A Layered Communication System Generator

Yung-Chen Hung and Gen-Huey Chen
Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, Republic of China

Although many approaches have been proposed for modeling and automatically validating communication protocol entities, no approaches have been proposed for automatically implementing various layered communication systems. In the article, we propose a layered communication system generator that can automatically implement various layered communication systems. The layered communication system generator is established on the basis of a generalized communication system framework. In addition, a layered communicating finite state machine is introduced to specify the layered communication systems, and a matrix approach is proposed for protocol validation. We have implemented the proposed layered communication system generator in DOS and UNIX environments, in which several real-world protocols from Department of Defense and IEEE 802 standards organizations have been successfully developed.

1. INTRODUCTION

Layered communication system generation is a process of developing new layered communication systems. The objective is to provide a systematic way of developing layered communication systems so that their correctness can be ensured.

To facilitate smooth communication among different information processing systems in a heterogeneous environment, we need a universal framework of computer networking architectures. The well-known seven-layer open system interconnection reference model, which was proposed by the International Standards Organization, is a worldwide standard for the creation of communication systems. On this basis, the layered communication system generator can be defined as follows: Given a communication specification under a layered architecture, the layered communication system generator constructs the corresponding executable communication system so that the interactions within the communication system are complete, live, and deadlock free. Completeness ensures that all possible inputs are handled and all possible outputs are received. Liveness ensures that there are no nonexecutable transitions. The absence of deadlocks guarantees that no two communication systems wait for each other forever.

In this article, we first propose a generalized communication system framework suitable for a variety of target machines. We use a layered communicating finite state machine (L_CFSM) to specify a layered communication system. In an L_CFSM, each protocol entity and interlayer synchronization are specified by a modified finite state machine (MFSM) and a pair of FIFO queues, respectively. The MFSM is represented by an event matrix and an action matrix. Based on these, we can automatically implement a given L_CFSM into a communication system while systematically validating the logical correctness of the given L_CFSM at the same time.

There has been some progress in creating an integrated set of tools for automatic protocol design. These tools are useful in the specification, validation, and implementation of a protocol. However, they are dedicated only for implementing a protocol entity [1-8]. The objective of this article is to show how to automatically implement a layered communication system from an L_CFSM, not only a protocol entity.

The procedure of developing a layered communication system consists of five steps and is outlined as follows (Figure 1):

1. Install the generalized communication system framework in the given target machine.
2. Input the specification of the layered communication system.
3. Transform protocol entities into event and action matrices.
4. Validate protocol entities to ensure they are logical error free.
5. Download the event and action matrices to the target machine.

Address correspondence to Professor Gen-Huey Chen, Dept. of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, Republic of China

© Elsevier Science Publishing Co., Inc.
655 Avenue of the Americas, New York, NY 10010

0164-1212/92/$5.00

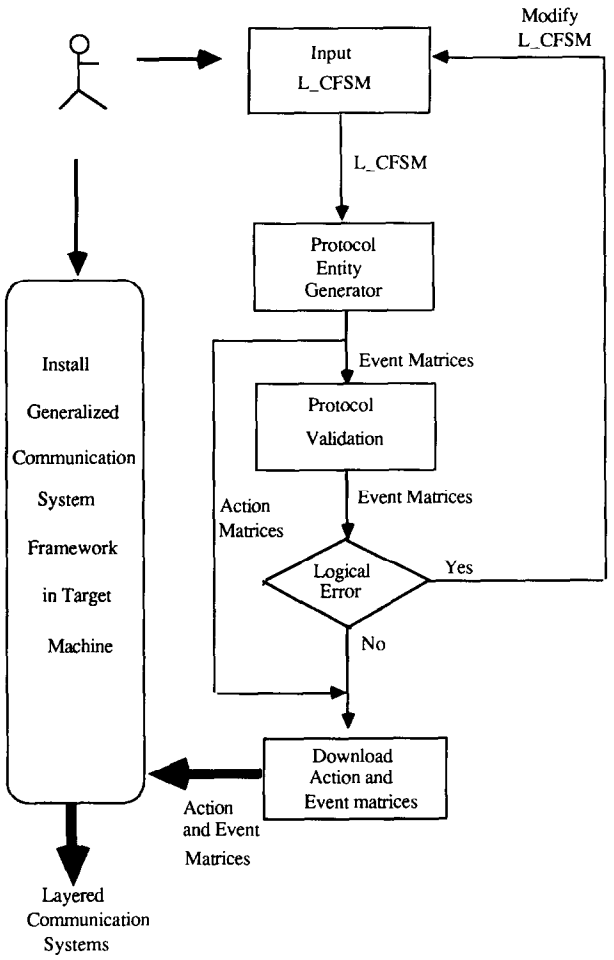

Figure 1. Overview of the layered communication system generator.

