

Towards Analyzing and Synthesizing Protocols

PITRO ZAFIROPULO, COLIN H. WEST, HARRY RUDIN, MEMBER, IEEE, D. D. COWAN, MEMBER, IEEE, AND DANIEL BRAND

(Invited Paper)

Abstract—The production of error-free protocols or complex process interactions is essential to reliable communications. This paper presents techniques for both the detection of errors in protocols and for prevention of errors in their design. The methods have been used successfully to detect and correct errors in existing protocols. A technique based on a reachability analysis is described which detects errors in a design. This “perturbation technique” has been implemented and has successfully detected inconsistencies or errors in existing protocol designs including both X.21 and X.25. The types of errors handled are state deadlocks, unspecified receptions, nonexecutable interactions, and state ambiguities. These errors are discussed and their effects considered. An interactive design technique is then described that prevents design errors. The technique is based on a set of production rules which guarantee that complete reception capability is provided in the interacting processes. These rules have been implemented in the form of a tracking algorithm that prevents a designer from creating unspecified receptions and nonexecutable interactions and monitors for the presence of state deadlocks and ambiguities.

I. INTRODUCTION

THE GROWING trends both to increase the sophistication of functions implemented in information-handling systems and to distribute these functions in different processes has resulted in an enormous growth in complexity. This complexity is particularly acute in the interactions or protocols which specify how these processes are synchronized and communicate with one another. However, formal methods are gradually being introduced to describe these interactions [1]–[8].

The benefits of using such formal methods have already proven to be substantial: the imprecise interpretation which is characteristic of prose description has been eliminated, formal proofs are now possible, and the door is opened to techniques for computer-aided validation and computer-supported synthesis or design of such interactions or protocols. It is these last two areas, computer-aided validation of protocols and computer-supported synthesis of protocols that this paper examines. These have been the main lines of research in protocols at the IBM Zurich Research Laboratory. This work has been guided by two main objectives:

- 1) automation of these techniques using computers, and
- 2) primary concern with protocol “syntax,” i.e., the logical structure of message exchange as opposed to their “semantics” or intended function.

The first objective is to provide automated tools to lighten the task of the designer while at the same time achieving a thorough analysis in the face of great complexity. A concern

primarily with syntax also guarantees a widespread applicability.

In the last years there has been a sharp increase in activity in the area of protocol investigation and the work of many individuals should be referenced here. Instead, reference will be made only to closely related work and to several survey papers. Most convenient is the paper of Bochmann and Sunshine [9], while Sunshine [10] and very recently Merlin [11] have written excellent reviews. Protocols were the subject of a conference held in Liège early in 1978; the Proceedings provide an excellent overview of the field [12].

Both our work in validation and in synthesis is based on syntactical properties derived from notions of physical causality and completeness [13, 14]. In the case of validation, a protocol is examined for these properties by means of a reachability analysis similar to that of Sunshine [3], Bochmann [15], and implemented in an automated validation system [16, 17]. Hajek has also developed a validation system using reachability analysis [18].

In validating protocols, such as the CCITT X.21 and X.25 and data flow control from IBM’s SNA, we have found that the designer(s) of a protocol do not usually foresee all the syntactic properties of the design, in that the protocol may be incomplete or logically inconsistent [19], [20], [8]. From this experience we feel well justified in examining only the limited aspect of protocol syntax. In theory, compared with assertion-proving techniques we test for little; in practice these few tests have turned out to be very effective.

An automated validation process is usually intended for a protocol in an advanced state of development, while for a protocol in the early stages of design, a synthesis technique is preferable. This paper describes two methods of analyzing protocol behavior, and both techniques can be used for either validation or synthesis. The first method, the perturbation technique, has already been implemented as an automated validation system which has had extensive use in examining existing protocols. The second method based on a set of production rules has been incorporated into an automated synthesis system. A protocol developed through the use of these production rules will be free of the same errors checked by the perturbation approach. To the authors’ knowledge there has been nothing published in the area of automated protocol synthesis other than our own first attempt [21].

The techniques of validation and synthesis and the tools described in this paper have wide-spread applicability to the entire field of cooperating processes since a protocol is a very general concept. We quote the definition given by Merlin [11] to indicate this generality: “Given a system of cooperating

Manuscript received May 16, 1979; revised December 2, 1979.

The authors are with the IBM Zurich Research Laboratory, Rüschlikon, Switzerland.

processes such that the cooperation is done through the exchange of messages, a protocol is the set of rules which governs this exchange." This statement implies that protocols are not just concerned with the correct transfer of data, but pervade all areas where interaction between processes is inherent.

II. MODELING OF PROTOCOLS

A model with which to represent protocols and interaction examples is required; we employ a representation similar to the one proposed by Bartlett *et al.* [22] and used by Bochmann [15]. Fig. 1 shows a simple access authorization protocol in which each interacting process is modeled by a finite-state graph, and the two initial states are identified by states labeled 0.

The messages exchanged between the processes are represented by integers. Message transmission is represented by the negative value of the corresponding integer, and message reception by its positive value. For example, the message ACCESS-REQUEST is represented by the integer 1, its generation is represented by traversal of the arc labeled -1 in process *A* and its reception by traversal of the arc labeled $+1$ in process *B*. The integer representation is a notational detail, but one that is compact and which lends itself to numerical manipulation.

Messages are assumed to be exchanged between processes over perfect FIFO channels. However, nonideal channels (i.e., ones which lose and distort messages) may be represented as additional processes (see Appendix I). Interactions between more than two processes may also be represented.

III. TYPES OF DESIGN ERRORS

We make two basic assumptions about protocols and interactions. First, we are not concerned with explicit time constraints such as transmission and response delays, and second, we assume the processes to be correctly initialized (all in their zero or reset states) prior to the start of an interaction. Within this framework we can handle four potential design errors, namely, state deadlocks, unspecified receptions, nonexecutable interactions, and state ambiguities. Fig. 2 shows a two-process interaction example that exhibits all these errors, each of which is explained separately in the following sections. Although the form of these design errors is syntactic, their successful resolution must consider their semantic intent. Since we are not concerned with the semantics or meaning of the interaction, messages in Fig. 2 are given no descriptive identifiers. Other potential design errors can be formulated; for example, channel overflow has been incorporated into the automated validation system [16].

A. State Deadlocks

Different types of deadlocks are definable within the context of process interactions but we shall only be concerned with state deadlocks. We define: a state deadlock occurs when each and every process has no alternative but to remain indefinitely in the same state. Stated differently, a state deadlock is present when no transmissions are possible from the current state of each process and when no messages are in

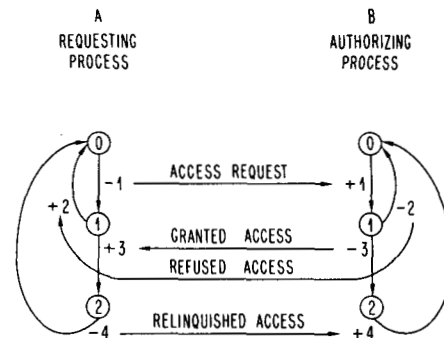


Fig. 1. Simple access authorization protocol.

