During the final quarter (October 1, 1986 through December 31, 1986) of this contract research progressed and several papers were published. Investigations to determine when system state analysis of the shared variable protocol model is adequate for complete error analysis were continued. This was done by devising a number of example protocols to test various hypotheses for constraints on the model. Final results of this investigation are still incomplete but the research continues to hone in on constraints that should provide a firm theoretical (but yet practical) result.

R.E. Miller also met with Mr. D. Otis Wolkins, GTE Vice President for Quality Services, on November 13, 1986 and subsequently phoned him on December 19, 1986. In these discussions Miller provided Wolkins with a proposal for continued research, with the hope of obtaining restoration of funding for the second and third years as originally promised by GTE. This restoration, at approximately $50,000 per year for calendar years 1987 and 1988, would enable this research on protocols to continue under GTE sponsorship. In both conversations Mr. Wolkins noted that he would look into this matter and notify Miller.

Recently published papers, partially supported by this contract, are attached. They are:


Protocol Analysis and Synthesis by Structured Partitions *

Tat Y. CHOI **
GTE Laboratories, 40 Sylvan Road, Waltham, MA 02254, USA

Raymond E. MILLER ***
School of Information and Computer Science, Georgia Institute of Technology, Atlanta, GA 30332, USA

Protocols can be viewed as predefined sequences of message exchanges between machines for performing network control functions and for providing network services. One way of modeling protocols is by using communicating finite state machines. The interaction between finite state machines can be very involved, even for few states. To counteract the complexity for protocol analysis and synthesis, we propose a partitioning method based on protocol subgraphs. We found that if there are 'cross interactions' involving the entire protocol graph, then the protocol is not decomposable by our technique and must be analyzed or synthesized as a whole. However, if there are subunits of message exchanges within the protocol graph that are self-contained, in other words, if there are no 'cross interactions' between protocol subgraphs, then the protocol is decomposable. For protocols that are decomposable, we show that it is only necessary to examine a subspace of the entire protocol graph to understand the behavior of the protocol and to guarantee its progress properties. This allows us to analyze and synthesize protocols based on these subgraphs.

Keywords: Network Protocols, Communicating Finite State Machines, Analysis, Synthesis, Structured Partitions, Progress Properties

1. Introduction

Communicating finite state machines have been successfully used in modeling many complex protocols [1–4]. Reachability analyses of these protocols often run into global state explosion problems, even when the individual machines contain relatively few states [5]. In this paper, we study communicating finite state machines by looking for decompositions of the state transition graphs representing these machines. For a two party protocol, the pair of state transition graphs is referred to as a protocol graph. By partitioning the state transition graphs into subgraphs, we show how it is possible to decompose a protocol graph into smaller protocol subgraphs. Each protocol subgraph corresponds to a phase or subphase of the protocol.

Under the condition of no 'cross interactions' between subgraphs, we develop a relationship between the reachability spaces of the protocol subgraphs and that of the entire protocol graph.

Tat Y. Choi received the B.S. degree from the University of Illinois, M.S. degree from Brown University, and the Ph.D. degree from Georgia Institute of Technology, all in Electrical Engineering. At present, he is with the Telecommunication Research Laboratories at GTE Laboratories. His current research interests are in network control algorithms, integrated service networks, protocol analysis and construction methods, and parallel processor architectures for communication processing. He is a member of IEEE and ACM.

* Technical Report GIT-ICS-86/18
** This work was carried out while the author was at Georgia Institute of Technology and revised at GTE Laboratories
*** This work was supported in part by National Science Foundation Grant MCS-8103608.

North Holland
Computer Networks and ISDN Systems 11 (1986) 367-381
From this relationship, we are able to infer that if the component protocol subgraphs have the required progress properties, then the entire protocol graph also has the required progress properties. This technique then can be used to simplify the analysis and synthesis of protocols.

In section 2, we present the protocol model we use. In section 3, we introduce the 'structured partition' of a protocol graph and study the properties of these partitions. In section 4, we discuss the different kinds of interactions that can occur between protocol subgraphs, the relationship between the reachability spaces of the protocol subgraphs and that of the entire protocol graph, and how one can infer the progress properties of a protocol graph from that of its component subgraphs. In section 5, we describe a protocol analysis and synthesis method based on structured partitions. In the conclusion, we present a summary of our work and its relation to other work. Also, we discuss some areas for further study.

2. The Finite State Model for Protocols

2.1. The Protocol Model

A protocol system consists of processes P₁ and P₂ interconnected by communication channels C₁₂ and C₂₁. See Figure 1. In the finite state model, processes are represented by finite state machines which send and receive messages [6,7]. A commonly used model for protocols is one based on the work of Zafiropulo et. al. [8,9]. In this model, a protocol is defined as follows:

2.1.1. Notation

(i) Throughout this paper, we will be dealing with two processes P₁ and P₂, two channels C₁₂ and C₂₁, and entities associated with these two processes. For convenience, we will use subscripts i and j to denote these two processes and let i, j ∈ {1, 2} and i ≠ j. This will be assumed to be the conditions for i and j unless specified otherwise.

(ii) We use x, y and z to denote messages and X, Y and Z to denote message sequences. We use justaposition to represent the concatenation of message sequences. For example, xY represents the concatenation of message x with sequence Y.

(iii) we use λ to represent the empty sequence.

2.1.2. Process Description

**Definition:** A protocol machine PM = (S, M, 0, τ) where

(i) S = {S₁, S₂}: S₁ and S₂ represent the sets of states of processes P₁ and P₂ respectively.

(ii) M = {M₁₂, M₂₁}: M₁₂ represents the set of messages that can be sent from process P₁ to P₂, and M₂₁ represents the set of messages that can be sent from process P₂ to P₁.

(iii) O = (o₁, o₂): o₁ and o₂ are the initial states for processes P₁ and P₂ respectively.

(iv) τ, a partially defined transition function: Sᵢ × Σᵢ → Sᵢ, where

Σᵢ = {−x | x ∈ Mᵢ₁} ∪ {+y | y ∈ Mᵢ₂}

with −x representing the sending of message x and +y representing the receiving of message y.

For processes which contain final states, we define a set F = (F₁, F₂) where F₁ and F₂ are the sets of final states for processes P₁ and P₂ respectively.

We can extend the transition function of Pᵢ to sequences in the normal fashion, by letting

τ(s, T) = τ(τ(s, o), T')

where T = oT' and T' is a sequence of events from Σᵢ.

2.1.3. Channel Description

In many communication systems, a first-in-first-out discipline is enforced on the sending and receiving of messages. Thus the channels can be modeled as FIFO queues.

**Definition:** A channel Cᵢⱼ is a FIFO queue connecting process Pᵢ to process Pⱼ. The contents of Cᵢⱼ is labeled cᵢⱼ; it is a sequence of messages x ∈ Mᵢ₁. If Cᵢⱼ is empty, we designate this as cᵢⱼ = λ.
The capacity of a channel can be finite or infinite. For finite capacity channels, the maximum capacity of a channel \( C_{ij} \) is given by \( k_{ij} \). In reality, all practical protocols utilize finite capacity channels. In this paper, we treat the channels as ideal. Non-ideal channels can be modeled by the introduction of channel events. These channel events can be used to model the loss, duplication or resequencing of messages in the channel.

2.1.4. Dynamics of Protocol Systems

The dynamics of the protocol system can be described by global states and transitions between global states.

**Definition:** A global state is a pair \( (S, C) \) where \( S = (s_1, s_2) \) with \( s_1 \) and \( s_2 \) representing the current states of processes \( P_1 \) and \( P_2 \) respectively and \( C = (c_{12}, c_{21}) \) with \( c_{12} \) and \( c_{21} \) representing the current contents of channels \( C_{12} \) and \( C_{21} \) respectively.

For processes which contain final states, a final global state is defined as follows:

**Definition:** A final global state \( (S_f, C_f) \) is given by \( ((f_1, f_2), (\lambda, \lambda)) \) where \( f_1 \) and \( f_2 \) are final states of processes \( P_1 \) and \( P_2 \) respectively.

The transitions between global states can be defined by a precedence relation \( \prec \) between global states.

**Definition:** A global state \( (S, C) \prec (S', C') \) if and only if there exists \( i, j \) and \( c_{ij} \) such that the elements of \( (S', C') \) can be derived from \( (S, C) \) by applying one of the following execution rules:

1. \( s' = \tau(s, -x) \) and \( c_{ij} = c_{ij}^x \)
2. \( s' = \tau(s, +x) \) and \( c_{ij} = xc_{ij}^x \)

Except for the elements affected by the one execution rule applied, all other elements of \( (S', C') \) are equal to the elements of \( (S, C) \).

**Definition:** A global state \( (S, C) \) is said to be reachable from the initial global state \( (S_0, C_0) \) if and only if \( (S_0, C_0) \prec \bullet (S, C) \) where

\[
(S_0, C_0) = ((o_1, o_2), (\lambda, \lambda))
\]

and \( \bullet \) denotes the reflexive, transitive closure of \( \prec \).

2.2. Progress Properties for Protocols

The progress properties of interest can be stated in terms of the error conditions that may arise in the execution of the protocol. A protocol that is error free during execution is said to have the required progress properties.

Before stating these error conditions, several definitions are in order.

**Definition:** A transition at state \( s \) for event \( \sigma \) is specified if and only if \( \tau(s, \sigma) \) is defined.

**Definition:** A transition at state \( s \) for event \( \sigma \) is executable if and only if there exists a reachable global state

\[
((s_1, s_2), (c_{12}, c_{21}))
\]

with \( s = s_1 \) or \( s = s_2 \) and in addition, if \( \sigma \) is a receive transition \(+x\) then for \( s = s_1, c_{21} = xY \) for some sequence \( Y \) and for \( s = s_2, c_{12} = xZ \) for some sequence \( Z \).