Although we have suggested the installation of the generalized communication system framework to be the first step of the procedure, it is not necessary for the installation to precede the input of the L_CFSM. The installation must be complete only before downloading the action and event matrices to the target machine.

Also, note that in Figure 1, the action matrices need not be processed through protocol validation and logical error checks, since they bear no relation to protocol validation. In fact, whenever logical errors are detected in the event matrices, the L_CFSM needs to be modified, which may cause changes in the action matrices.

The approaches to protocol validation are mainly classified into two categories: perturbation analysis [5, 8] and the relational algebraic approach [9]. The perturbation analysis cannot systematically express the validation procedure. On the other hand, the relational algebraic approach is based on relational operations (e.g., projection, selection, replacement, join, θ-join, etc.) and its realization relies on a powerful relational data base system (e.g., INGRES). We propose a new approach in which protocol validation is performed systematically by matrix operations.

The rest of this article is organized as follows. A generalized communication system framework is presented in section 2. The L_CFSM is introduced in section 3. In section 4, we show a protocol entity generator, which transforms each protocol entity into an event matrix and an action matrix. In section 5, we propose a matrix approach to protocol validation. The implementation experience of the layered communication systems is discussed in section 6. Finally, in section 7, we summarize the features of the proposed layered communication system generator and suggest some future research directions. Appendices list the event and action matrices of TCP in our implementation.

2. A GENERALIZED COMMUNICATION SYSTEM EFRAMEWORK

In this section, we present a generalized communication system framework (see Figure 2) suitable for a multivendor network system. This framework contains a set of modules for interlayer synchronization, interlayer flow control, memory management, and time management. We assume that each protocol entity can multiplex more than one protocol entity in its upper layer. For simplicity, we show only one protocol entity in each layer in Figure 2. The framework consists mainly of unidirectional FIFO queues, a switcher, a memory manager, a time manager, layered protocol entities, and a protocol entity generator. This environment is suitable for various layered communication systems.

Queue Structure

A pair of unidirectional FIFO queues are used for intermodule, interlayer, and intercomputer synchronization. They are classified into three categories.

1. External queues, which are necessary in communicating finite state machines [10-12], are used for intercomputer synchronization.
2. Internal queues are used for interlayer synchronization (see Figure 3).
3. Service queues are used for synchronization between protocol entities and the time manager (Figure 3).

Each protocol entity can send (receive) packets into (from) a subset of queues called its outgoing (incoming) queues. The reasons why we adopt the queue structure in the framework are as follows.