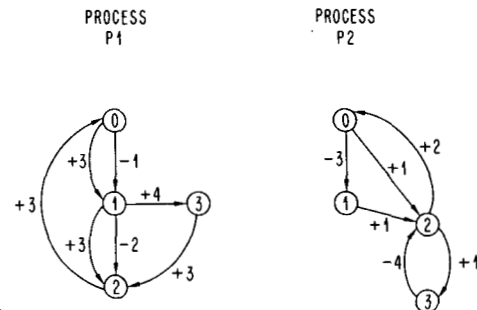


Fig. 2. Two-process interaction example containing various design and potential design errors.

transit, i.e., all channels are empty. This type of deadlock occurs in the interaction of Fig. 2 when *P1* transmits message 1 at the same time that *P2* transmits message 3. As a result both *P1* and *P2* enter states 1 and then 2 where they must wait to receive messages (no transmissions possible). As no further messages are in transit, the processes have no alternative but to wait indefinitely in these states.

State deadlocks usually represent errors but there are exceptions. Protocols may be designed to terminate in states with no exit when their function is complete. We therefore consider state deadlocks as potential errors that must be detected. Their evaluation is then a matter of semantics.

B. Unspecified Receptions

An unspecified reception occurs when a positive arc that can be traversed is missing, in other words when a reception that can take place is not specified in the design. For example, if in Fig. 2 *P2* transmits message 3, and *P1* on receiving message 3 transmits message 2, then state 1 of *P2* will receive message 2, yet this reception is not specified in the design.

Unspecified receptions are harmful since in the absence of adequate recovery procedures, occurrence of an unspecified reception causes the respective process to enter an unknown state via a transition not specified in the design. As a consequence, the occurrence of an unspecified reception causes the subsequent behavior of the interaction to be unpredictable.

Protocols can be protected by state-check mechanisms [2], [4]. These mechanisms initiate recovery procedures when states receive messages which they are not designed to accept. Unfortunately, in the case of unspecified receptions, recovery procedures can adversely modify the interaction

semantics as the occurrence of an unspecified reception is not caused by an operational malfunction yet is handled in the same manner. For example, if a connection setup protocol contains an unspecified reception and such a reception occurs in every connection setup attempt, then the ensuing recovery procedures will not fulfill the intended purpose, namely to set up a connection. In other words, error recovery procedures should not be invoked unless the error for which they have been designed has occurred.

Thus, unspecified receptions are design errors. They are more common than expected: a number of unspecified receptions were identified in the CCITT X.21 interface version of 1976 [19]. These were brought to the attention of CCITT and are reflected in the current X.21 working papers.

C. Nonexecutable Interactions

A nonexecutable interaction is present when a design includes message transmissions and receptions that cannot occur under normal operating conditions. A nonexecutable interaction is equivalent to dead code in a computer program and is illustrated in Fig. 2. No normal interaction sequences can cause state 2 of *P2* to receive message 1, hence state 3 is not entered and message 4 cannot be generated. Consequently, state 3 of *P1* cannot be reached.

The creation of nonexecutable interactions must be treated with great caution. If the designer erroneously believes that state 2 of *P2* can receive message 1 during normal operation, then the nonexecutable interaction represents a design error. On the other hand, if the designer's intention is to create recovery actions to handle abnormal conditions, and he purposely wants *P2* to enter state 3 if abnormal (error) conditions cause state 2 to receive message 1, then it does not represent a design error. In order to distinguish between normal and abnormal conditions, it is probably good design practice to design and validate a protocol for normal operation before adding recovery actions.

D. Stable-State Pairs and State Ambiguities

A stable-state pair (*x*, *y*) is said to exist when a state *x* in one process and a state *y* in the other can be reached with both channels empty. In such a case, states *x* and *y* coexist until the next transmission occurs. Monitoring stable-state pairs is useful for detecting loss of synchronization, i.e., the presence of unintended stable-state pairs or the absence of intended ones. A case of special interest is when ambiguity occurs among stable states. A state ambiguity exists when a state in one process can coexist stably with several different states in the other process. Fig. 2 contains state ambiguities. For example, if both processes are in their initial states (state 0), and *P1* transmits messages 1 followed by 2 while *P2* only receives messages, then *P1* reaches state 2 while *P2* returns to state 0. Thus, state 0 of *P2* can coexist stably with both state 0 and state 2 of *P1*. State ambiguity is closely related to the adjoint-state concept [15]: state ambiguity implies that the cardinal number of the corresponding adjoint-state set is greater than 1.

State ambiguities do not necessarily represent errors but they must be treated with caution. If, for example, the de-

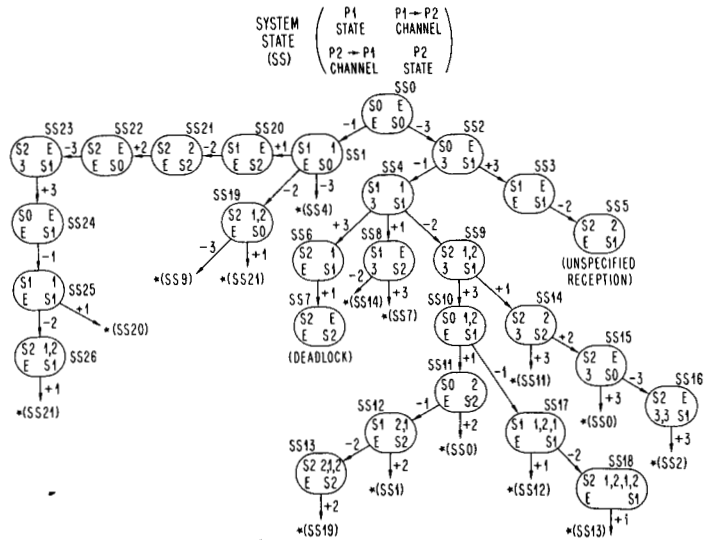


Fig. 3. Corresponding reachability tree for the example in Fig. 2.

signer's intention was that state 0 of *P1* coexist stably solely with state 0 of *P2*, then the identified state ambiguity does represent an error. We therefore consider state ambiguities as potential design errors that need monitoring. State ambiguities are detectable via an examination of syntax; their evaluation is a matter of semantics.

IV. ANALYZING INTERACTIONS

In this section we describe techniques to detect the presence of design and potential design errors in an interaction or protocol. Our first approach was based on an analysis of dialogues of interaction between communicating processes [13], [23], [24]. It was significantly improved and generalized in a method based on a technique of perturbation [16]. This technique is a reachability analysis conceptually similar to one proposed by Sunshine [3]. This perturbation method has been programmed and has successfully detected errors in protocols.

A. The Perturbation Analysis

We describe the perturbation method by analyzing in Fig. 3, the example of Fig. 2. A system state consisting of a two-dimensional array is defined where the elements on the main diagonal represent the individual process states (element 1, 1 is state of *P1* and so on) and each off-diagonal element *i*, *k* represents the message content of the communication medium from process *Pi* to process *Pk*. Fig. 2 represents a two-process interaction; hence the system states *SS* in Fig. 3 are 2 × 2 arrays.