Error conditions called unspecified reception, nonexecutable transition, and state deadlock can be defined as follows:

**Definition:** A reception for message \( x \) at state \( s \) is called an unspecified reception if and only if there is a reachable global state \( ((s_1, s_2), (c_{12}, c_{21})) \) with (i) \( s = s_1 \) and \( c_{21} = xY \) for some sequence \( Y \) and both \( \tau(s_1, +x) \) and \( \tau(s_1, -x) \) are unspecified for any message \( x \) or (ii) if \( s = s_2 \) and \( c_{12} = xZ \) for some sequence \( Z \) and both \( \tau(s_2, +x) \) and \( \tau(s_2, -x) \) are unspecified for any message \( x \).

**Definition:** A transition \( \tau(s, \sigma) \) is nonexecutable if and only if it is specified and not executable.

**Definition:** A state deadlock is a global state \( S = ((s_1, s_2), (\lambda, \lambda)) \) where both \( \tau(s_1, -x) \) and \( \tau(s_2, -x) \) are unspecified for any \( x \) and \( S \) is not a final global state.

**Definition:** A protocol is said to have the required progress properties if it does not contain any unspecified receptions, nonexecutable transitions or state deadlocks.

Protocols can be validated using reachability analysis [10–12]. In this method, we analyze the behavior of a protocol by constructing a reachability graph. The reachability graph is a graph whose nodes are labeled with global states and whose edges correspond to transitions. It can be constructed by successively generating all global states that are reachable from the initial global state. For finite capacity channels, the reachability graph is always finite. For infinite capacity channels, the reachability graph can be infinite. It has been shown that the problem of deciding whether a protocol possesses specific properties is, in general, unsolvable [8]. To circumvent this problem,
we divide protocols into two classes, those that contain finite reachability graphs and those for which the reachability graph is infinite. For the class of protocol that contain infinite reachability graphs, a restricted solution to the problem can be obtained by introducing a channel bound $k_{ij}$ as a parameter of validation. Then, a validated protocol is guaranteed to have the required progress properties up to this bound by generating the reachability graph only up to global states in which the maximum number of messages in each channel is $k_{ij}$.

3. The Structured Partition of a Protocol Graph

In the following sections we use state transition graphs to represent finite state machines. Let $G_i$ be the state transition graph representing process $P_i$. The node which is labeled with the initial state of $P_i$ is referred to as the initial node of $G_i$. For the purpose of protocol decomposition, we define a structured partition $\pi$ of $G_i$ into subgraphs as follows:

**Definition:** A node $v$ of a subgraph is a header node $h$ if and only if it is the first node encountered by all paths joining nodes outside the subgraph to nodes belonging to the subgraph.

**Definition:** A node $v$ of a subgraph is an exit node $e$ if and only if it has a successor node which does not belong to the same subgraph.

**Definition:** A partition $\pi = \{G_1, \ldots, G_n\}$ of graph $G_i$ into subgraphs is a Structured Partition if and only if

(i) The initial node of $G_i$ is a header node.
(ii) Every subgraph $G_k$ belonging to $\pi$ contains a header node $h$.

By definition, the exit nodes of a subgraph are always connected to the header nodes of other subgraphs. A header node can also serve as an exit node. Figure 2 shows a structured partition $\pi$ of graph $G_i$ into $n$ subgraphs. The minimal subgraph is a single node, in which case the single node acts as both a header and exit node.

The structured partition $\pi$ of a graph $G_i$ is not unique. Let $SP_i = \{\pi_k\}$ be the set of structured partitions of $G_i$.

**Definition:** A partition $\pi_k$ of graph $G_i$ is smaller than or equal to partition $\pi_j$, denoted by $\pi_k \leq \pi_j$, if and only if every subgraph belonging to $\pi_k$ is contained in a subgraph belonging to $\pi_j$.

**Lemma 3.1.** The set of all structured partitions $SP_i = \{\pi_k\}$ of graph $G_i$ forms a partially ordered set.

**Proof:** Let $\pi_p$, $\pi_q$, $\pi_r$ be structured partitions of $G_i$. The binary relation $\leq$ on $SP_i$ is a partial ordering of $SP_i$ since it is:
(i) reflexive: $\pi_p \leq \pi_p$ for all $\pi_p$ in $SP_i$
(ii) antisymmetric: $\pi_p \leq \pi_q$ and $\pi_q \leq \pi_p$ implies $\pi_p = \pi_q$
(iii) transitive: $\pi_p \leq \pi_q$ and $\pi_q \leq \pi_r$ implies $\pi_p \leq \pi_r$.

**Lemma 3.2.** The largest structured partition $\pi(1)$ of graph $G_i$ is one that has a single block that includes the entire graph $G_i$.

**Proof:** The entire graph $G_i$ satisfies the structured requirements since $G_i$ has an initial node which acts as the header node $h$. $\pi(1)$ is the largest structured partition since $\pi_k \leq \pi(1)$ for every $\pi_k$ belonging to $SP_i$.

**Lemma 3.3.** The smallest structured partition $\pi(0)$ of graph $G_i$ is one in which every node of $G_i$ is a subgraph.
I. Example 3.1.

Consider the graph \( G = \{v_1, v_2, v_3, v_4\} \) shown in Figure 3a. The partially ordered set of structured partitions \( SP = \{\pi(0), \pi_1, \pi_2, \pi_3, \pi_4, \pi(I)\} \) of graph \( G \) is shown in Figure 3b where:

\[
\pi(0) = \{\overline{v_1}, \overline{v_2}, \overline{v_3}, \overline{v_4}\},
\]

\[
\pi_1 = \{v_1, \overline{v_2}, \overline{v_3}, \overline{v_4}\},
\]

\[
\pi_2 = \{\overline{v_1}, v_2, \overline{v_3}, \overline{v_4}\},
\]

\[
\pi_3 = \{\overline{v_1}, \overline{v_2}, v_3, \overline{v_4}\},
\]

\[
\pi_4 = \{\overline{v_1}, \overline{v_2}, \overline{v_3}, v_4\},
\]

\[
\pi(I) = \{\overline{v_1}, v_2, v_3, \overline{v_4}\}.
\]

An alternate way of representing a protocol machine \( PM \) is using a protocol graph \( PG \).

**Definition:** A Protocol Graph \( PG(G_1, G_2) \) consists of two state transition graphs \( G_1 \) and \( G_2 \) representing processes \( P_1 \) and \( P_2 \) respectively.

### 4. A Structure Theory for Communicating Finite State Machines

Consider the protocol graph \( PG(G_1, G_2) \). Let \( G_1 \) be partitioned into \( m \) subgraphs \( \pi = \{G_1, \ldots, G_m\} \) and \( G_2 \) be partitioned into \( n \) subgraphs \( \pi' = \{G'_1, \ldots, G'_n\} \). Let \( G_1 \) and \( G'_1 \) be the initial subgraphs for \( G_1 \) and \( G_2 \) respectively, where they contain the initial nodes for each graph. The protocol graph \( PG(G_1, G_2) \) can be converted into a protocol tree \( PT(T_1, T_2) \) by converting graphs \( G_1 \) and \( G_2 \) into trees \( T_1 \) and \( T_2 \) respectively. Each node of tree \( T_i \) is labeled with a subgraph of \( G_i \). The root node of \( T_i \) is labeled with the initial subgraph of \( G_i \). Each immediate descendent to the node labeled with subgraph \( G_k \) or \( G'_k \) in tree \( T_i \) represents an immediate successor subgraph to \( G_k \) or \( G'_k \) in graph \( G_i \). Let \( G_k \) be a subgraph belonging to the \( k^{th} \) level of tree \( T_1 \) and \( G'_k \) be a subgraph belonging to the \( k^{th} \) level of tree \( T_2 \). Here, we assume that when process \( P_1 \) is in subgraph \( G_k \), process \( P_2 \) is in subgraph \( G'_k \).
Table 1
Four Regions of Interaction.

<table>
<thead>
<tr>
<th>Regions of Interaction</th>
<th>Process P_1</th>
<th>Process P_2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Region 1</td>
<td>(G_k \cup h)</td>
<td>(G_k \cup h')</td>
</tr>
<tr>
<td>Region 2</td>
<td>(G_i)</td>
<td>(G'_i)</td>
</tr>
<tr>
<td>Region 3</td>
<td>(G_k \cup h)</td>
<td>(G'_i)</td>
</tr>
<tr>
<td>Region 4</td>
<td>(G_i)</td>
<td>(G_k \cup h')</td>
</tr>
</tbody>
</table>

4.1. Single Successor Case

Consider the case where \(G_k\) has only one successor subgraph \(G_i\) and \(G'_i\) has only one successor subgraph \(G'_i\). See Figure 4. Let \(G_i \cup h\) denote the union of \(G_k\) with the header node \(h\) of \(G_i\) and \(G'_i \cup h'\) denote the union of \(G'_i\) with the header node \(h'\) of \(G'_i\). Between subgraphs \(G_i \cup G_i\) and \(G'_i \cup G'_i\), there are four regions of interaction. See Table 1. We can distinguish between two kinds of interactions.

(i) Expected Interactions

In both regions 1 and 2, while process \(P_1\) is in subgraph \(G_i \cup h\) or \(G_i\), process \(P_2\) is in subgraph \(G'_i \cup h'\) or \(G'_i\), respectively. When they exchange messages in these subgraphs, we refer to them as ‘expected interactions’ between subgraphs.

(ii) Cross Interactions

In regions 3 and 4, while process \(P_1\) is in subgraph \(G_i \cup h\) or \(G_i\), process \(P_2\) is in subgraph \(G'_i \cup h'\) or \(G'_i\), respectively. This can happen because both processes execute asynchronously and one process can execute multiple transitions and enter into its successor subgraph while the other process lags behind. By itself, this does not signify that a process in one subgraph will exchange messages with the other process in its predecessor or successor subgraph. However, if they do so, we refer to this as a ‘cross interaction’ between subgraphs. The following example shows a cross interaction.