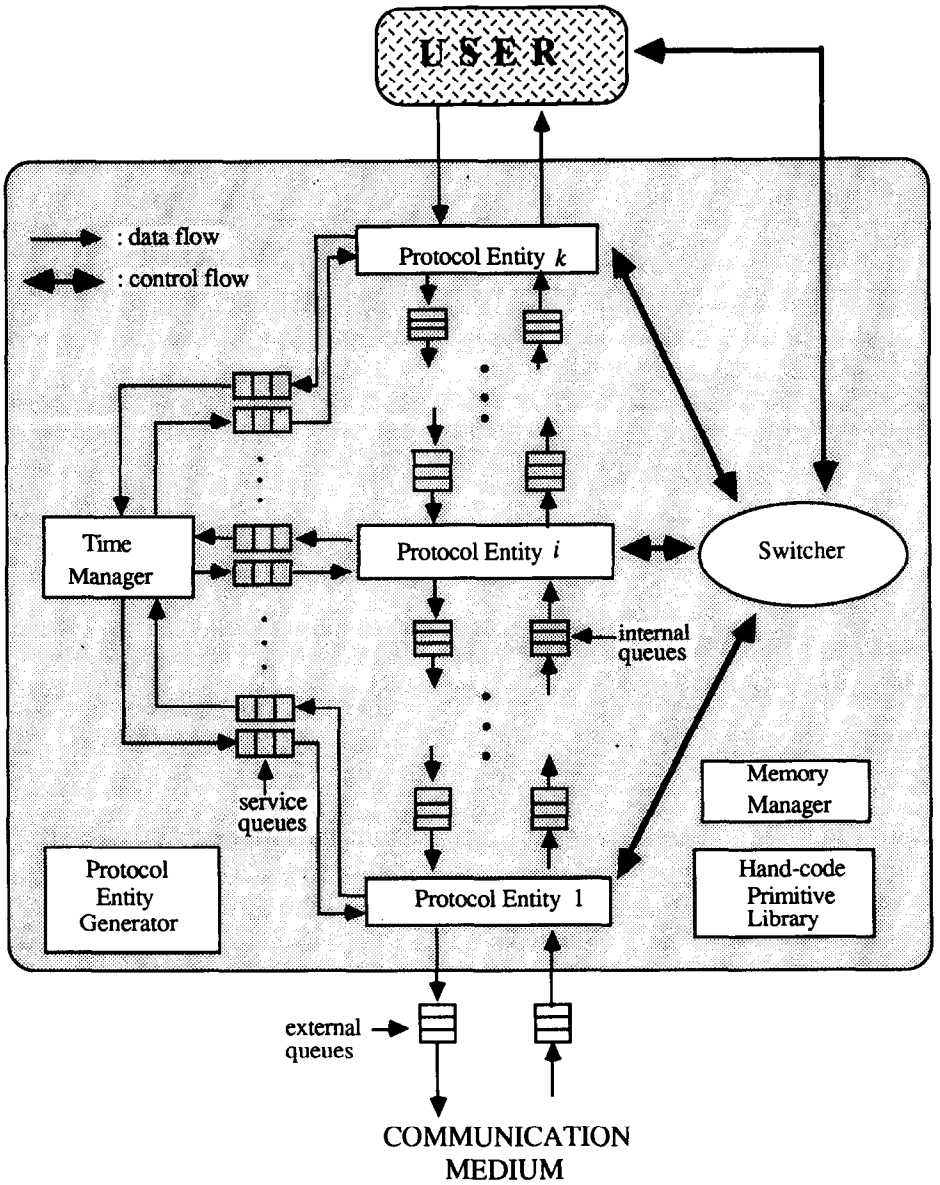

Figure 2. A generalized communication system framework.

Interlayer synchronization. The queue structure is the most commonly used technique for interprocess synchronization in operating systems. It is also suitable for interlayer synchronization. In a layered communication system, each protocol entity can multiplex more than one protocol entity in its upper layer, which is called upward multiplexing. In upward multiplexing, the communication between the multiplexed protocol entities is called local delivery. For example, in Figure 3, protocol entity C multiplexes protocol entities A and B, and the local delivery between protocol entities A and B must go through protocol entity C. Generally, a layered communication system can be implemented by either treating each protocol entity as a separate process or taking the whole communication system as a process. The latter approach often implements each protocol entity as a function or a subroutine, and so the interlayer synchronization can be implemented by function or subroutine calls. However, there are two weaknesses in such an implementation. One is that a long path of function or subroutine calls due to consecutive local deliveries will make the whole computer system fail. The other is that a race condition will occur between changing protocol entity states and returning from function calls when local deliveries proceed. Therefore, we adopt the former implementation, in which each protocol entity is represented by a separate process. In such an implementation, interprocess syn-

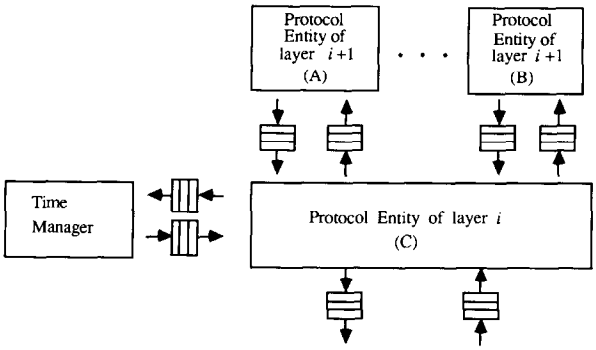

Figure 3. Queues for synchronization between adjacent layers and between the time manager and protocol entities.

chronization is realized through the use of the queue structure.

Interlayer flow control. Queues are a good checkpoint to measure the workload of the protocol entities and to detect bottlenecks in a layered communication system.

Communication system migration. In a layered communication system, queues are a good interface between adjacent layers, With the use of queues, layers may be arbitrarily grouped into three parts: the upper part stays in the host computer, the middle part is downloaded to a front-end processor, and the lower part is realized by a special hardware chip (adaptor).

Switcher

As mentioned above, each protocol entity is represented by a separate process. Therefore, how CPU switches between these processes is important. If the layered communication system runs in a single-user environment (e.g., DOS), a switcher is needed to allocate CPU among these processes. On the other hand, if the layered communication system runs in a multiuser environment (e.g., UNIX), a new switcher is still needed because the original OS scheduling criteria cannot fit the layered communication system. The switcher is responsible for allocating CPU to the process with the highest priority. The priorities of processes are usually determined by the number and urgency of packets in the incoming queues.

The switcher itself is a main process in a layered communication system. When it gets CPU to run, it switches CPU to the highest priority protocol entity, which then removes packets from incoming queues (dequeue), processes them, inserts some packets into outgoing queues (enqueue), and finally returns CPU to the switcher. By repeatedly switching CPU among the protocol entities, the switcher can make the communication system more productive and the interlayer flow smoother.