One begins by defining *SS0* which is the initial system state. It consists of both processes in *S0* (state 0) and both channels empty (represented by *E*). *SS0* is then "perturbed" into all possible successor states reachable by executing a single transition in one of the individual processes *P1*, *P2* (in Fig. 2). Thus, either *SS1* is entered by *P1* transmitting message 1 (*P1* enters *S1* and places 1 in channel *P1* → *P2*) or *SS2* is entered by *P2* transmitting message 3 (*P2* enters *S1* and places 3 in channel *P2* → *P1*).

The procedure continues by perturbing each of these new system states in turn. Thus considering SS_2 , either SS_3 is entered by P_1 receiving message 3 (P_1 takes 3 from channel $P_1 \rightarrow P_2$ and enters S_1) or SS_4 is entered by P_1 transmitting message 1 (P_1 enters S_1 and places 1 in channel $P_1 \rightarrow P_2$). The procedure continues until no new system states are created, thus indicating that all reachable system states have been determined. Asterisks in the ensuing reachability tree indicate system states that have been previously generated by perturbation of earlier states.

The method has the attractive property that it creates the reachability tree for any n -process interaction by simply defining the system states as $n \times n$ arrays. For example, the system states for a three-process interaction are 3×3 arrays, each consisting of three process states and six channels some of which may remain empty. Certain types of interactions can cause unbounded growth in the number of messages in transit (see Section VI). In order to contain such unlimited growth, bounds are set on the channel-storage capacity. These bounds make it possible to detect when a prescribed channel-storage capacity is exceeded.

B. Error Detection via Analysis

Deadlocks are identified in a reachability tree by system states with all channels empty (E in Fig. 3) and no departing transitions. For example, the deadlock described in Section III-A (P_1 and P_2 in S_2) is identified by SS_7 . Such system states represent deadlocks because there are no further receptions (all channels empty) and no possible further transmissions (no departing transitions).

Unspecified receptions are identified by system states with no departing transition to absorb the next output from one of the channels. For example, the unspecified reception discussed in Section III-B (message 2 cannot be received in S_1 of P_2) is identified by SS_5 where the next $S_1 \rightarrow S_2$ channel output is message 2, yet there is no transition out of SS_5 to absorb that message.

Stable-state pairs (tuples for many-process interactions) are identified in the reachability tree by system states having all channels empty. State ambiguities are identified by a particular process state appearing in a plurality of such system states. For example, the state ambiguity discussed in Section III-D is identified by state S_0 of process P_2 appearing in both system states SS_0 and SS_{22} . Fig. 3 identifies other ambiguities, for example, SS_3 , SS_{24} represent an ambiguity with respect to S_1 of P_2 .

Nonexecutable interactions are identified as state transitions present in the design that are absent in the reachability tree. For example, P_2 in Fig. 2 contains a -4 arc which never appears in the tree of Fig. 3.

V. SYNTHESIZING INTERACTIONS

An alternative to testing an existing design for errors is to create from the outset a design devoid of the errors considered here. In this section we shall describe a mechanism (or tool) which is used interactively by a designer to create a protocol or interaction. The tool prevents the occurrence of unspecified receptions and immediately notifies the designer of the pres-

ence of state deadlocks and ambiguities. This immediate response has the advantage that at this point in time, the designer has the most insight into the resolution of the design problem. The tool is based on three production rules which create only those arcs needed to prevent unspecified receptions. A tracking algorithm then specifies where and when to apply the rules. Both tracking algorithm and production rules have been automated using a novel programming method called data-directed design [25], [26]. The rules are based on a study of the cause-and-effect relationships that occur when two entities exchange messages. They are currently limited to two-process interactions.

A. Production Rules

Three rules governing the derivation of two-process interactions are described in this section and proofs for their necessity and sufficiency are given in Appendix II. These rules are a modification of an earlier version which was developed [21] but was found to be incomplete. The relative simplicity of the rules rests on the fact that they are designed to produce tree-structured graphs. Section V-B shows how interactions can be constructed from such graphs. We now explain the rules.

The first rule specifies all receptions of a message whose transmission directly succeeds the reception of a previous message. Consider Fig. 4(a) where P_2 upon receiving message x transmits message e . If P_1 transmits no further messages before receiving e , then it receives e in the state entered upon transmitting x . Hence, a $+e$ arc is appended to $-x$ in P_1 . On the other hand, if P_1 transmits y before e is received, then e is received after y is transmitted. Hence a $+e$ must be appended to $-y$. We append $+e_y$ instead to note the fact that in this case messages e , y occur concurrently, or collide. Two messages are said to collide when neither is received before the other is transmitted. As we shall see, identifying collisions via subscripts is necessary for Rule 3. The subscript refers to all collisions. Thus, as shown in Fig. 4(a) if z is also transmitted before e is received, then we append $+e_{y,z}$ to $-z$. We now formulate the first rule using the generalized example in Fig. 4(b) where $-s$ represents a transmission sequence.

Rule 1: If $-e$ is appended to $+x$ P_2 then:

- a) append $+e$ to $-x$;
- b) append $+e_s$ to every negative arc sequence $-s$ attached to $-x$.

Part a) specifies collisionless receptions whereas part b) specifies all receptions associated with collisions.

The second rule specifies all receptions of a message whose transmission directly succeeds the transmission of a previous message. Consider Fig. 5(a) where P_2 transmits e directly after transmitting x . Therefore, P_1 can receive e directly after x . Hence, $+e$ is appended to $+x$ in P_1 . If P_1 transmits y before receiving x , then not only do y and x collide but y and e also collide. Then e is received after $+x_y$ and we append $+e_y$ to $+x_y$. Finally, if P_1 transmits z after traversing $+x_y$ but before receiving e , then e is received after z is transmitted. In this case e collides with both y and z , hence $+e_{y,z}$ must be

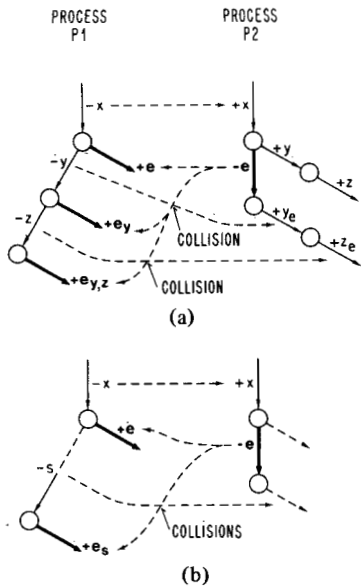


Fig. 4. Derivation of production Rule 1.

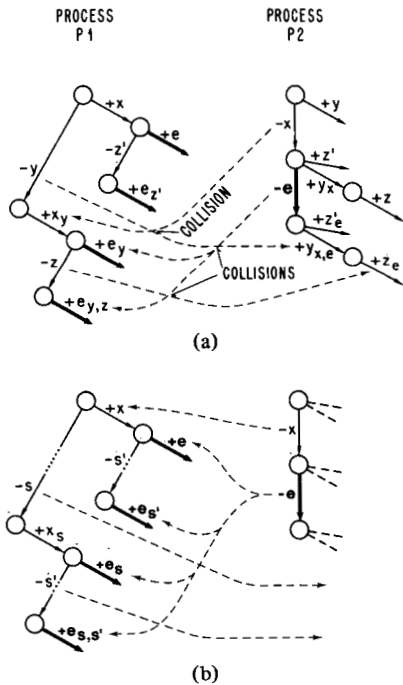


Fig. 5. Derivation of production Rule 2.

appended to $-z$. Similar circumstances hold true if $P1$ transmits z' . We now formulate the second rule using the generalized example shown in Fig. 5(b) where $-s$ and $-s'$ represent transmission sequences.