Example

Let process \(P_1\) be in state \(p\) of subgraph \(G_i \cup h\) while process \(P_2\) is in state \(p'\) of subgraph \(G'_i \cup h'\) and both channels are empty. See Figure 5. Process \(P_2\) can make the transition \(-x_1\) and reach state \(q'\) of subgraph \(G'_i \cup h'\). \(P_1\) can then make the sequence of transitions \((-x_2, +x_1}\) and reach state \(q\) of subgraph \(G_i \cup h\). Following this, \(P_2\) can perform \(+x_2\) and reach state \(h'\) of \(G'_i\). At this point, \(P_1\) in state \(q\) of \(G_i \cup h\) can send message \(y\) and \(P_2\) in state \(h'\) of \(G'_i\) can receive message \(y\). This is an example of a cross interaction while the processes are in region 3.

Proposition 4.1. If the protocol graph \(PG(G_i \cup h, G'_i \cup h')\) has the required progress properties, then all the messages sent by process \(P_1\) in subgraph \(G_i \cup h\) will be received by process \(P_2\) in subgraph \(G'_i \cup h'\) and vice versa, i.e., there can be no ‘cross interaction’ between the two subgraphs.

Proof: The proof is by contradiction. Assume that cross interaction exists between the two subgraphs, then the protocol graph \(PG(G_i \cup h, G'_i \cup h')\) must contain either unspecified receptions or state deadlocks or both.

If there are no cross interactions, the reachabil-
The reachability graph of $G(G_k U G, G_k U G')$ is the union of the reachability graphs of $G(G_k U G, G_k U G')$ and $G(G, G)$ plus a transition region. See Figure 6. The two reachability graphs have a common node, the global state $((h, h'), (\lambda, \lambda))$. The transition region represents intermediate global states where one process executes multiple transitions and enters into subgraph $G_i$ or $G_i'$, while the other process lags behind in $G_k U G'$ or $G_k U G$ respectively. It is generated as a result of connecting subgraph $G_i$ to $G_k U G$ and $G_i'$ to $G_k U G'$. For every sequence of transitions through the transition region, there is an equivalent sequence through the union of the reachability graph of $G(G_k U G, G_k U G')$ and that of $G(G, G)$. By equivalent sequences, we mean that the two sequences contain the same set of events except for the order in which they occur [16]. In cases where both $P_1$ and $P_2$ cannot race ahead, the transition region is absent.

From the relationship between the reachability graphs, we have the following lemma.

**Lemma 4.1.** If the protocol subgraphs $PG(G_k U G, G_k U G')$ and $PG(G, G)$ have the required progress properties, then the protocol graph $PG(G_k U G, G_k U G')$ also has the required progress properties.

**Proof.**
(i) On the progress property w.r.t unspecified receptions: For every transition sequence $T'$ through the transition region, there is an equivalent sequence $T$ through the union of the reachability graphs of $PG(G_k U G, G_k U G')$ and $PG(G, G)$. Since there are no unspecified receptions for the sequence $T$, there should not be any unspecified receptions for the sequence $T'$.

(ii) On the progress property w.r.t nonexecutable transitions:
Let $\Sigma$ and $\Sigma'$ be the sets of transitions specified in the protocol graphs $PG(G_k U G, G_k U G')$ and $PG(G, G)$ respectively. For the protocol graph $PG(G_k U G, G_k U G')$, the set of specified transitions is $\Sigma U \Sigma'$. Since the reachability graph of $PG(G_k U G, G_k U G')$ is the union of reachability graphs of $PG(G, G)$ and $PG(G_k U G, G_k U G')$ plus a transition region, every transition $\tau \in \Sigma U \Sigma'$ must exist in the reachability graph of $PG(G_k U G, G_k U G')$.

(iii) On the progress property w.r.t state deadlocks: The argument is similar to unspecified transactions.

From the relationship between the reachability graphs, we have the following lemma.
receptions except in this case since there are no
state deadlocks for sequence \( T \), there should not
be any state deadlocks for sequence \( T' \).

The converse, however, is not necessarily true.

**Lemma 4.2.** If the protocol graph \( PG(G_1 \cup G_2, G'_1 \cup G'_2) \) has the required progress properties, it does not follow that the protocol graph \( PG(G_1 \cup h, G'_1 \cup h') \) or \( PG(G_2, G'_2) \) has the required progress properties.

**Proof:** The proof is by counterexample. Consider the protocol graph \( PG(G_1 \cup G_2, G'_1 \cup G'_2) \) shown in Figure 7. The subgraphs are \( G_1 \cup h = (s_1, s_2, s_3) \) and \( G_2 = (s_3, s_4) \) for process \( P_1 \). Similarly, we have \( G'_1 \cup h' = (s_1, s_3, s_5) \) and \( G'_2 = (s_3, s_4) \) for process \( P_2 \). The protocol graph \( PG(G_1 \cup G_2, G'_1 \cup G'_2) \) has the required progress properties while the protocol graph \( PG(G_1 \cup h, G'_1 \cup h') \) contains an unspecified reception for message \( x \) in the global state \( (s_3, s_1, (\lambda, x, x, x)) \). This message can be received by process \( P_1 \) in subgraph \( G_2 \).

4.2. Multiple Successor Case

Similar arguments can be applied for the case of multiple successor subgraphs. Let \( \{ G_p \} \) be the set of successor subgraphs to \( G_k \) and \( \{ G'_q \} \) be the set of successor subgraphs to \( G'_k \). Let \( H_k = \{ h_p \} \) be the set of header nodes of successor subgraphs \( G_p \) and \( H'_k = \{ h'_q \} \) be the set of header nodes of successor subgraphs \( G'_q \). Let \( G_k \cup H_k \) and \( G'_k \cup H'_k \) denote the union of subgraph \( G_k \) and \( G'_k \) with the set of header nodes \( H_k \) and \( H'_k \). Let \( G_{np} = \bigcup_{p=1}^{m} G_p \) and \( G'_{nq} = \bigcup_{q=1}^{n} G'_q \). The following two lemmas apply for the case of multiple successor subgraphs.

**Lemma 4.3.** If the protocol subgraph \( PG(G_k \cup H_k, G'_k \cup H'_k) \) has the required progress properties and for every reachable global state \( (h_p, h'_q), (\lambda, \lambda) \) of \( PG(G_k \cup H_k, G'_k \cup H'_k) \), the protocol subgraph \( PG(G_p, G'_q) \) also has the required progress properties, then the protocol graph \( PG(G_k \cup G_{np}, G'_k \cup G'_{nq}) \) has the required progress properties.

**Proof:** The reachability graph of \( PG(G_k \cup G_{np}, G'_k \cup G'_{nq}) \) is the union of the reachability graph of \( PG(G_k \cup H_k, G'_k \cup H'_k) \) and the reachability graphs of \( PG(G_p, G'_q) \) plus a set of transition regions. The argument for the progress properties is similar to Lemma 4.1.

**Lemma 4.4.** If the protocol graph \( PG(G_k \cup G_{np}, G'_k \cup G'_{nq}) \) has the required progress properties, it does not follow that the protocol subgraphs \( PG(G_k \cup H_k, G'_k \cup H'_k) \) or \( PG(G_p, G'_q) \) have the required progress properties, where \( (G_p, G'_q) \) is any pair of reachable successor subgraphs to \( (G_k \cup H_k, G'_k \cup H'_k) \).

**Proof:** Consider the case where \( m = n = 1 \). The counter example in Lemma 4.2 applies.

4.3. The Entire Protocol Graph

Finally, the situation can be generalized to include the entire protocol graph. In this case, there is the possibility of a process \( P \) racing ahead by more than one successor subgraph. The reachability graph of the entire protocol graph will therefore include additional transition regions resulting from such transition sequences. However, for the purpose of analysis, it is sufficient to consider only one level of successor subgraph at a time. This is because the racing ahead of a process \( P \) through \( n \) levels of successor subgraphs can be treated as the racing ahead through \( n-1 \) levels, if we consider the first two levels of subgraphs as a single level. Similarly, the racing ahead through \( n-1 \) levels of successor subgraphs can be treated as the racing ahead through \( n-2 \) levels if we consider the first two levels of the remaining \( n-1 \) levels of successor subgraph as a single level. This reduction in the levels of successor subgraphs continues until there is only one level of successor subgraph.

**Theorem 4.1.** Given a protocol graph \( PG(G_1, G_2) \) where \( G_1 \) can be partitioned into \( m \) subgraphs and \( G_2 \) can be partitioned into \( n \) subgraphs, the protocol graph \( PG(G_1, G_2) \) has the required progress properties if:

(i) the initial protocol subgraph \( PG(G_1 \cup H_1, G'_1 \cup H'_1) \) has the required progress properties,

(ii) if the protocol subgraph \( PG(G_k \cup H_k, G'_k \cup H'_k) \) has the required progress properties and \( (h_p, h'_q), (\lambda, \lambda) \) is a reachable global state of \( PG(G_k \cup H_k, G'_k \cup H'_k) \), then the protocol graph...
Fig. 8. A Non-Decomposable Protocol.

amount of coupling or cross interaction between different parts of the protocol graph. If there are few cross interactions between subgraphs, then the protocol is decomposable. However, if there exists cross interactions involving the entire graph, then the protocol is not decomposable. In the following, we present examples of a non-decomposable and a decomposable protocol.

Example 5.1

Consider the protocol graph $PG(G_1, G_2)$ given in Figure 7 which is reproduced in Figure 8 for easy reference. Since graphs $G_1$ and $G_2$ are isomorphic, the sets of structured partitions $SP_1$ and $SP_2$ are identical. From Example 3.1, $SP_1 = \{ \pi(0), \pi_1, \pi_2, \pi_3, \pi_4, \pi(1) \}$. The partition for which all the protocol subgraphs have the required progress properties is $\pi(1) = \{ \overline{v_1, v_2, v_3, v_4} \}$ and there is no $\pi_1 < \pi(1)$ for which all the protocol subgraphs belonging to $\pi_1$ have the required progress properties. Hence, the protocol graph $PG(G_1, G_2)$ is non-decomposable.