Memory Manager

For convenient use of the layered communication system, we provide a uniform logical view of memory utilization. The memory manager defines a fixed-size (16, 256, or 512 bytes) block of memory as a logical memory unit. The memory manager also provides a set of routines including allocation, deallocation, enqueue, and dequeue to realize the necessary memory operations. The allocation (deallocation) routine is invoked to allocate (free) a logical memory unit. The enqueue and dequeue routines are used as basic queue operations. Because the protocol entities share a common memory pool, pointers, instead of data packets, are physically transmitted between the protocol entities.

Time Manager

In a layered communication system, there are some time-related mechanisms such as retransmission, timeout, and piggyback. The framework provides a time manager for the following purposes: start a timer, stop a timer, keep track of timers, and send an alarm signal to a protocol entity as timeout. The interface between the time manager and the protocol entities consists of a pair of queues, through which start and stop events can be transmitted from protocol entities to the time manager and timeout events can be transmitted in reverse direction.

Protocol Entity

A protocol entity consists of two parts, nonlogical and logical [1].

Nonlogical part. Some tasks in a protocol entity, such as formatting, encoder, decoder, error-checking, syntax-checking, fragmentation, reassembly, event interpreting, and synchronizing, can be considered nonlogical. Their implementations may require special considerations concerning the target machine. The nonlogical part is implemented by hand. In our implementation, a primitive library is provided to support nonlogical primitives and an action matrix is used to drive these primitives. The action matrix can be auto-

J. SYSTEMS SOFTWARE
1992; 18:157-170

matically generated from an MFSM by the protocol entity generator, which we describe in section 4.

Logical part. The remaining part of a protocol entity, excluding the nonlogical part, is considered logical. It is the most important part of a protocol entity and in our implementation it is represented by an event matrix. The event matrix can be automatically generated from an MFSM by the protocol entity generator.

The framework we have proposed here uses the queue structure to realize interlayer communication. Before a protocol entity sends a packet, which is the unit of exchanges between the protocol entities, it executes the primitives to encapsulate appropriate addressing, control information, and data into the packet. The packet is then sent to an upper, lower, or peer protocol entity, and the protocol entity that receives the packet will determine which event is to occur (by interpreting the control information), the next state, and the primitives to be executed (by considering the current state, the event to occur, the event matrix, and the action matrix). In this way, interlayer control can be realized through the use of the queue structure and primitives.

From the above discussion we know that the protocol entities are event driven. In fact, a protocol entity is an alternative representation of an MFSM and its next state is determined by three factors: current state, event, and event matrix (Figure 4). It also determines which primitives are to be executed during its execution by three factors: current state, event, and action matrix.

3. LAYERED COMMUNICATING FINITE STATE MACHINES

The layered communication system we consider in this section consists of two host computers and two bounded unidirectional external queues. Each host computer has a k-layer communication architecture and $(k-1)\cdot 2$ internal queues. For example, Figure 5 shows the abstract model of a three-layer communication system.

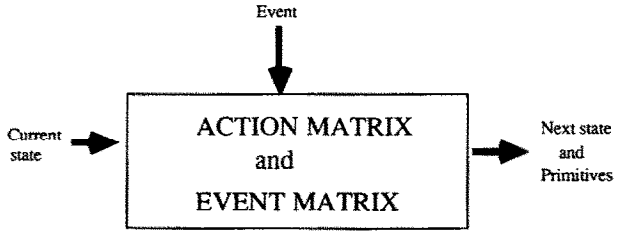

Figure 4. Execution of a protocol entity.

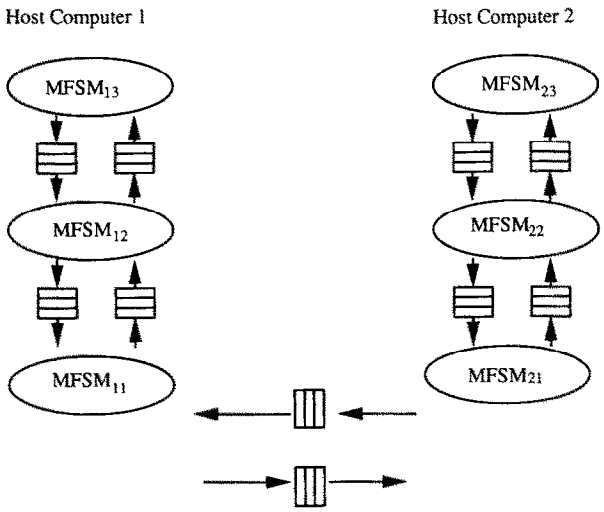

Figure 5. Abstract model of a three-layer communication system.

Note that the timer mechanism is optional since it is not necessarily needed in each protocol entity. Thus, for simplicity, the time manager is not shown in the abstract model.