Rule 2: If $-e$ is appended to $-x$ then:

- a) to every $+x$ and $+x_s$ append $+e$ and $+e_s$, respectively;
- b) to every negative arc sequence $-s'$ attached to $+x$ or $+x_s$ append $+e_s$, and $+e_{s,s'}$, respectively.

A third production rule is necessary because new cause-and-effect mechanisms come into play when a negative arc is appended to a subscripted reception. Consider Fig. 6(a)

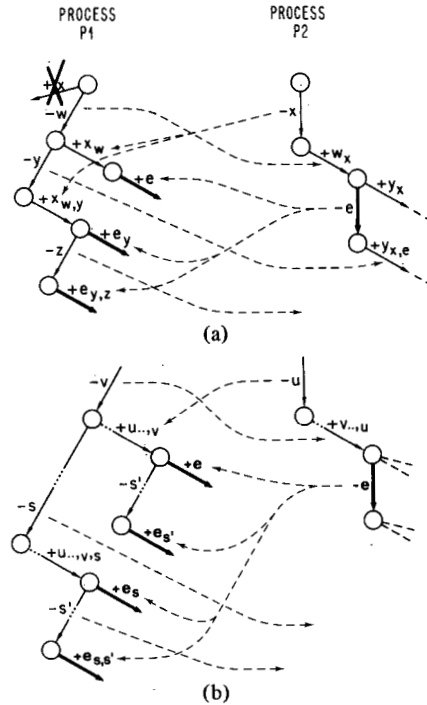


Fig. 6. Derivation of production Rule 3.

where $P2$ transmits e directly after receiving $+w_x$, i.e., after receiving a w that collides with an x . Message e is the next $P2$ transmission after x . Therefore, $P1$ receives e directly after x . But w is received before e is transmitted, hence $P1$ can only receive e after transmitting w . Therefore, $P1$ can only receive e after it both transmits w and receives x . Hence, $+e$ must be appended to $+x_w$. The arc $+e$ is not indexed because, as shown in Fig. 6(a), no collisions are associated with its transmission. If on the other hand, $P1$ transmits y before receiving x , then x collides with both w and y , whereas e collides only with y . Hence, $+e_y$ is appended to $+x_w,y$.

Finally, the mechanism of the third reception case $+e_{y,z}$ is identical to that of $+e_{y,x}$ in Fig. 5(a). We now formulate the third rule using the generalized example in Fig. 6(b) where $-s$ and $-s'$ represent transmission sequences and “...” stands for an arbitrary message sequence.

Rule 3: If $-e$ is appended to $+v...u$ in $P2$, then within the tree with root $-u$:

- a) append $+e$ to $+u...v$ and $+e_s$ to every $+u...v,s$;
- b) to every negative arc sequence $-s'$ attached to $+u...v$ or $+u...v,s$ append $+e_s'$ or $+e_{s,s'}$, respectively.

Part b) of Rule 3 describes the same specification mechanism as part b) of Rule 2.

A few notational conventions simplify application of the production rules. For example, entering the initial states via a fictitious message exchange as shown in Fig. 7 enables Rules 1 or 2 to specify reception arcs appended to initial states. Furthermore, to generate only exercisable sequences, the rules require that every negative arc within one process be uniquely specified. The ensuing problem of representing different transmission instances of a same message is solved as follows. The first transmission of a message 8 is represented by -8 ,

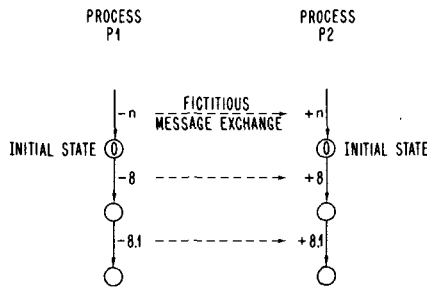


Fig. 7. Example showing minor notational extensions.

the second transmission is represented by -8.1 and so on (see Fig. 7). The eleventh occurrence would be specified as 8.10 and would be considered different from 8.1 .

B. On Using the Rules

We require an algorithm that specifies where and when to apply the rules. The algorithm is based on an incremental design approach requesting designer intervention whenever semantic-dictated decisions are needed. The designer creates state diagrams, but in order to describe the algorithm, we will consider tree structures. Consider the design portrayed in Fig. 8(a). The algorithm begins by automatically creating the fictitious message exchange $(-n, +n)$, which initializes both processes. It then requests a first design action. The designer complies and creates the transmission of message 1 in $P1$ by specifying $P1,0 \rightarrow (-1) \rightarrow (1)$ where $P1$ is the process considered, 0 is the departure state, 1 the entry state and -1 , the message transmitted. The algorithm then invokes Rule 2 (-1 is appended to $-n$) which creates $P2,0 \rightarrow (+1) \rightarrow (?)$ and requests the designer to specify the entry state identified by $(?)$. He specifies this as state 2. The algorithm again requests the next designer action which is $P2,0 \rightarrow (-3) \rightarrow (1)$. This new arc is appended to a reception, hence Rule 1 is invoked and creates the arcs $P1,0 \rightarrow (+3) \rightarrow (?)$ and $P1,1 \rightarrow (+3_1) \rightarrow (?)$. The designer then specifies the entry states as 1 and 2, respectively. This specification causes the node representing state 1 to appear twice. We are building trees, and tree nodes have at most one entry arc, hence state names may appear more than once.

Creating arc -3 in $P2$ causes arcs -3 and $+1$ to have a common origin, namely state 0. Hence, it is possible for $P2$ to receive message 1 after transmitting message 3. This reception can be specified by reapplying Rule 1 to arc -1 . But a much simpler method is to duplicate arc $+1$, append it to arc -3 and index it accordingly. Indexing is necessary, for this arc can only be traversed if messages 1, 3 collide. We call this reception-replication. The algorithm automatically executes reception-replication, thereby creating the arc $P2,1 \rightarrow (+1_3) \rightarrow (?)$ with the designer then specifying $(?)$ to be state 2.