Example 5.2

Consider the protocol graph $PG(G_1, G_2)$ shown in Figure 9a. Similar to Example 5.1, the sets of structured partition $SP_1$ and $SP_2$ are identical and $SP_1 = \{ \pi(0), \pi_1, \pi_2, \pi_3, \pi_4, \pi(1) \}$. The smallest partition for which all the protocol subgraphs have the required progress properties is $\pi(0) = \{ \overline{v_1, v_2, v_3, v_4} \}$. The protocol subgraphs for partition $\pi(0)$ are shown in Figure 9b. In fact, the protocol subgraphs belonging to all partitions $\pi_1 > \pi(0)$ also have the required progress properties. The protocol graph $PG(G_1, G_2)$ is an example of a decomposable protocol.
5.2. Protocol Analysis and Synthesis

In this section we present a method for protocol analysis and synthesis based on structured partitions. We adopt the following notation:

\[ \pi(0) = \text{the smallest structured partition of graph } G_1 \]

\[ \pi_k = \text{a structured partition of graph } G_1 \]

\[ \pi'_k = \text{a structured partition of graph } G_2 \]

If the protocol graph \( PG(G_1, G_2) \) is decomposable, then there exists minimal structured partitions \( \pi_{min} < \pi(I) \) and \( \pi'_{min} < \pi'(I) \) such that all the protocol subgraphs \( PG(G_1, G_2) \) of \( \pi_{min} \) and \( \pi'_{min} \) have the required progress properties according to Theorem 4.1. Partitions \( \pi_{min} \) and \( \pi'_{min} \) are minimal if there are no \( \pi_k < \pi_{min} \) and \( \pi'_k < \pi'_{min} \) such that the above statement is true. For protocol analysis by structured partitions, we wish to find \( \pi_{min} \) and \( \pi'_{min} \).

In the following, we present a method for finding \( \pi_{min} \) and \( \pi'_{min} \) by searching \( SP_1 \) and \( SP_2 \). We begin our analysis with the protocol subgraphs of \( \pi(0) \) and \( \pi'(0) \) and whenever we detect the possibility for cross interactions between subgraphs, we construct the immediate successors to \( \pi(0) \) and/or \( \pi'(0) \) until the cross interactions are eliminated. This process of forming larger and larger partitions continues until we arrive at \( \pi_{min} \) and \( \pi'_{min} \).

Let \( G_u \) and \( G_v \) be subgraphs of partition \( \pi_k \) and \( \pi'_k \) respectively. Consider the protocol subgraph \( PG(G_u \cup H_u, G_v \cup H'_v) \). Any error conditions in the reachability graph of \( PG(G_u \cup H_u, G_v \cup H'_v) \) can be classified as either ‘definite’ or ‘potential’ protocol errors. Definite errors are errors not only for the protocol subgraph \( PG(G_u \cup H_u, G_v \cup H'_v) \), but will eventually become errors for the entire protocol graph. Potential errors are errors that may vanish as a result of forming larger partitions. This is due to the fact that \( \pi_k \) and/or \( \pi'_k \) is too small and as a result we have cross interactions between subgraphs. Let

\[ p = \text{any node of subgraph } G_u \]
\[ q = \text{any node of subgraph } G_v \]
\[ G_s = \text{immediate successor subgraph to } G_u \]
\[ G'_s = \text{immediate successor subgraph to } G'_v \]
\[ h_s = \text{header node of subgraph } G_s \]
\[ h'_s = \text{header node of subgraph } G'_s \]

Table 2 makes the distinction between ‘definite’ and ‘potential’ protocol errors.

Consider state deadlock of the form \( \langle (p, q), (\lambda, \lambda) \rangle \). We classify this as a definite error because this state deadlock remains even after we form the union of subgraphs \( G_u \) and \( G'_v \) with their successor subgraphs. However, we consider \( \langle (h_s, q), (\lambda, \lambda) \rangle \) as a potential error because this state deadlock may vanish when we form a larger parti-
Table 2
Definite and Potential Protocol Errors.

<table>
<thead>
<tr>
<th>Type</th>
<th>Definite Protocol Error</th>
<th>Potential Protocol Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>State deadlocks</td>
<td>(p,q), (λ,λ)</td>
<td>(P,q), (λ,λ)</td>
</tr>
<tr>
<td>Unspecified reception</td>
<td>(p,q), (X or λ or p)</td>
<td>(p,q), (X or λ or p)</td>
</tr>
<tr>
<td>Nonexecutable transitions</td>
<td>Not classified</td>
<td>Not classified</td>
</tr>
</tbody>
</table>

Fig. 10: The Session Protocol of S.62.

The initial state for both the Called and Calling Terminal is $\epsilon$. Let $G_1$ and $G_2$ be the state transition graphs for the Called and Calling Terminal, respectively. The minimal partitions $\pi_{\text{min}}$ and $\pi'_{\text{min}}$ are

protocol is specified in terms of state transition graphs shown in Figure 10 (from Figure 1/S.62) [14]. We do not consider the document transfer part of the procedure. The state transition diagrams cover two modes of operations, namely, One Way Communication (OWC) and Two Way Alternate (TWA). The abbreviations represent the following messages:

CSS = Command Session Start
CSE = Command Session End
CSCC = Command Session Change Control
RSSN = Response Session Start Negative
RSSP = Response Session Start Positive
RSEP = Response Session End Positive
RSCCP = Response Session Change Control Positive

Example 5.3

The S.62 session protocol forms part of the control procedures for Teletex service. Before a document can be exchanged between two Teletex terminals, a session must first be established between the two terminals. The protocol entities involved in the session protocol are the Calling Terminal and the Called Terminal. The session
The protocol subgraphs $PG(G_1)$ partitioned according to $Tr(0)$ and $Tr'(0)$ are shown in Figure 11. The validation of these protocol subgraphs can be done by sight.

Example 5.4

The X.25 Call Set Up protocol is used for setting up virtual circuits in a packet switched network. The protocol entities involved are the Data Terminal Equipment (DTE) and Data Circuit Terminating Equipment (DCE). The protocol is specified in terms of a combined state diagram Figure B-2/X.25 in reference [15], which can be translated into separate state transition graphs for the DTE and DCE. See Figure 12. The abbreviations represent the following messages:

- **CA** = Call Accepted
- **CC** = Call Connected
- **CR** = Call Request
- **IC** = Incoming Call
- **CA** = Call Accepted
- **CC** = Call Connected
- **CR** = Call Request
- **IC** = Incoming Call

![Diagram](image1)

![Diagram](image2)

![Diagram](image3)
The POSET of structured partitions $SP_1$ and $SP_2$ for the DTE and DCE are shown in Figures 13 and 14 respectively. The initial state for both the DTE and DCE is $s_1$. Let $G_1$ and $G_2$ be the state transition graphs for the DTE and DCE respectively. We begin our analysis with the smallest partition $\pi(0)$ and $\pi'(0)$ for $G_1$ and $G_2$ where

\[ \pi(0) = \{\bar{s}_1, \bar{s}_2, \bar{s}_3, \bar{s}_4, \bar{s}_5\} \]

and

\[ \pi'(0) = \{\bar{s}_1, \bar{s}_2, \bar{s}_3, \bar{s}_4, \bar{s}_5\}. \]

The initial protocol subgraph $PG(G_1 \cup H_1, G'_1 \cup H'_1)$ for the partition pair $\pi(0), \pi'(0)$ is shown in Figure 15a. It contains a potential error which is an unspecified reception at the global state $(s_2, s_1), (CR, IC))$. To eliminate this potential error, we construct larger partitions $\pi_1$ and $\pi'_1$ for the DTE and DCE respectively where

\[ \pi_1 = \{\bar{s}_1, \bar{s}_2, \bar{s}_3, \bar{s}_4, \bar{s}_5\} \]

and

\[ \pi'_1 = \{\bar{s}_1, \bar{s}_2, \bar{s}_3, \bar{s}_4, \bar{s}_5\}. \]

The initial protocol subgraph $PG(G_1 \cup H_1, G'_1 \cup H'_1)$ for the partition pair $\pi_1, \pi'_1$ is shown in Figure 15b. It contains two potential errors which are state deadlocks $((s_2, s_2), (A, A))$ and $((s_3, s_1), (A, A))$. To eliminate these potential errors, we construct large partitions $\pi_{\text{min}}$ and $\pi'_{\text{min}}$ for the DTE and DCE respectively where

\[ \pi_{\text{min}} = \{\bar{s}_1, \bar{s}_2, \bar{s}_3, \bar{s}_4, \bar{s}_5\} \]

and

\[ \pi'_{\text{min}} = \{\bar{s}_1, \bar{s}_2, \bar{s}_3, \bar{s}_4, \bar{s}_5\}. \]

The protocol subgraphs $PG(G_1 \cup H_1, G'_1 \cup H'_1)$ of $PG(G_1, G_2)$ partitioned according to $\pi_{\text{min}}$ and $\pi'_{\text{min}}$ are shown in Fig. 15c. The search for the minimal partitions terminates at this point since all the protocol subgraphs shown in Figure 15c have the required progress properties.

Instead of partitioning a protocol graph into subgraphs for analysis, we can synthesize a complex protocol graph by connecting together simpler protocol subgraphs. If the individual protocol subgraphs have the required progress properties and the connections between subgraphs satisfy the structured partition requirements, then the resultant protocol graph will also have the required progress properties. The technique does not specify how individual subgraphs can be constructed. One can use, for example, the production rules of Zafiropulo, et. al. for this purpose [9].

The synthesis technique for $PG(G_1, G_2)$ can be described as follows:

1. Construct the initial protocol subgraph $PG(G_1 \cup H_1, G'_1 \cup H'_1)$. Let subgraphs $G_1$ and $G'_1$ label the root nodes of trees $T_1$ and $T'_1$ to be constructed.
2. Determine which nodes of subgraph $G_1 \cup H_1$ and $G'_1 \cup H'_1$ can act as header nodes for successor subgraphs to $G_1$ and $G'_1$ respectively. This can be done by examining the reachability graph of $PG(G_1 \cup H_1, G'_1 \cup H'_1)$. If in the reachability graph, there exist terminal nodes of the form $(p, q), (q, q)$, then $p$ and $q$ can be designated as header nodes $h_x$ and $h'_x$, respectively.