Each protocol entity in the layered communication system is represented by an MFSM. Like a finite state machine, an MFSM can be expressed as a directed graph whose nodes represent the states of the protocol entity and arcs represent the transitions between the states. The state transition is triggered by an event occurrence and is possibly accompanied by some primitives.

A layered communication system, which consists of a set of rules that govern the exchange of packets between computers, can be represented by an L_CFSM as defined below.

Definition 1. An L_CFSM is a 10-tuple

$$\begin{aligned}&(\langle S_{ij}\rangle_{i=1\text{ or }2,\ j=1,\ldots,k},\\ &\quad \langle A_{ij}\rangle_{i=1\text{ or }2,\ j=1,\ldots,k},\\ &\quad \langle O_{ij}\rangle_{i=1\text{ or }2,\ j=1,\ldots,k},\\ &\quad \langle M_{ij}\rangle_{i,\ j=1\text{ or }2,\ i\neq j},\\ &\quad \langle M'_{ip}\rangle_{i=1\text{ or }2,\ p=1,\ldots,k-1},\\ &\quad \langle M''_{ip}\rangle_{i=1\text{ or }2,\ p=1,\ldots,k-1},\\ &\quad \langle C_{ij}\rangle_{i,\ j=1\text{ or }2,\ i\neq j},\\ &\quad \langle C'_{ip}\rangle_{i=1\text{ or }2,\ p=1,\ldots,k-1},\\ &\quad \langle C''_{ip}\rangle_{i=1\text{ or }2,\ p=1,\ldots,k-1},\ succ),\end{aligned}$$

where

k is the number of layers,

S_{ij} is the state of the protocol entity (denoted by P_{ij}) of the jth layer in computer i,

A_{ij} is the set of primitives that are executed by the protocol entity P_{ij},

O_{ij} is the initial state of the protocol entity P_{ij},

M_{ij} is the set of packets that can be sent from computer i to computer j through an external queue (denoted by Q_{ij}),

M'_{ip} is the set of packets that can be sent downward from the $(p+1)$th layer to the pth layer in the computer i through an internal queue (denoted by Q'_{ip}),

M''_{ip} is the set of packets that can be sent upward from the pth layer to the $(p+1)$th layer in the computer i through an internal queue (denoted by Q''_{ip}),

C_{ij} is the capacity of the external queue Q_{ij},

C'_{ip} is the capacity of the internal queue Q'_{ip},

C''_{ip} is the capacity of the internal queue Q''_{ip},

succ is a partial function from $S_{ij} \times E_{ij}$ to $S_{ij} \times 2^{A_{ij}}$, where E_{ij} is the set of events that can occur to protocol entity P_{ij}.

Note that the packets that appear in different queues are different. That is, the sets M_{ij}, M'_{ip} and M''_{ip} are disjoint.

For example, let us consider Figure 6, where

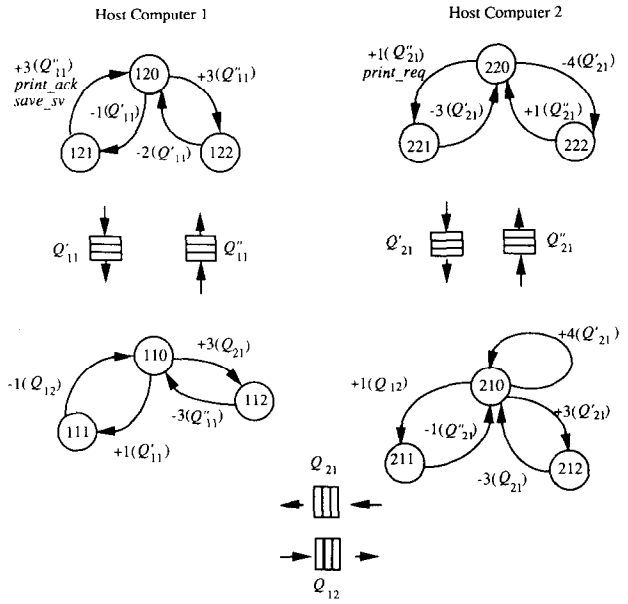

Figure 6. Example of a two-layer communication system.