The next designer action is to create the transmission $P1,1 \rightarrow (-2) \rightarrow (2)$. At this point the tree structure $P1$ contains two copies of state 1. Hence, the algorithm appends a second transmission $P1,1 \rightarrow (-2.1) \rightarrow (2)$ to the second copy of state 1 (in general, if state i transmits message e , then arc $-e$ is appended to the first created node i , arc $-e.1$

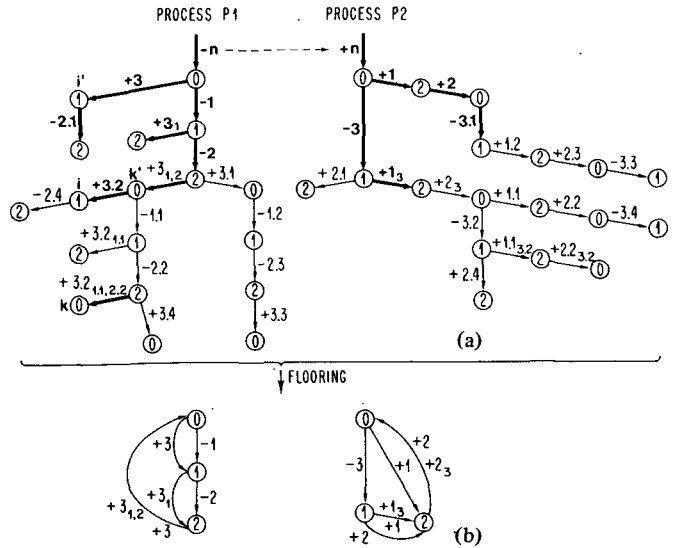


Fig. 8. Synthesis design example. Thickly lined arcs in (a) are explicitly discussed in the text.

to the second created node i , etc. and the rules are applied in the creation sequence). The algorithm then invokes Rule 2 for arc -2 and Rule 1 for arc -2.1 . This creates reception arcs in $P2$. One such arc is $P2,(2) \rightarrow (+2) \rightarrow (?)$. The designer specifies its entry state as 0 ($(?)$ set to 0). He thereby creates a cycle which enables $P2$ to retransmit message 3. The algorithm takes care of this by automatically appending an arc -3.1 to the arc $+2$ and invoking Rule 2 which in turn creates further receptions, and so on. In this way the algorithm adds arcs to the trees. This tree growth would continue indefinitely if it were not for a termination mechanism that halts the growth when the configuration of Fig. 8(a) is reached. The designer could then enter a further message transmission if he so wished. The above-mentioned termination mechanism is an important part of the algorithm and is described in Section V-D.

It is worth noting that when the algorithm creates a duplicate arc such as $+3.2$ (duplicate of $+3$ because $+3$ and $+3.2$ have same departure state) in $P1$, then its entry state must be equal to that of the original arc $+3$ and hence, no designer intervention is needed.

When the designer is finished, the algorithm collapses the tree structures by using a "flooring" operation to obtain the finite-state graphs of the actual interaction, shown in Fig. 8(b). The flooring operation drops all decimal fractions from message numbers and merges identical states and arcs in each tree. It is important to note that the algorithm masks the complexity of the tree structures from the designer by displaying all arc identifiers without decimal fractions and by not displaying duplicate arcs. The designer therefore need not even realize that the algorithm uses trees as internal representation. The reader will note that the interaction we have just designed (Fig. 8) is very similar to that of Fig. 2. In fact, it is the same interaction devoid of unspecified receptions and of nonexecutable interactions. The monitoring of deadlocks and ambiguities during the synthesis process is discussed in the next section.

C. Error Prevention via Synthesis

The algorithm together with the production rules specify those and only those positive arcs that must be created to prevent unspecified receptions. Hence, it is not possible to create nonexecutable interactions (see Section III-C).

Every time an arc pair $(-e; +e)$, $(+e_y; +y_e)$ or $(+e\dots, y; +y\dots, e)$ is created the corresponding entry states (i, k) represent a stable-state pair. Hence, stable-state pair monitoring is quite easy. A state deadlock (see Section III-A) is present if for such a pair neither state has a negative departing arc. The algorithm monitors state deadlocks by testing for the absence of negative departing arcs in every created stable state pair.

State ambiguities (see Section III-D) can be monitored in the following way. Every time a new stable-state pair (i, k) is created, it is stored in a list. If the list already contains a pair (i, x) or (x, k) , then a state ambiguity is identified.

D. Termination

As mentioned in Section V-B, the design rules could be applied continually, defining infinite trees. It is necessary to stop the growth at a point when continuation cannot reveal any new information about the protocol. This section presents a method for termination.

Termination is achieved by deleting negative arc copies. When the algorithm creates a new tree node, it tests whether certain repetition criteria are fulfilled. If they are, the node is marked "dead." Dead nodes are a form of duplicate nodes. They are treated differently in that a transmission arc as well as its corresponding reception arcs are deleted if they all turn out to be appended below dead nodes. Thus, in the example of Fig. 8, the whole process is complete because all further arcs are deleted.

We now describe the criteria that define a node dead. Consider the situation where the algorithm specifies an arc $+e$ with entry node i . This node i is marked "dead" if there already exists an arc $+e'$ with entry node i' , where e, e' represent the same message, the nodes i, i' represent the same state and i' has no dead-node predecessors. For example, in process $P1$ of Fig. 8(a), the entry node i of arc $+3.2$ is dead. This is so as $P1$ already contains an arc $+3$ with entry node i' where i, i' represent the same state 0, 3.2 and 3 the same message and i' has no dead predecessors. Similarly, if the algorithm specifies an arc $+e_s$ with entry node k , then k is marked dead if there already exists an arc $+e'_s$ with entry node k' where in addition to the above requirements being fulfilled, s and s' represent the same message sequence. For example, in process $P1$ of Fig. 8(a), the entry node k of arc $+3.2.1.1, 2.2$ is dead. This is so because $P1$ already has an arc $+3.1,2$ with entry node k' where $(1.1, 2.2), (1, 2)$ represent the same message sequence, k and k' the same state, 3.2 and 3 the same message, and k' has no dead-node predecessors.

Appendix III shows that this method is valid, i.e., it will not cause any receptions to be missed in the graph. It also shows that it will terminate the growth of the trees for any protocol where both channels are bounded. The unbounded-channel case is discussed in the next section.

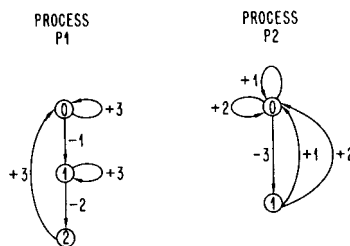


Fig. 9. Interaction exhibiting unbounded-channel growth. Indexing not shown.

VI. THE UNBOUNDED CHANNEL

In this section we consider interactions that can lead to unbounded growth in the number of messages transmitted by one process but not yet received by the other. One example of such an interaction is shown in Fig. 9. $P2$ can transmit message 3 after every message reception. Assume $P2$ does this and that at the same time $P1$ transmits messages 1 and 2 with sufficient speed so that it receives all messages in state 2. Then for every message $P1$ receives, it transmits two messages. Hence the number of messages in transit, i.e., in the $P1$ to $P2$ channel grows without bound. This is a generic example from which more complicated ones can be derived. Another type of interaction that can lead to unbounded-channel growth are transmission cycles. Such a cycle would be present if in $P1$ of Fig. 9, arc -2 were modified so as to enter state 0. $P1$ would then contain a transmission cycle $-1, -2$.

The perturbation method (Section IV-A) sets bounds on the maximum channel capacity. Hence, a perturbation analysis will always terminate when interactions exhibiting unbounded-channel growth are considered. The same holds true for the synthesis case when one sets upper bounds on the index sequences and on the number of consecutive transmissions. The consequence of these termination mechanisms is that interactions exhibiting unbounded channel growth may not be fully analyzable or synthesizable. This limitation is by no means unique to our termination mechanisms. It is a necessary property of all termination mechanisms, as will be proven in a forthcoming paper. Consequently, we can improve termination mechanisms to cover more and more practical protocols, but we must always be prepared for protocols that can never be completely analyzed or synthesized.