3. For every pair of header nodes $(h_x, h'_x)$ chosen in step 2, construct a protocol subgraph $PG(G_x \cup H_x, G'_x \cup H'_x)$, using $h_x$ and $h'_x$ as the header nodes for $G_x \cup H_x$ and $G'_x \cup H'_x$, respectively. Let the subgraphs $G_x$ and $G'_x$ be the immediate descendants to $G_1$ and $G'_1$ in trees $T_1$ and $T_2$ respectively.

4. Determine which nodes of subgraphs $G_x \cup H_x$ and $G'_x \cup H'_x$ can act as header nodes for successor subgraphs to $G_x$ and $G'_x$ respectively, using the method described in step 2. For each header node pair $(h_x, h'_x)$, construct a protocol subgraph $PG(G_x \cup H_x, G'_x \cup H'_x)$. Let subgraphs $G_p$ and $G'_q$ be immediate descendants to $G_x$ and $G'_x$ respectively in trees $T_1$ and $T_2$. If the protocol subgraph $PG(G_p \cup H_p, G'_q \cup H'_q)$ has been constructed before, then $G_p$ and $G'_q$ can be regarded as duplicate nodes of trees $T_1$ and $T_2$ respectively, if their subsequent behavior in both cases are identical. If the protocol subgraph $PG(G_p \cup H_p, G'_q \cup H'_q)$ is a final subgraph, then $G_p$ and $G'_q$ are terminal nodes of trees $T_1$ and $T_2$ respectively. Repeat step 4 going down the trees until every leaf node of trees $T_1$ and $T_2$ is either a duplicate node or a terminal node.

5. Finally, convert the protocol tree $PT(T_1, T_2)$ into the protocol graph $PG(G_1, G_2)$ by merging identical nodes in trees $T_1$ and $T_2$. As examples, the S.62 session protocol and the X.25 call set up protocol can be synthesized by constructing the protocol subgraphs shown in Figures 11 and 15c respectively.

6. Conclusions

Communicating finite state machines (CFSMs) is a well established model for protocols. Recent research has focused on methods for simplifying the analysis of CSFMs [16,17] as well as constructing CFSMs so that specific progress properties are preserved [18,19]. In this paper, we approach CFSMs from the standpoint of decomposition. We showed how CFSMs can be decomposed using structured partitions. In the process, we developed an analysis and synthesis method for CFSMs based on protocol subgraphs.

The result obtained for analysis is similar to [16] in the following respect. The transition region discussed in this paper represents the redundancy in the reachability graph discussed in [16]. In addition, we pointed out that there exists protocols that are non-decomposable. For protocols that are decomposable, the reduction in the number of global states that need to be examined for the required progress properties comes at the expense of additional computations required for finding structured partitions.

It has come to our attention that the Multi-phase Construction technique developed independently by Chow et. al. is similar to our synthesis method [20]. A phase discussed in [20] corresponds to a protocol subgraph discussed in this paper. In reference [20], the reader will find a similar method applied to the synthesis of the token ring and IBM BSC protocols.

The partitioning technique discussed in this paper applies to two process protocols. The extension to protocols involving $n$ processes can be carried out in a straightforward manner. A more interesting problem is how to extend the partitioning technique to protocols modeled by Extended Finite State Machines (EFSMs). This is an area for further research.

Acknowledgement

The authors wish to thank M. Gouda and other reviewers for their many helpful comments regarding this paper.

References


A CONSTRUCTION TECHNIQUE FOR SELF SYNCHRONIZING FINITE STATE PROTOCOLS

Raymond E. Miller*

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

ABSTRACT

A technique is presented for constructing a finite state protocol from an originally given finite state specification of one process. We present a construction, showing that it provides send-receive symmetric solutions which are "self-synchronizing". Two lemmas are given that provide insight into the types of interactions that arise in these types of finite state protocols. In essence we show that interactions occur between the processes only through "isomorphic transitions" and that during any interaction between the processes at most one of the two FIFO queues of messages is nonempty.

INTRODUCTION

Considerable work has been done on developing methods for analyzing and synthesizing communication protocols (refs. 1-4, 7). Although a number of different formalisms have been used one of the simplest models is the finite state model (refs. 2, 4). Here, a pair of communicating processes is modeled by a finite state system for each process, and a pair of FIFO queues to hold messages being sent in each direction. In each machine the transitions between states represent the sending and receiving of messages. Each queue represents a one-directional, unbounded FIFO channel to hold the messages sent by one process, but not yet received by the other process. The finite state model clearly has limitations. Extensions to the finite state model exist, for example see ref. 9 in this proceedings. More "powerful" models such as Petri nets and formal programming languages have also been used in some works. Nevertheless, the finite state model has proved to be quite effective in treating many problems in practical communication protocols. Because of this, as well as the relation of this paper to the previous work of refs. 3, 5 we use this finite state model here.

The construction of finite state protocols that are free from error was first studied by Zafiropoula et al. in ref. 3. Here an interactive stepwise process was developed in which the designer interacted with the process to check for various error conditions during construction. In ref. 4 the design process consisted of constructing the protocol out of smaller protocol subgraphs, each of which satisfied certain properties, and were interconnected in prescribed ways. In ref. 5 a construction process was developed that started with an initial machine describing one process, constructing the other machine for the other process, and then modifying both machines so that they satisfied some guaranteed progress properties.

The approach proposed here is closely related to, and was stimulated by, that given in ref. 5. It differs, however, in several important ways. Our construction yields a send-receive symmetric pair of processes, whereas the procedure in ref. 5 starts out by constructing send-receive symmetric processes, but then modifies them to obtain the guaranteed progress properties. Also, the construction in ref. 5 to guarantee progress, results in some messages being disregarded. Our approach creates protocols where no messages need to be disregarded. To do this, we need to add some "dummy messages" in the protocols. We feel this alternative of adding dummy messages may, at times, be advantageous over ignoring messages, and that the maintenance of send-receive symmetric processes has distinct advantages. Finally, our construction differs from that in ref. 5 in the sense that rather than constructing the pair of machines and modifying both, we first modify the original machine by adding dummy messages and states, and then construct the symmetric second machine. The process is very simple and straightforward, and progress properties are insured. In essence, each "mixed state" of the machine is replaced by two states, neither of which is mixed, and then it is proved that the send-receive symmetric machines without mixed states are "self-synchronizing".

In Section I we define the finite state protocol, and various properties, more precisely. In Section II we give two lemmas that provide important characterizations of the types of interactions that can occur between send-receive symmetric finite state protocols. Section III presents the construction and proves the self-synchronizing property we desire. Section IV provides some concluding remarks.

I. FINITE STATE PROTOCOLS

In this section we repeat some of the definitions of finite state protocols and correctness properties that we will use (refs. 4, 5, 6). Also, we introduce several new definitions convenient for this paper.

Definition: A Finite State Protocol FSP = (S, M, Σ, Q, τ) consists of:

(i) S = {S1, S2}, two disjoint finite sets representing the states of processes P1 and P2, respectively.

(ii) M = {M12, M21}, two finite sets, where M12 represents the messages that can be sent from pro-

---

* This work was partially supported through a contract with GTE Laboratories.
For processes which contain final states, we define a final state.

Definition: A final state is a state from which no transitions are possible.

It should be noted that r is a function and thus each process has a unique behavior.

We enforce a first-in first-out discipline on the sending and receiving of messages. Thus, the channels are modeled as FIFO queues.

We define as FIFO queues connecting process P_i to process P_j.

Definition: A channel C_{ij} is a FIFO queue connecting process P_i to process P_j. The contents of C_{ij} is labeled c_{ij}, it is a string of symbols from M_{ij} and represents the FIFO queue of messages being sent from process P_i to process P_j. Each entry in c_{ij} is an element of M_{ij}.

In this paper, we treat the channels as ideal. Nonideal channels can be modeled by the introduction of channel events. These channel events can be used to model the loss, duplication or resequencing of messages in the channel.

Definition: A global state is a pair \(<S, C>\) where S = \(\{s_1, s_2\}\) with s_1 \in S_1 and s_2 \in S_2 representing the current states of processes P_1 and P_2, respectively and C = \(\{c_{12}, c_{21}\}\) representing the current contents of channels C_{12} and C_{21}, respectively.

For processes which contain final states, we define a final global state as follows:

Definition: A final global state \(<S_f, C_f>\) is given by \(<\{s_f, s'_f\}, \{A, A\}>\) where s_f and s'_f denote the final states of processes P_1 and P_2, respectively and A represents the empty channel.

Definition: A stable state pair \(\{s, s'\}\) is a pair derived from a global state \(<S, C>\) where S = \{s, s'\} and C = \(\{c_{12}, c_{21}\}\) with both c_{12} and c_{21} empty.

A yields relation \(\rightarrow\) on global states is defined as follows:

Definition: A global state \(<S, C>\) \(\rightarrow\) \(<S', C'>\) if and only if there exist i, j and c_{ij} satisfying one of the following execution rules:

(i) \(s_i' = r_i(s_i, -x)\) and \(c_{ij}' = c_{ij} x\) (See 1.)

(ii) \(s_j' = r_j(s_j, +x)\) and \(c_{ij} = c_{ij} x\)

(iii) \(s_i' = r_i(s_i, \lambda)\) and \(c_{ij}' = c_{ij}\)

Except for the one execution rule applied, all other elements of \(<S', C'>\) are equal to the elements of \(<S, C>\). If \(<S, C>\) \(\rightarrow\) \(<S', C'>\) we say that \(<S, C>\) yields \(<S', C'>\).

Definition: A global state \(<S, C>\) is said to be reachable from the initial global state \(<S_0, C_0>\) if and only if \(<S, C>\) \(\rightarrow\) \(<S', C'>\) where \(<S_0, C_0>\) = \(<\{s_0, c_0\}, \{A, A\}>\) and \(\rightarrow\) denotes the reflexive, transitive closure of \(\rightarrow\).