$k = 2$

$S_{11} = \{110, 111, 112\}$, $S_{12} = \{120, 121, 122\}$

$S_{21} = \{210, 211, 212\}$, $S_{22} = \{220, 221, 222\}$

$A_{12} = \{print\_ack, save\_sv\}$, $A_{11} = A_{21} = \{\ \}$,

$A_{22} = \{print\_req\}$

$M'_{11} = \{1(Q'_{11}), 2(Q'_{11})\}$, $M''_{11} = \{3(Q''_{11})\}$,

$M_{12} = \{1(Q_{12})\}$, $M_{21} = \{3(Q_{21})\}$

$M'_{21} = \{3(Q'_{21}), 4(Q'_{21})\}$, $M''_{21} = \{1(Q''_{21})\}$,

$E_{11} = \{+1(Q'_{11}), +3(Q_{21}), -1(Q_{12}), -3(Q''_{11})\}$

$E_{12} = \{+3(Q''_{11}), -1(Q'_{11}), -2(Q'_{11})\}$

$E_{21} = \{+1(Q_{12}), +3(Q'_{21}), +4(Q'_{21}),$
$-1(Q''_{21}), -3(Q_{21})\}$

$E_{22} = \{+1(Q''_{21}), -3(Q'_{21}), -4(Q'_{21})\}$,

$O_{11} = 110$, $O_{12} = 120$, $O_{21} = 210$, and $O_{22} = 220$.

The function *succ* is defined for 17 argument pairs that are represented by the 17 arcs in Figure 6. Three examples of defined *succ* values are

$succ(121, +3(Q''_{11})) = (120, \{print\_ack, save\_sv\})$

$succ(220, +1(Q''_{21})) = (221, \{print\_req\})$

$succ(120, -1(Q'_{11})) = (121, \{\ \})$.

An example of an undefined *succ* value is $succ(110, -4(Q_{12}))$. In Figure 6, the plus and minus signs indicate receiving and sending events, respectively. For example, $+3(Q''_{11})$ represents an event that a packet $3(Q''_{11})$ received from the queue Q''_{11} and $-4(Q'_{21})$ represents an event that a packet $4(Q'_{21})$ sent to the queue Q'_{21}.

The states of the protocol entities and the contents of the queues will change during the execution of the layered communication system. We define a global state of the layered communication system as

$$(s_{11}, \ldots, s_{1k}, s_{21}, \ldots, s_{2k}, D_{12}, D_{21}, D'_{11}, \ldots, D'_{1(k-1)},$$
$$D''_{11}, \ldots, D''_{1(k-1)}, D'_{21}, \ldots, D'_{2(k-1)},$$
$$D''_{21}, \ldots, D''_{2(k-1)}),$$

where s_{ij} is the current state of the protocol entity P_{ij}, and D_{ij}, D'_{ij}, and D''_{ij} are the sequences of packets in Q_{ij}, Q'_{ij}, and Q''_{ij}, respectively.

Initially, each protocol entity P_{ij} is in state O_{ij} and all the queues are empty.

4. PROTOCOL ENTITY GENERATOR

As mentioned in the previous section, a protocol entity is modeled as an MFSM. The behavior of an MFSM is again represented by a directed graph, which can be transformed into two matrices, an event matrix and an action matrix. The protocol entity generator

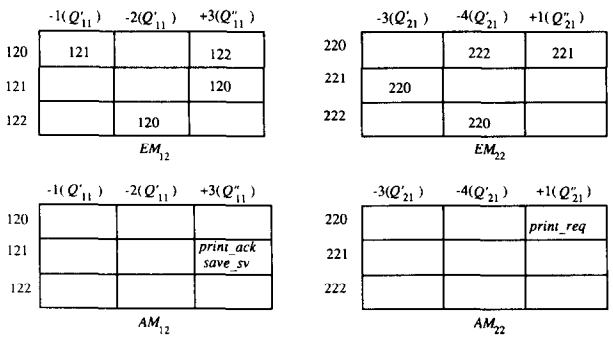

Figure 7. Examples of event and action matrices.

is responsible for transforming MFSMs into the event and action matrices. We denote the event matrix and the action matrix of protocol entity P_{ij} by EM_{ij} and AM_{ij}, respectively. For example, EM_{12}, AM_{12}, EM_{22} and AM_{22}, with respect to the example in Figure 6, are shown in Figure 7. Each row and column of EM_{ij} and AM_{ij} represent a state and an event, respectively. The entries of EM_{ij} and AM_{ij} are determined as follows: If $succ(s_1, x) = (s_2, W)$, then $EM_{ij}(s_1, x) = s_2$ and $AM_{ij}(s_1, x) = W$; otherwise $EM_{ij}(s_1, x)$ and $AM_{ij}(s_1, x)$ are null, where $s_1, s_2 \in S_{ij}$, $x \in E_{ij}$ and $W \in 2^{A_{ij}}$. The meaning of $EM_{ij}(s_1, x) = s_2$ is that protocol entity P_{ij} in state s_1 enters the states s_2 due to event x. Similarly, $AM_{ij}(s_1, x) = W$ represents that protocol entity P_{ij} in state s_1 executes the subset W of primitives due to event x.

