST, C H., AND ZAFIROPULO, P Automated protocol validation. One chain of development Proc Computer Network Protocols Conf., Liege, Belgium, Feb 1978, pp F4-1–F4-6
21. SCHULTZ, G D., ROSE, D B., WEST, C H., AND GRAY, J.P. Executable description and validation of SNA. *IEEE Trans. Commun COM-28*, 4 (Apr. 1980), 661–667.
22 STENNING, N V. A data transfer protocol *Comput Networks 1*, 2 (Sept 1976), 99–110
23. SUNSHINE, C.A *Communication Protocol Modeling*. Artech House, Dedham, Mass., 1981
24 SYMONS, F J.W Representation, analysis and verification of communication protocols. Rep. No 7380, Telecom Australia Research Labs., November 1980.
25. WEST, C H. General technique for communications protocol validation *IBM J. Res. Devel. 22*, 4 (July 1978), 393–404.
26. WEST, C H., AND ZAFIROPULO, P Automated validation of a communications protocol. The CCITT X 21 recommendation *IBM J Res Devel 22*, 1 (Jan. 1978), 60–71
27. YOELI, M., AND BARZILAI, Z. Behavioral description of communication switching systems using extended Petri nets. *Digital Processes 3* (1977), 307–320.
28 ZAFIROPULO, P. Protocol validation by duologue-matrix analysis *IEEE Trans Commun COM-26*, 8 (Aug 1978), 1187–1194
29 ZAFIROPULO, P., WEST, C H., RUDIN, H., COWAN, D.D., AND BRAND, D. Towards analyzing and synthesizing protocols *IEEE Trans Commun COM-28*, 4 (Apr 1980), 651–661

RECEIVED JANUARY 1981; REVISED APRIL 1981, ACCEPTED NOVEMBER 1981