A transition at state s for event \(\sigma\) is specified if and only if \(r(s, \sigma)\) is defined.

Definition: A transition at state s for event \(\sigma\) is executable if and only if there exists a reachable global state \(<s_i, s_j>, \{c_{ij}, c_{ji}\}\) with s = s_i or s = s_j, and if \(\sigma\) is a receive transition \(+x\), then for s = s_i, c_{ij} = x Y and for s = s_j, c_{ji} = x Z with Y and Z being arbitrary.

Definition: A state s is called a sending (or receiving) state iff all of the transitions leaving s are sending (or receiving, respectively) transitions. Otherwise s is called a mixed state.

Definition: We say that the sequence of global states \(<S_0, C_0>, <S_1, C_1>, <S_2, C_2>, \ldots\> is a sequence following \(<S, C>\) iff \(<S, C>\) \(\rightarrow\) \(<S_1, C_1>\) and for each \(i > 1, <S_{i-1}, C_{i-1}>\) \(\rightarrow\) \(<S_i, C_i>\).

If such a sequence of global states starts with \(<S_0, C_0>\) then each of these global states is reachable, and the sequence is caused by the prescribed sequence of events T from \(\Sigma\) that results from the sequence of execution rules applied in the sequence of yield relations.

Since the messages sent by P_i in T to P_j, i ≠ j and i, j = 1, 2, pass through channel C_{ij}, the k^{th} message sent by P_i must be the k^{th} message received by P_j (if indeed P_j receives this k^{th} message). Assume that k^{th} message is x and is both sent and received. The sending by P_i creates a state transition in P_i and we designate this as p = \(r_i(s, -x)\). Similarly we designate the transition of receiving x by P_j by t = \(r_j(s, +x)\). With this terminology we call the pair:

\[\{(p = r_i(s, -x), t = r_j(s, +x))\}\]

1. We use \(c_{ij} x\) to represent the concatenation of message sequence c_{ij} with x.
Let the finite state machines for processes $P_i$ and $P_j$ be represented by finite state graphs $G_1$ and $G_2$, respectively. Then the protocol graph $PG(G_1, G_2)$ denotes the protocol between processes $P_i$ and $P_j$.

**Definition:** A protocol is structurally balanced (SB) if there is a one-to-one correspondence between the subgraphs of $G_i$ representing process $P_i$ and the subgraphs of $G_j$ representing process $P_j$.

**Definition:** A protocol is send-receive (S-R) symmetric if the finite state graphs $G_1$ and $G_2$ representing processes $P_i$ and $P_j$, respectively are isomorphic and for every send event associated with an edge in one graph, there is a corresponding receive event associated with the corresponding edge in the other graph.

The isomorphism described here consists of an isomorphism between nodes of $G_1$ and $G_2$ as well as between labels of messages on edges where the nodes of $G_1$ and $G_2$ represent the same message. This isomorphism is unique.

### II. TWO LEMMAS

As we have mentioned earlier, the construction technique we describe replaces mixed states in the original machine with pairs of states such that the modified machine has no mixed states. Also, the construction yields an S-R symmetric pair of machines. Thus, the following lemma, which we call the interaction lemma, provides a useful basic property for such finite state protocols.

**Lemma 1:** (Interaction Lemma)

Let $P = (P_1, P_2)$ be a finite state protocol which is S-R symmetric and has no mixed states. Then, for $i, j = 1, 2$, and any positive integer $k$, the $k^{th}$ interaction from $P_i$ to $P_j$ occurs through isomorphic transitions between $P_i$ and $P_j$.

A simple inductive proof on $k$ proves this lemma ref. 8.

We should note, of course, that this lemma only states how the $k^{th}$ interaction will proceed if it actually occurs, but it does not guarantee that the $k^{th}$ interaction will actually occur.

**Lemma 2:** Let $P = (P_1, P_2)$ be a finite state protocol which is S-R symmetric and has no mixed states. Any reachable global state $<S, C>$ of $P$, where $C = \{c_{12}, c_{21}\}$, has either $c_{12}$ or $c_{21}$ empty.

The proof of this lemma can also be found in ref. 8.

Lemmas 1 and 2 provide considerable insight into the behavior of the S-R symmetric protocols with no mixed states. Essentially, the behavior is such that the two machines are forced into keeping in step with each other. If one machine races ahead with send messages, then the other must receive these messages prior to sending any messages. It is possible, however, that one or both of the machines contain cycles that consist totally of send messages. If such a cycle is entered, the machine may go around this cycle indefinitely and the other machine may fall behind in receiving these messages causing an unbounded queue of messages to build up in the channel. Clearly, if neither machine has such a “send cycle” then the channel queues must be bounded, and the bound for $c_{ij}$ is the longest consecutive sequence of send signals that exist in any path in $P_i$. A formal proof of this result is not included here, but is quite obvious. Also, this means that an algorithm to determine the upper bound on the length of the queue necessary for $C_i$ depends only upon the send signal sequences in paths of $P_i$.

### III. THE CONSTRUCTION TECHNIQUE

With the lemmas of the previous section we are now ready to describe our construction technique. We assume that the original protocol specification is given in terms of a finite-state machine description of one of the processes, say $P_1$, and that we wish to construct the companion process $P_2$ such that the resulting finite state protocol has guaranteed progress properties. In this sense our objective is identical to that of ref. 5, but our construction technique, and thereby the resulting finite state protocol, provide a different result. We capture the notion of guaranteed progress in a property we call “self-synchronizing”.

**Definition:** Consider an S-R symmetric finite state protocol $P$. $P$ is said to be self-synchronizing if and only if for each reachable global state $<S, C>$ = $(s_1, s_2, c_{12}, c_{21})$ the following condition holds:

For every sequence $<S, C, <S_1, C_1>, \ldots$, following $<S, C>$ there exists an $n$ such that $<S_n, C_n> = (s_1^n, s_2^n, c_{12}, c_{21})$ is stable and $s_1^n$ and $s_2^n$ are isomorphic states.

We now describe the construction technique.

### THE CONSTRUCTION

Assume process $P_1$ is given as a finite state system

$$P_1 = (S_1, \Sigma_1, o_1, r_1)$$

where: $S_1$ is a finite set of states

$\Sigma_1$ is the set of send and receive signals

where $-x$ represents sending of message $x$ and $+x$ represents receiving of message $x$

$o_1$ is the initial state

$r_1$ is the transition function

We also assume that each cycle of $P_1$ contains at least one transition with a send message and one transition with a receive message. That is, $P_1$ contains no “send-cycles” or no “receive-cycles”.

**Step 1:** For each mixed state $s$ of $P_1$, replace $s$ with a pair of states $s'$ and $s''$ such that all incoming transitions to $s$ become incoming transitions to $s'$, all outgoing send transitions of $s$ are outgoing send transitions from $s'$ to the same states as for $s$, all outgoing receive transitions of $s$ are outgoing receive transitions from $s'$ to the same
states as for $s_1$ and one send transition with a signal $-D$ (where $-D$ or $+D$ do not appear in $\Sigma_1$) goes from $s'$ to $s''$. See Figure 1 for a pictorial description of this transformation.

![Figure 1: The Transformation](image)

It should be noted that we are being somewhat informal in our description for Step 1. The terms "incoming transitions" and "outgoing transitions" have not been formally defined, but are intuitively obvious. Also, when we say that transitions go to "the same states as for $s_1$" this clearly creates problems if some of these states are mixed states and have been previously split into two states. Yet, it should be clear that they go to the state which has the outgoing $-D$ transition.

Observations about Step 1:

1. This creates a system $P_1'$ which has two states for each mixed state of $P_1$ plus the same number of send and receive states as $P_1$.
2. $P_1'$ has no mixed states.
3. $P_1'$ has no send-cycles or receive-cycles.
4. $P_1'$ is a deterministic finite state system.
5. The same $-D$ can be used for each added transition without confusion since the state-signal-pair determines the next state.

Step 2: Construct a finite state system $P_2'$ from $P_1'$ by creating an isomorphic graph to $P_1'$ where each send (receive) edge of $P_1'$ is replaced with a receive (send) edge of $P_2'$ with the same message as $P_1'$.

Observations about Step 2:

1. $P_1'$ and $P_2'$ are S-R symmetric.
2. $P_1'$, $P_2'$, along with channels $C_{12}$ and $C_{21}$, form an S-R symmetric finite state protocol.
3. Since $P_1'$ has no send-cycles or receive-cycles, neither does $P_2'$.

Theorem: The finite-state protocol $P = (P_1', P_2)$ is send-receive symmetric and self-synchronizing.

Proof: That $P$ is S-R symmetric is immediate from the construction. We need to prove the self-synchronizing property. The interaction lemma shows that each interaction of $P_1'$ and $P_2'$ occurs via isomorphic transitions. Now, since there are no send or receive cycles, one machine (say $P_1'$) can send a sequence of messages to the other machine ($P_2'$) through a sequence of send transitions, but this must eventually reach a receive state $s_a$. During this process $P_2'$ can only wait or respond with the isomorphic receive transitions. Each transition of $P_2'$ will be from one receive state to another receive state until the message is received from $P_1'$ that caused the transition to the receive state. Since $P_2'$ is moving from receive state to receive state it cannot send any messages until it reaches the state isomorphic to $s_a$ of $P_1'$ (call this $s_a'$). Now $\{s_a, s_a', c_{12}, c_{21}\}$ is a state satisfying the self-synchronizing definition; $s_a$ and $s_a'$ are isomorphic states. Since $P_1'$ was sending messages $c_{21}$ was empty (by Lemma 2) during the previous steps while $P_2'$ was receiving messages, and $P_2'$ has not since sent any message. Now when $P_2'$ reaches $s_a'$ it has received all messages from $P_1'$ from the last sending sequence thus $c_{12}$ is also empty. (That $c_{12}$ is empty could be proved more formally by an inductive proof to show that there were no other outstanding messages in $c_{12}$).