After all MFSMs are transformed into the event and action matrices, protocol validation can be performed on these matrices instead of on protocol entities themselves. If the protocol entities are logical error free, then these matrices are downloaded to the generalized communication system framework, which has been installed in the target machine. After that, the layered communication system is well established. If not, the specification of the layered communication system should be modified.

In the next section, a matrix approach is proposed for protocol validation.

5. PROTOCOL VALIDATION USING A MATRIX APPROACH

The function of protocol validation is to detect potential logical errors in an L_CFSM. There are three types of logical errors: deadlock unspecified reception, and nonexecutable transition. A deadlock is a global state in which only receiving events are defined, but all the queues are empty. Thus, if the layered communication system gets into a deadlock state, it will stay in that state indefinitely. An unspecified reception is a global state in which only receiving events are defined, not all the queues are empty, and the first packet in each nonempty queue is not specified in the receiving events; in other words, no receiving events may occur in an unspecified reception state since the first packet in each nonempty queue is not the desired one. A nonexecutable transition is a transition (i.e., an event occurrence) in an MFSM that never occurs during the execution and exists because of some inconsistencies or syntatic errors in the design of the MFSM.

In this section, the protocol validation is realized by performing reachability analysis using a matrix approach. The reachability analysis is a procedure to generate all reachable global states of the layered communication system, starting from the initial global state. All executable transitions can be found during the reachability analysis. Thus, if logical errors exist, they will be found after the reachability analysis.

Before presenting the matrix approach, let us consider a simplified example shown in Figure 8. Figure 8*a* shows a one-layer communication system. The global state is of the form (a, b, X, Y), where a and b are the states of protocol entities A and B, respectively, and X and Y are the sequences of packets in queues Q_{AB} and Q_{BA}, respectively. In Figure 8, for simplicity, the queue symbol (Q_{AB} or Q_{BA}) is omitted for each event since no ambiguity occurs. Also, primitives are not shown since they bear no relation to protocol validation.

The matrix approach to the reachability analysis is divided into three stages as described below.

Stage 1: Construct the event matrices. As described in section 4, the event matrices can be automatically constructed from the L_CFSM. For example, the event matrices that are constructed for the protocol entities of Figure 8*a* are shown in Figure 8*b*.

Stage 2: Construct the patterns. For each protocol entity P_{ij}, we use an $f \times 1$ matrix $PA_{ij,S}$ to indicate which events may happen during global state S, where f is the number of events in E_{ij} (Figure 8*c*). Such a matrix is called a pattern. Each row of $PA_{ij,S}$ represents an event, and the event sequence is the same as the event sequence of EM_{ij}. We denote the events by $x_{ij1}, x_{ij2}, \ldots, x_{ijf}$ in sequence. The contents of each pattern $PA_{ij,S}$ tell the status of the incoming queues (whether they are empty) and the outgoing queues (whether they are full) of P_{ij}. Depending on the entry of $PA_{ij,S}$, the corresponding event is enabled or disabled. For a receiving event X_{ijw}, $PA_{ij,S}(x_{ijw}) = 1$ if the incoming queue associated with it is not empty, and 0 otherwise. For a sending event

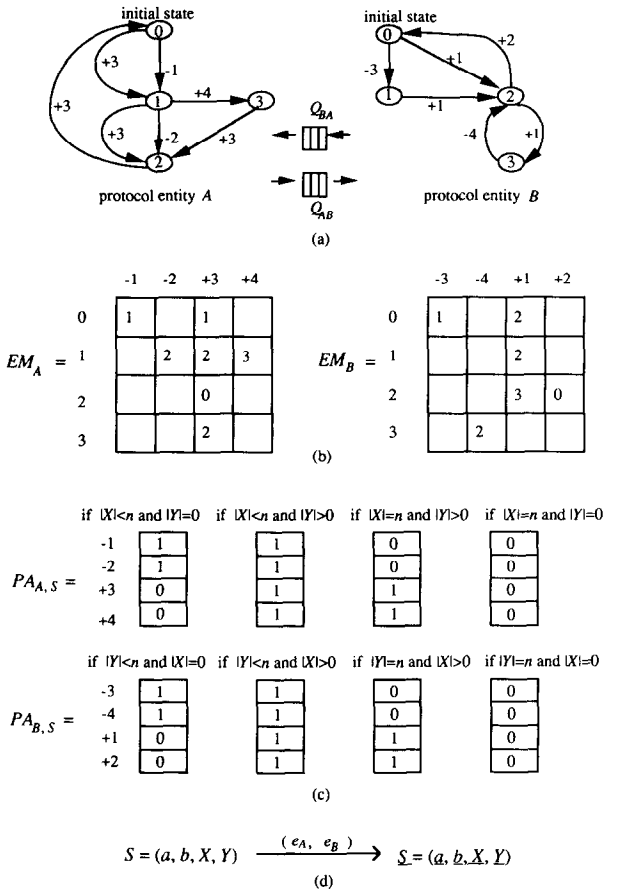

Figure 8. Reachability analysis using a matrix approach. *a*, An example of a one-layer communication system. *b*, Event matrices of the protocol entities. *c*, Patterns to induce next global states. *d*, State transition.

x_{ijw}, $PA_{ij,S}(x_{ijw}) = 1$ if the outgoing queue associated with it is not full, and 0 otherwise.