It is interesting to consider design criteria that guarantee unbounded-channel capacity and hence, guarantee complete analysis and synthesis. One such criterion is that every cycle in an interacting process that contains one or more transmission arcs must also contain at least one collisionless reception. This limits the channel capacity because when a collisionless reception occurs, the transmitting channel of the receiving process is empty. Hence, the transmitting channel is emptied every time a message generation cycle is traversed, thereby causing the channel capacity to be bounded.

VII. CONCLUSIONS

Two approaches to improving protocol correctness have been described. The first, perturbation, is implemented as a method for validating an existing protocol, while the second is a set

of production rules applied in a stepwise interactive manner to synthesize a "correct" design. The underlying principles of both approaches are equivalent in that the production rules could be used for validation purposes and the perturbation method could be used for synthesis purposes. Both approaches require limits on the channel content when handling protocols or interactions that exhibit unbounded-channel growth. This limitation can be transformed, for example, into design criteria which when fulfilled prevent unbounded-channel growth. But some form of limitation is a necessary condition for there is no solution to the general problem of reception specification.

In the case of validation, a thorough analysis of the CCITT X.21 circuit-switched network interface specification has already been published [19]. Some of the results of applying the perturbation technique to the data-flow-control portion of IBM's SNA network architecture are discussed in [8].

The validation procedure has also been applied to the packet-level portion of the CCITT X.25 packet-switched network interface specification. The results, which were independently discovered by Belsnes and Lynning [27], were submitted by IBM to study group VII of the CCITT [20]. The reader interested in X.25 may wish to examine the issue of *Computer Communication Review* devoted to this topic [28]. In the definition of X.25, it was found that a collision of the DCE-CLEAR-INDICATION message coming from the network could collide with the DTE-CALL-REQUEST coming from the terminal. According to the specification, the network was to identify this collision as a "local procedure error" even though such a collision is allowed by the same protocol specification. Thus, the "procedure-error" indication became ambiguous, being used both for the identification of natural collisions and actual protocol violations. The repair to this anomaly was also validated by the same method [20]. The correction has since been accepted by the CCITT study group VII's Rapporteurs' group.

An experiment was also performed using the protocol synthesis package to try to duplicate the same X.25 level-3 specification. During the redesign of this portion of the protocol (for the error-free channel), the synthesis package demanded that the receptions resulting from the previously mentioned collision be resolved as soon as the developing design makes them possible. Terminating these receptions as recommended [20] leads to the successful complete design.

Our work and that of others in protocol specification and validation has only examined one aspect of a large and important area which perhaps should be called "interaction science." Work of others on such topics as concurrent programming is exploring this science from a different viewpoint. Many of the problems inherent in distributed processing will be resolved as this science develops.

APPENDIX I

FURTHER CONSIDERATIONS ABOUT THE MODEL

The representation described in Section II can be used to model both nonideal communication channels and interactions between more than two processes [16]. This is illustrated by

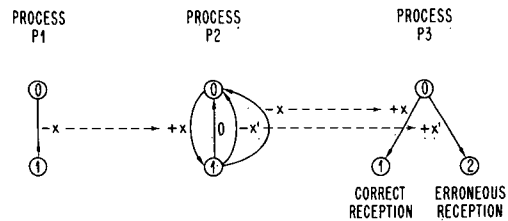


Fig. 10. Interaction example demonstrating how to model many process interactions and how to include communication channels that can lose and distort messages.

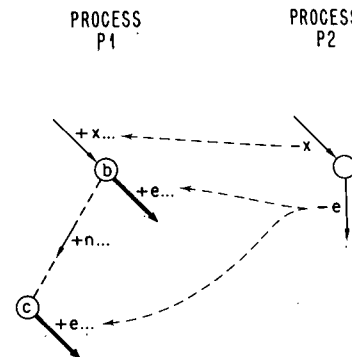


Fig. 11. Derivation of sufficiency proof for Rule 2.

the very simple three-process interaction shown in Fig. 10. Process P_1 transmits message x to process P_2 , P_2 models a nonideal channel from P_1 to P_3 , and P_3 receives messages from P_2 . Message x' (generated by P_2) represents a corruption by the channel of message x , and the arc with identifier 0 represents a nonevent, i.e., a state transition that generates no messages. P_2 is initially in state 0. On receiving message x from P_1 it enters state 1 and can proceed in one of three ways: either it faithfully retransmits x to P_3 by transmitting x or it corrupts x by transmitting x' to P_3 or it loses x by traversing arc 0. Thus, P_3 can either receive message x or a corrupted version x' or no message at all.

APPENDIX II

SUFFICIENCY AND NECESSITY PROOFS FOR THE PRODUCTION RULES

We present arguments which demonstrate that the production rules, derived in Section V-A, are both necessary and sufficient. We say that the rules are sufficient if they create enough arcs to prevent unspecified receptions and that they are necessary if every created arc is needed to prevent unspecified receptions. The proofs assume that arc replication (Section V-B) is replaced by repeated application of the rules. We begin with the sufficiency proof and consider Fig. 11.

1) Assume the rules insufficient and let e be the first message that manifests this, i.e., there exists a state c of P_1 that can receive e yet this reception is not specified by the rules.

2) Consider first the case that $-e$ is appended to a negative arc $-x$, i.e., that Rule 2 causes this unspecified reception. Later, we will consider $-e$ appended to reception arcs.

3) By virtue of 2) and the fact that FIFO channels are assumed, message x is always received before message e .

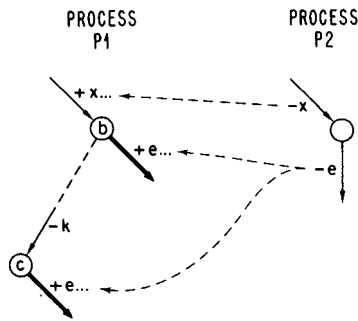


Fig. 12. Derivation of necessity proof for Rule 2.

4) Hence, state c must be below a reception of x ; let b be the entry state of that reception.

5) The path from b to c must contain at least one positive reception arc, say arc $+n$... because otherwise Rule 2 would specify the reception of e in state c .

6) Since $P1$ would receive message n after x and before e , $P2$ must traverse $-x$ followed by $-n$ followed by $-e$.

7) But this contradicts our initial assumption that $-x$ then $-e$ be consecutively traversed.

8) Hence, there is no reception of message e in $P1$ not specified by Rule 2.

We outline the rest of the proof. The above derivation (steps 2-8) is repeated for the case where arc $-e$ is appended to an arc $+x$, i.e., where Rule 1 causes the insufficiency. It is then repeated for the case where arc $-e$ is appended to an arc $+x_s$, i.e., where Rule 3 causes the insufficiency. Since we obtain a contradiction with the assumptions of steps 1 and 2 in all three cases, the rules are sufficient. We now prove with the help of Fig. 12 that the rules are necessary.

1) Assume that the rules overspecify and that e is the first message that manifests this, i.e., there exists a state c in $P1$ that cannot receive message e yet the rules specify this reception.