Example 1: To illustrate the construction consider the system $P_1$ shown in Figure 2(a). This is an example taken from ref. 5. In this example state 3 is the only mixed state. Thus Step 1 replaces state 3 with the pair of states $3'$ and $3''$ giving the machine $P_1'$ of Figure 2(b). Step 2 of the construction yields process $P_2'$ shown in Figure 2(c). Clearly $P_1'$ and $P_2'$ are send-receive symmetric. Stable global states occur for $P_1$, $P_2$ states $(1,1)$, $(2,2)$, $(3',3')$ and $(3'',3'')$ and it is easy to see that any sequence goes through one of these stable states every few steps, so the protocol is self-synchronizing.
With this example we see that when machine $P_1$ reaches state 3 it could either send message 3, returning to state 2 (i.e., stay in the +2, -3 cycle), or wait to receive message 4 to exit from the cycle. Our construction adds a "dummy" message $D$ which machine $P_1'$ sends to $P_2'$ as an indication of exiting the (+2, -3) cycle. In this sense machine $P_1'$ has total control over when the protocol machines are to get out of the cycle. Other constructions in ref. 8 show variations on which machine exercises control.

Comparing with the method of Gouda and Yu (ref. 5), their procedure creates a machine $P_2$ directly from $P_1$ which is isomorphic to $P_1$. Then, however, the two machines can get out of synchronization because of the mixed states. Modifications to $P_1$ and $P_2$ are then made to cause resynchronization, but this involves ignoring some of the transmitted messages that occur as the machines get out of synchronization. In contrast, our approach needs to add the "dummy" message, but this insures that the machines remain synchronized and that no messages need be ignored.

IV. CONCLUSIONS

We have shown a construction technique for finite state protocols which starts with a specification in terms of a finite state description for one of the processes and ends with a finite state protocol which is S-R symmetric and self-synchronizing. Guaranteed progress is insured through this self-synchronizing property, where the construction also insures that no messages transmitted by one process get ignored by the other process. We do not get unspecified receptions, nonexecutable transitions, or deadlocks in the protocol unless there are inherent problems with the original specification of $P_1$. In ref. 8 these error conditions are defined; also, two additional constructions are shown that have the same properties as given here and are closely related to this construction. They provide for alternatives to which of the two processes is "controlling" parts of the message interchange.

ACKNOWLEDGEMENTS

The initial concepts for this construction arose in discussions with two Ph.D. students, Robert Hernandez and Gilbert Lundy, in a graduate study group, and I acknowledge their contributions. The formulation of the properties, such as self-synchronization, the lemmas, and the proof that the constructions actually have the desired properties, are the result of subsequent independent work.

REFERENCES

AN APPROACH TO MODELING COMMUNICATION PROTOCOLS USING
FINITE STATE MACHINES AND SHARED VARIABLES

Raymond E. Miller
and
G. M. Lundy

School of Information and Computer Science
Georgia Institute of Technology
Atlanta, Georgia 30332

ABSTRACT

A method for the specification of communication protocols is discussed which models both processes and channels as extended finite state machines, and in which the processes communicate through shared variables. Each process also has its own local variables, and state transitions are associated with enabling predicates and actions. An analysis using system state exploration is discussed, which helps limit the state explosion that occurs when the standard reachability analysis using global states is done. An example protocol specification is given, and some other possible methods of analysis are discussed.

1. Introduction

In the past several years, a considerable amount of research has been carried out developing formal models and verification techniques for communication protocols. This emphasizes both the practical importance and the complexity of these to computer network designers. The methods which have been used can be grouped into two broad classes: methods using finite state machines, and methods using programming language models. The pure finite state machine model has the advantage that correctness properties such as freedom from deadlocks are analyzable by reachability analysis. It is also simple and easy to understand. Its primary disadvantage is that, with no memory (other than use of states) complex protocols with sequences (e.g., HDLC) cannot be modeled and analyzed without a "state explosion". Even with automated reachability analysis, the number of global states in a nontrivial protocol can be so large that even a computer analysis is impractical. Program language models have the power that any protocol can be modeled, but many properties become undecidable. Protocols expressed in programming languages may be close to the implementation, but are also often difficult to understand (as is the case with any complex program).

These facts, and particularly some experience in modeling of HDLC protocols using the pure finite state model led the authors to search for another way, by starting with the finite state model, but extending it slightly to remedy these defects, while retaining ease of analysis advantages. There have been several papers using extended or unified models. Our model is similar to that of Bochmann and Gecsei ref. 1, but is different in several ways. We model each channel explicitly using a finite state machine (with variables). Each entity or machine has its own local variables as in ref. 1, but there are also shared variables. It is through shared variables that processes communicate. Thus, we have neither queues to represent channels, as in most finite state models, nor the "distantly initiated actions" of (ref. 1). Because channels are machines, we are able to more explicitly model errors as well as timing considerations.

In the next section we define the model. First we define a "system of communicating machines," and then define a "protocol system" by restrictions on the model. In sections 3 and 4, a simple example protocol and its verification are given to illustrate the model. Further examples can be found in (ref. 9).

2. The Model

First we define a general model which we call a "system of communicating machines". The general model might be used to model several different types of systems in addition to protocols, such as parallel programs or operating systems. Then we place some restrictions on the "system" to get another slightly less general model, which we refer to as a "protocol system".

2.1. Systems of Communicating Machines

A system of communicating machines is an ordered pair $O = (M, V)$, where

$M = \{ m_1, m_2, \ldots, m_n \}$

is a finite set of machines, and

$V = \{ v_1, v_2, \ldots, v_k \}$

is a finite set of shared variables, with two designated subsets $R$ and $W$ specified for each machine $m_i$. The subset $R$ of $V$ is called the set of read access variables for machine $m_i$, and $W$ the set of set of write access variables for $m_i$.

Each machine $m_i \in M$ is defined by a tuple $(S_i, s_i, L_i, N_i, \tau_i)$ where

1. $S_i$ is a finite set of states;
2. $s_i \in S_i$ is a designated state called the initial state of $m_i$;
3. $L_i$ is a finite set of local variables;
4. $N_i$ is a finite set of names, each of which is associated with a unique pair $(p, a)$ where "p" is a predicate on the variables of $L_i \cup R_i$, and "a" is an action on the variables of $L_i \cup W_i \cup R_i$.

Specifically, an action is a mapping $a : L_i \times R_i \rightarrow L_i \times W_i$.

3.8.1.
from the local variables and read access variables to the local variables and write access variables.

5. \( r_i : S, X \rightarrow S \) is a transition function, which is a partial function from the states and names of \( m_i \) to the states of \( m_i \).

Machines model the entities, which in a protocol system are processes and channels. In a parallel programming system the machines may each model separate programs, which run in parallel. The shared variables are the means of communication between the entities.

Intuitively, \( R_i \) is the subset of \( V \) to which \( m_i \) has read access, and \( W_i \) is the subset to which \( m_i \) has write access. A machine will be allowed to make a transition from one state to another when the predicate associated with the name for the transition is true. Upon taking the transition the action associated with the transition name also occurs. The action changes local and shared variables in the means of communication between the entities.

The set \( L_i \) of local variables specifies a name and a range for each. The range must either be finite (discrete values) or countably infinite. Each variable may be assigned an initial value.

A system state is a tuple of all machine states. That is, let \((M, V)\) be a system of \( n \) communicating machines, and let \( s_i, \quad 1 \leq i \leq n \) denote the state of machine \( m_i \). Then the \( n \)-tuple \((s_1, s_2, \ldots, s_n)\) is the system state of \((M, V)\). The initial system state is the system state such that every machine is in its initial state.

A global state may be defined as a triple \((\bar{S}, \bar{V}, \bar{L})\) where:

1. \( \bar{S} \) is a system state;
2. \( \bar{V} \) is the tuple of values currently assigned to the variables of \( V \);
3. \( \bar{L} \) is the tuple of values currently assigned to the variables of \( L_i \), the local variables of \( m_i \), for all \( i, 1 \leq i \leq n \).

That is, the global state of a system consists of the system state, plus the values of all variables, both local and shared. The initial global state is that global state such that every machine is in its initial state, and all variables (both local and shared) for which initial values are specified have the specified value, and all other variable values are left undefined.

We say that a system state \( \bar{S} = (s_1, \ldots, s_n) \) corresponds to the global state \((\bar{S}', \bar{V}', \bar{L}')\) iff \( \bar{S} = \bar{S}' \), that is, all the machines are in the same state.

Let \( r (s_1, n) = s_2 \) be a transition which is defined on a machine \( m_i \) of a system of communicating machines \((M, V)\). Transition \( r \) is said to be enabled if the enabling predicate \( p \) (associated with name \( n \)) is true. Transition \( r \) may be executed whenever \( m_i \) is in state \( s_1 \) and the enabling predicate \( p \) is enabled. The execution of \( r \) is an atomic action, in which both the state change and the action \( "a" \) (associated with \( n \)) occur simultaneously.

We emphasize that the execution of a transition is an atomic action; that is, it is indivisible, and it is assumed to occur instantaneously. We also point out that transitions in different machines can occur simultaneously; if transition \( r \) is enabled in machine \( m_i \), and \( r \) in machine \( m_j \), \( i \neq j \), then either \( r \) may occur first, \( r_n \) may occur first, or both may occur simultaneously. This holds for any number of transitions, up to the number of machines in the system. We will see an example of simultaneous transitions later in the paper.

2.2. Protocol Systems

A protocol system is a system of communicating machines, with the following restrictions:

1. the set \( M \) of machines is partitioned into two subsets \((P, C)\). Hereafter we will use an ordered triple \((P, C, V)\) when referring to a protocol system.

Intuitively, the set \( P \) is the set of processes, while the set \( C \) is a set of channels connecting the processes.

2. the set \( V \) of shared variables is restricted to one process per variable. That is, only one member of the set \( P \) may have access (either read or write) to a variable.

This corresponds to the intuitive notion that a computer can share variables with a channel but not with another computer across the network.