2) Consider first the case that $-e$ is appended to a negative arc $-x$, i.e., that Rule 2 causes this overspecification. Later, we will consider $-e$ appended to reception arcs.

3) By virtue of 2) and the fact that FIFO channels are assumed, message x is always received before message e .

4) Hence, state c must be below a reception of x ; let b be the entry state of that reception.

5) $P1$ enters state b on receiving x , hence state b can receive e , and Rule 2 specifies a reception of e in state b .

6) $c \neq b$ because otherwise e could be received in state c and the assumptions of 1) would be contradicted.

7) Since $c \neq b$ and Rule 2 specifies reception of message e by state c , there must be a negative-arc sequence connecting state b to c .

8) The entry state of any negative-arc sequence attached to state b can also receive message e (no time constraints assumed).

9) By virtue of 7), state c is the entry state of such a negative-arc sequence, hence state c can receive message e .

10) But this contradicts our initial assumption that state c cannot receive e , hence all receptions of e specified by Rule 2 are occurable.

We outline the rest of the proof. The above derivation (steps 2-10) is repeated for the case where arc $-e$ is appended to an arc $+x$, i.e., where Rule 1 causes the overspecification. It is then repeated for the case where arc $-e$ is appended to an arc $+x_s$, i.e., where Rule 3 causes the overspecification. Consequently, all receptions of e specified by the rules can occur. Hence, the rules are necessary.

APPENDIX III

OUTLINE OF PROOF FOR THE TERMINATION ALGORITHM

We have to prove two facts about ignoring some arcs as described in Section V-D:

1) that it will not cause any arcs to be missed in the protocol, and

2) that it will terminate the building of the trees, provided the channels cannot grow without bounds.

The proofs will only be outlined due to space limitations. For the first point consider a situation when a reception $+e_s$ is added to a tree with entry node i , and assume that the node i is declared dead because of a previous reception $+e'_s$ with entry node i' . Let the entry nodes of the transmission arcs $-e$ and $-e'$ be j and j' , respectively. Consider two executions: one brings the two processes into nodes i and j , the other into nodes i' and j' . There is no way to distinguish between these two executions because the nodes i and i' represent the same process state, j and j' represent the same process state, and the contents of the two channels are also the same (namely, one channel is empty, the other contains the messages represented by the sequences s and s'). Therefore, no matter how the execution from i, j continues, there must be an equivalent execution where the processes are in states i' and j' , respectively. From this, one can prove that for every arc that could possibly be generated (if the design rules were allowed to run forever), there is an equivalent arc attached to an equivalent node generated under the limitations of Section V-D.

To show termination, we will show that no infinite branch can be generated in either tree. For the sake of argument assume an infinite branch. First, this infinite branch must contain an infinite number of receptions, for otherwise there would exist a cycle consisting of transmissions only (see Section VI), contradicting our assumption of bounded channels. Secondly, this infinite branch must contain a dead node because there must be a message whose reception is repeated infinitely often along the branch, but there is only a finite number of nonequivalent combinations of channel contents and entry node. Thus, every branch is either finite or contains a dead node. Therefore, there is only a finite number of transmissions that are both transmitted and received above dead nodes. Keeping a finite number of transmission arcs keeps the trees finite.

ACKNOWLEDGMENT

The authors would like to thank the referees for their helpful and detailed comments.

REFERENCES

- [1] J. Postel, "A graph model analysis of computer communication protocols," Univ California, Los Angeles. Rep. UCLA-ENG-741, Jan. 1974.
- [2] T. Piatkowski, "Finite-state architecture," Syst. Develop. Div. (now Syst. Commun. Div.), Research Triangle Park, NC, IBM Tech. Rep. TR-29.0133, Aug. 1975.
- [3] C. A. Sunshine, "Interprocess communication protocols for computer networks," Ph.D. dissertation, Dep. Comput. Sci., Stanford Univ. Stanford, CA, 1975.
- [4] IBM Corp., "Systems network architecture format and protocol reference manual: Architectural logic," Pub. SC30-3112-1, File S370-30, 1976.
- [5] J. Hajek, "Automatically verified data transfer protocols," in *Proc. Int. Conf. Comput. Commun.*, Kyoto, Japan, Sept. 1978, pp. 749-756.
- [6] G. V. Bochmann, "A general transition model for protocols and services," this issue, pp. 643-650.
- [7] A. Danthine, "Protocol representation: finite state architecture," this issue, pp. 632-643.
- [8] G. D. Schultz, D. B. Rose, C. H. West, and J. P. Gray, "Executable, description and validation of SNA," this issue, pp. 661-677.
- [9] G. V. Bochmann and C. A. Sunshine, "Formal methods in communication protocol design," this issue, pp. 624-631.
- [10] C. A. Sunshine, "Survey of protocol definition and verification techniques," in *Proc. Comput. Network Protocols Symp.*, Liège, Belgium, Feb. 1978.
- [11] P. M. Merlin, "Specification and validation of protocols," *IEEE Trans. Commun.*, vol. COM-27, pp. 1761-1680, Nov. 1979.
- [12] A. Danthine, Ed., in *Proc. Comput. Network Protocols Symp.*, Liège, Belgium, Feb. 1978, see also special issue on Computer Network Protocols, *Comput. Networks*, vol. 2, Sept. Oct. 1978.
- [13] P. Zafiropulo, "Protocol validation by duologue-matrix analysis," *IEEE Trans. Commun.*, vol. COM-26, pp. 1187-1194, Aug. 1978.
- [14] P. Zafiropulo, "Protocol validation by duologue-matrix analysis," IBM Res. Rep. RZ 816, Feb. 1977.
- [15] G. V. Bochmann, "Finite state description of communication protocols," in *Proc. Comput. Network Protocols Symp.*, Liège, Belgium, Feb. 1978.
- [16] C. H. West, "General technique for communications protocol validation," *IBM J. Res. Develop.*, vol. 22, pp. 393-404, July 1978.
- [17] —, "Computer-aided validation of communications protocols," IBM Res. Rep. RZ 817, Feb. 1977.
- [18] J. Hajek, "Protocols verified by APPROVER," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 9, Jan. 1979.
- [19] C. H. West and P. Zafiropulo, "Automated validation of a communications protocol: The CCITT X.21 recommendation," *IBM J. Res. Develop.*, vol. 22, pp. 60-71, Jan. 1978.
- [20] IBM Europe, "Technical improvements to CCITT recommendation X.25," submission to Study Group VII, Oct. 1978.
- [21] P. Zafiropulo, "Design rules for producing logically complete two-process interactions and communications protocols," in *Proc. 2nd Int. Conf. Comput. Software and Applications*, Chicago, IL, pp. 680-685, Nov. 1978.
- [22] K. A. Bartlett, R. A. Scantelbury, and P. T. Wilkinson, "A note on reliable full-duplex transmission over half-duplex links," *Commun. Ass. Comput. Mach.*, vol. 12, pp. 2260-2261, May 1969.
- [23] C. H. West, "An automated technique of communications protocol validation," *IEEE Trans. Commun.*, vol. COM-26, pp. 1271-1275, Aug. 1978.
- [24] H. Rudin, C. H. West, and P. Zafiropulo, "Automated protocol validation: One chain of development," in *Proc. Comput. Network Protocols Symp.*, Liège, Belgium, Feb. 1978.
- [25] D. D. Cowan and C. J. P. Lucena, "Some thoughts on the construction of programs—A data-directed approach," in *Proc. 3rd Jerusalem Conf. Inform. Technol.*, Jerusalem, Israel, Aug. 1978.
- [26] D. D. Cowan, J. W. Graham, J. W. Welch, and C. J. P. Lucena, "A data-directed approach to program construction," *Software Practice and Experience*, to be published.
- [27] D. Belsnes and E. Lynning, "Some problems with the X.25 packet level protocol," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 7, pp. 41-51, Oct. 1977.
- [28] A. A. McKenzie, Ed., *Comput. Commun. Rev.*, vol. 7, Ass. Comput. Mach. Special Interest Group on Data Communications, Oct. 1977.



Pitro Zafiropulo received the B.S. degree in electrical engineering and the Ph.D. degree from the Swiss Federal Institute of Technology, Zurich in 1961 and 1968, respectively.

He is currently a member of the Data Networks Group at the IBM Zurich Laboratory, Rüschlikon, Switzerland, working in the area of protocol validation. He joined IBM in 1968 and has worked in the areas of speech recognition, in-house communications, and data networks. In 1962 he spent one year at the European Organization for Nuclear Research (CERN) in Geneva, Switzerland, working in the areas of spark and Wilson chambers. In 1963 he joined the Hasler Stiftung Research Laboratory and worked on pulse-code-modulated transmission. In 1973 he received an IBM Outstanding Contribution Award for his work in the area of digital PABX's. In 1978 he received an IBM Outstanding Innovation Award for his work in the area of protocol validation.



Colin H. West received the B.S. degree in physics in 1960 and the Ph.D. degree in elementary particle physics in 1965, both from Imperial College, London, England.

He joined the IBM Zurich Laboratory in 1971 and has worked on laboratory automation, computer graphics, communications, and computer networks. He is currently working on the further development of communications protocol validation. From 1961 to 1966 he was a visiting Scientist at the European organization for Nuclear Research (CERN) in Geneva, Switzerland, and subsequently held postdoctoral positions in the Department of Physics and in the Moore School of Electrical Engineering of the University of Pennsylvania Philadelphia. He has received an IBM Outstanding Innovation Award for his work on the automated validation of communications protocols.

Dr. West is a member of the American Physical Society.



Harry Rudin (S'55-M'62) received the B.E., M.E., and D. Eng. degrees from Yale University, New Haven, CT, in 1958, 1960, and 1964, respectively.

From 1961 to 1964 he served as Instructor in Electrical Engineering at Yale. In 1964 he joined Bell Telephone Laboratories where he worked in the area of data communications, mainly on automatic equalization techniques. He has been with the IBM Zurich Research Laboratory, Rüschlikon, Switzerland since 1968, where he has worked on computer communications systems. Here his activities first centered on traffic theoretic aspects of information flow in computer networks, particularly on the problems of dynamic multiplexing, network dimensioning, routing, and flow control. Recently his activities have been on the formal specification of the protocols which define interprocess interaction in computer communication systems, particularly with their automatic verification and computer-supported design.

Dr. Rudin is an Editor of the IEEE TRANSACTIONS ON COMMUNICATIONS, a correspondent for IEEE COMMUNICATIONS MAGAZINE, and active on a number of IEEE Communications Society committees.



D. D. Cowan (S'58-M'60) was born in Toronto, Ont., Canada, in March 1938. He received the B.A.Sc. degree in engineering physics from the University of Toronto in 1960 and the M.Sc. and Ph.D. degrees in applied mathematics from the University of Waterloo in 1961 and 1965, respectively.

He has been on the faculty of the University of Waterloo since 1962 and was the Chairman of the Computer Science Department from 1967 to 1972 and Associate Dean of the Faculty of Mathematics from 1974 to 1978. He is currently a Professor of Computer Science. He is also involved with a number of programs of cooperation between Computer Science Departments in South America and the University of Waterloo.

His research interests include computer communications, programming constructs and methods, and software engineering. He spent the academic year 1978-1979 at the IBM Zurich Research Laboratory, Rüschlikon, Switzerland, where he participated in the research work reported in this paper.

Dr. Cowan is a member of the Association for Computing Machinery and the IEEE Computer Society.



Daniel Brand was born in Prague, Czechoslovakia, in 1949. He received the Ph.D. degree in computer science from the University of Toronto, Toronto, Ont., Canada, in 1976.

Since then he has been working at the IBM Thomas J. Watson Research Center, Yorktown Heights, NY, as a member of the microprogram verification group. He is currently spending one year at the IBM Zurich Research Laboratory, Rüschlikon, Switzerland. His research interests include software reliability, protocol verification, and automatic theorem proving.

Dr. Brand is a member of the Association for Computing Machinery.

Executable Description and Validation of SNA

GARY D. SCHULTZ, DAVID B. ROSE, C. H. WEST, AND JAMES P. GRAY

(Invited Paper)

Abstract—The definition of IBM's Systems Network Architecture (SNA) has evolved into a specification of a node in the form of a meta-implementation using formal, state-oriented descriptive techniques. This evolution is traced here, and the different formal techniques are described. The culmination of this process has been the development of a PL/I-based programming language, Format and Protocol Language (FAPL), as a descriptive tool. Using FAPL, the architects now define SNA by a programmed meta-implementation of a node.

In this form, it is precise, readily accessible to the implementing product designers and programmers, and structurally close to the implementations. The essential features of the meta-implementation and of FAPL are described, along with the implications and advantages of describing the architecture in an executable form. One major benefit, already being realized, is the capability to test the logical consistency and completeness of the executable description itself. The current status of the validation of the executable description and sample results obtained are described.

I. INTRODUCTION

THE 1960's and early 1970's were the design heyday and proving ground for operating systems within single computers and across tightly coupled ones. Today we are experi-

encing a new design era for coordinating data processing distributed over ensembles of cooperating processors, configured into networks.

Software engineering for operating systems developed layered structuring of systems, top-down design, structured programming, disciplined synchronization (e.g., semaphores) for cooperating processes, and research into proof-of-program-correctness methods. Today's era of *network architectures*, which are specifications of the message formats and interaction protocols for services provided within networks, has had the need for additional design innovations for the changed system context of loosely coupled system components, disparate processor architectures, and widely dispersed groups of people implementing a common network architecture.

This paper focuses on the evolving specification of IBM's Systems Network Architecture (SNA) and the formal techniques developed to design, describe, and test it. A survey of the flourishing literature on other formal techniques, developed independently of those described here, is outside the scope of this paper. We refer the reader to Sunshine's extensive survey [1] and other papers in this issue for discussions of parallel advances.

The next section presents a brief overview of SNA. Section III discusses the evolution of the architectural description of SNA into a state-oriented meta-implementation, and the

Manuscript received May 7, 1979; revised January 28, 1980.

G. D. Schultz, D. B. Rose, and J. P. Gray are with the IBM Corporation, Research Triangle Park, NC 27709.

C. H. West is with the IBM Zurich Research Laboratory, Rüschlikon, Switzerland.

0090-6778/80/0400-0661\$00.75 © 1980 IEEE