3. each channel machine \( c \in C \) is restricted to the sharing of variables with at most 2 processes and no other channels.

That is, a channel can be thought of as a connection between two machines.

2.3. Observations

Note that if the values of all variables are restricted to some finite range, then the model can be reduced to a simple finite state machine. Otherwise, an infinite number of "global states" are possible. However even if the number of global states is infinite, the number of "system states" is finite, because of the finiteness of each machine. In restricted cases this may allow us to do a reachability analysis on the system states. Even in the case that the values of all variables are of a finite range, the number of global states in the equivalent finite state system may be so large as to be unmanageable. We believe this model can be used to reduce these analysis difficulties.

3. An Example Protocol Specification

In this section we use the model to specify a simple alternating bit protocol. The protocol is basically the same as in Bochmann and Gecsei (ref. 1). The main purpose of this section is to illustrate the protocol model.

Fig. 1 depicts the state diagrams and the shared and local variables. There are four machines which correspond to the sender, receiver, and the two 1-way channels connecting them. The channel "chanSR" is the channel from sender to receiver, and "chanRS" from receiver to sender.

The variables shown in all upper case are shared by two machines (a process and a channel), and variables local to one machine are in lower case, or mixed upper and lower case.

In this protocol, both entities sharing the variables have both read and write access to the shared variable, but we shall see that the enabling predicates prohibit two
machines having write access to a shared variable at the same time. In fact, one would generally expect this to be the case in a correct protocol, because otherwise one machine could overwrite a variable written by the other before it is used.

The initial state of each machine is marked by a token "●". The possible values which the variables may have are enumerated, and an underscore indicates the initial value. Note that not all variables are initialized. The letter "E" is used to indicate an empty value. We interpret the undefined value as a unique, meaningless value, whereas "E" or empty has a specific meaning.

The enabling predicates and actions which go with each transition are shown in Table 1. The action "new(data)" which corresponds to the sender's "new" transition signifies that the "user" has given a new data block to the Sender to be transmitted. Similarly, the action "use(data)" which corresponds to the Receiver transition "use," signifies that the Receiver has passed a data block to the receiving user.

A normal sequence of communication is as follows:

1. the sending user places a data block X to be transmitted into the variable "Sdata" of Sender;
2. Sender executes transition "new";
3. Sender places the data block X into DATA-S, executing the transition "D";
4. ChanSR transmits X across to Receiver by copying X into DATA-R from DATA-S (the "XmitD" transition);
5. Receiver copies X into Rdata, and passes the data block X on to the receiving user, through the +D, D ≠∅, and use transitions;
6. Receiver then acknowledges the data block through ChanRS, by executing the "-A" transition, placing the acknowledgement into ACK-R;
7. ChanRS then passes the acknowledgement from Receiver back to Sender by executing "XmitA", which copies the ACK-R into ACK-S.

 Normally the protocols will be specified as above, without specifically listing each set and function as in the definition; but all of these should be obvious from the diagrams.

4. Verification

In this section we verify some properties of the model by a reachability analysis on the system states. This type of analysis can be used to show that the protocol has certain properties, one of the most important being freedom from deadlocks.

We say that a system is deadlocked if every machine m in the system is in a state X, such that no transition out of X is enabled, and the state is not a final system state. (A system state is final if every machine is in a final state).

This definition covers deadlocks as usually defined in finite state models, as well as some unspecified receptions. See ref. 9 for an explanation and comparison with unspecified receptions.

The reachability analysis is done in much the same way as for communicating finite state machines. A system state in our example is a 4-tuple <w,x,y,z> where w, x, y, and z represent the states of sender, chanSR, chanRS and receiver, respectively. The graph is started from the initial state <1,1,1,3> with all variables initialized.

The complete system state reachability graph is shown in ref. 9; there are 34 system states.
### Table 1: Action Tables for Protocol Transitions

<table>
<thead>
<tr>
<th>transition</th>
<th>enabling predicate</th>
<th>action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sender</td>
<td>new</td>
<td>$Sdata := \text{new}(data); \ seq := \ seq \oplus 1$</td>
</tr>
<tr>
<td></td>
<td>$-D$</td>
<td>DATA-$S := Sdata; SEQ-$S := \ seq;$</td>
</tr>
<tr>
<td></td>
<td>$+A$</td>
<td>$ack := ACK-S; ACK-S := E;$</td>
</tr>
<tr>
<td></td>
<td>$A=$</td>
<td>-</td>
</tr>
<tr>
<td>Receiver</td>
<td>use</td>
<td>$use(data) := Rdata; \ exp := \ exp \oplus 1$</td>
</tr>
<tr>
<td></td>
<td>$-A$</td>
<td>ACK-$R := \ exp; \ seqnb := 'none';$</td>
</tr>
<tr>
<td></td>
<td>$+D$</td>
<td>$\ exp := \ expel 1;$</td>
</tr>
<tr>
<td></td>
<td>$D#$</td>
<td>$\ exp := \ exp \oplus 1;$</td>
</tr>
<tr>
<td>ChanSR</td>
<td>XmitD</td>
<td>DATA-$S := Rdata; DATA-$R := E;\ exp := \ exp \oplus 1;$</td>
</tr>
<tr>
<td>ChanRS</td>
<td>XmitA</td>
<td>ACK-$R := E; ACK-$S := E;\ exp := \ exp \oplus 1;$</td>
</tr>
<tr>
<td></td>
<td>RetS</td>
<td>$DATA-S := Sdata; SEQ-$S := \ seq;$</td>
</tr>
<tr>
<td></td>
<td>RetR</td>
<td>$ack := ACK-S; ACK-S := E;$</td>
</tr>
<tr>
<td></td>
<td>ACK-$R := \ exp;$</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DATA-$R := E;$</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>DATA-$S := E;$</td>
<td>-</td>
</tr>
</tbody>
</table>

Several points are worth noting. First, this is a reachability graph of system states, not global states. If we attempted analysis of the global states, each node would contain far more information. Also there would be many more global states than system states, making such an analysis much more lengthy. Second, in carrying out the analysis one must insure that each system state is actually reachable. This means that there is at least 1 global state corresponding to each system state. In practice this means that the values of variables are considered, and only transitions which are enabled for some valid set of variables are taken.

Now we can also show the effect of allowing simultaneous transitions. The number of system states is not changed, but there are a number of additional system state transitions. An example is the system state $<3,2,1,4>$. Two transitions are enabled in this state, and either may go first or both simultaneously, so there are three transitions out of this state. (See fig. 2).

![Simultaneous transitions](image)

Analysis shows that there are no deadlock states; every state can transition to some other state, and the initial system state is reached again after a finite number of steps.

This analysis has been done with the assumption that such an analysis on system states is valid, and that different global states which "correspond" to the same system states are in some sense "equivalent". We believe that in this example it is fairly clear that such an assumption is valid; but we have not proved this, and are well aware of some other example specifications in which this is not the case. Further research is contemplated to determine necessary and sufficient conditions, or at least sufficient conditions, for such a system state analysis to be valid.

### 5. Approaches to Analysis

Having specified the protocol, the next step is to analyse it in some way in order to prove it has certain desired correctness properties, such as freedom from deadlocks, from unspecified receptions, or from other errors. Also, it is desirable to show that it is in some sense functionally correct, that is, meets its intended purpose. Rather than giving an analysis in this paper, we discuss various ways such an analysis might be done.

The method of global reachability analysis is theoretically possible, but as mentioned earlier, is very inefficient. Even for such a limited protocol as this, a state explosion occurs. For example, the protocol in ref. 2 was very similar to this, and an automated global reachability analysis generated nearly 70,000 global states.

A reachability analysis of system states would be more reasonable, and is a possibility; but even the number of system states in this protocol is large enough to make such an analysis by hand difficult. A program to generate system states might be helpful here. While there will undoubtedly be a much smaller number of system states than global states, whether that number is sufficiently small for a simple analysis is not known. An upper bound on the number of system states is the product of the number of states in all the machines. Many of these are undoubtedly not reachable. It would be desirable to derive conditions under which such a system state analysis is feasible.

Another possibility, is to examine the pieces of the system, insure that they each work individually, and then show that they are combined in such a way that the correctness properties shown in each piece are preserved.
This is, after all, the way the protocol is specified. This approach would be much like the decomposition in Choi and Miller (refs. 7-8).

Another way of exploring the pieces individually could be the method of projections, as discussed in Lam and Shankar (ref. 4). An "image protocol" of the piece could be defined, and certain safety and liveness properties proven in the smaller image protocol also hold in the original protocol, so long as the image protocol is constructed properly and meets certain conditions. While our protocol is not specified with the same model as in ref. 4, we believe that the projection method might be modified to apply to our model.

In programming language descriptions of protocols, correctness properties are often verified by making and proving an assertion. We believe that making an assertion about a state of a process, and proving it, is another possible approach to verifying some properties, particularly functional correctness.

We hope to investigate how these various analysis approaches could be used within our model.

6. Conclusions

A model for specifying communication protocols was presented, which is called a protocol system. Another more general model, called system of communicating machines was also defined. These models include local variables in each machine, and shared variables for communication. Communication channels are specified using machines which share variables with the processes they connect. A simple example specification was given, and reachability analysis by state was discussed.

The model presented here is more powerful than a pure finite state model, and the state reachability analysis shows promise toward reducing the complexity of protocol analysis.

Our use of machines to represent channels is more direct than the FIFO queue approach used in most finite state models. We also believe that for some applications it is also preferable to the "distantly initiated actions" of (ref. 1). Finally, we believe that the "protocol system" model captures the intuitive notion of a network, with the finite window size, and the passing of data and acknowledgments explicitly represented in the model.

Several areas for further work are open, some of which we have already mentioned. Can the model be used to specify more complex protocols? One such protocol is the complete HDLC. Another use might be to model broadcast networks. Another question concerns formal properties. What correctness properties, if any, can be shown to hold in the protocol system model? Since this is an extension of the finite state modeling approach it is hoped that some of the techniques developed there could be carried over to this model.

References