Let y_{ijw} denote the associated packet of the event x_{ijw}; that is, y_{ijw} is being processed when the event X_{ijw} occurs. The entries of $PA_{ij,S}$ can be determined as follows.

Case 1. x_{ijw} is a receiving event and $x_{ijw} \in E_{ij}$.

If ($y_{ijw} \in M_{pi}$ and $|D_{pi}| > 0$, where $p = 1$ or 2 and $p \neq i$) or ($y_{ijw} \in M'_{ij}$ and $|D'_{ij}| > 0$) or ($y_{ijw} \in M''_{i(j-1)}$ and $|D''_{i(j-1)}| > 0$),
then $PA_{ij,S}(x_{ijw}) = 1$;
else $PA_{ij,S}(x_{ijw}) = 0$.

Case 2. x_{ijw} is a sending event and $x_{ijw} \in E_{ij}$.

If ($y_{ijw} \in M_{ip}$ and $C_{ip} > |D_{ip}|$, where $p = 1$ or 2 and $p \neq i$) or ($y_{ijw} \in M'_{i(j-1)}$ and $C'_{i(j-1)} > |D'_{i(j-1)}|$) or ($y_{ijw} \in M''_{ij}$ and $C''_{ij} > |D''_{ij}|$),
then $PA_{ij,S}(x_{ijw}) = 1$;
else $PA_{ij,S}(x_{ijw}) = 0$.

Note that $|D_{ij}|$, $|D'_{ij}|$, and $|D''_{ij}|$ denote the number of packets in queues Q_{ij}, Q'_{ij}, and Q''_{ij}, respectively.

For example, Figure 8*c* shows the matrices (patterns) $PA_{A,S}$ and $PA_{B,S}$ for protocol entities A and B in Figure 8*a*, respectively. The capacities of the queues Q_{AB} and Q_{AB} are assumed to be n.

Stage 3: Reachability analysis. Starting from the initial global state, the reachability analysis generates all possible succeeding global states by repeatedly executing a single transition (i.e., an event occurrence) in one of the protocol entities, and then reaching these new global states in turn until no new global states can be created. Let $S = (s_{11}, \ldots, s_{1k}, s_{21}, \ldots, s_{2k}, D_{12}, D_{21}, D'_{11}, \ldots, D'_{1(k-1)}, D''_{11}, \ldots, D''_{1(k-1)}, D'_{21}, \ldots, D'_{2(k-1)}, D''_{21}, \ldots, D''_{2(k-1)})$ be the current global state. The layered communication system may change the global state when a certain event occurs. Let $\underline{S} = (\underline{s}_{11}, \ldots, \underline{s}_{1k}, \underline{s}_{21}, \ldots, \underline{s}_{2k}, \underline{D}_{12}, \underline{D}_{21}, \underline{D}_{11}, \ldots, \underline{D}'_{1(k-1)}, \underline{D}''_{11}, \ldots, \underline{D}''_{1(k-1)}, \underline{D}'_{21}, \ldots, \underline{D}'_{2(k-1)}, \underline{D}''_{21}, \ldots, \underline{D}''_{2(k-1)})$ be the next global state of S. Also, let $NS_{Pij}(S)$ be the set of the possible next global states $\underline{S}$ that can be induced from S after a certain event occurs to protocol entity P_{ij}, and $NT_{Pij}(S)$ the set of executable transitions in protocol entity P_{ij} when the global state is S. The sets $NS_{Pij}(S)$ and $NT_{Pij}(S)$ can be determined from EM_{ij} and $PA_{ij,S}$ as follows (incidentally, we define a pair of binary operators "$\otimes$" and "$\oplus$," which operate on EM_{ij} and $PA_{ij,S}$ for computing $NS_{Pij}(S)$ and $NT_{Pij}(S)$, respectively).

$$NS_{Pij}(S) = EM_{ij} \otimes PA_{ij,S}$$
$$= \{\underline{S} = (\underline{s}_{11}, \ldots, \underline{s}_{1k}, \underline{s}_{21}, \ldots, \underline{s}_{2k}, \underline{D}_{12}, \underline{D}'_{11}, \ldots, \underline{D}'_{1(k-1)}, \underline{D}''_{11}, \ldots, \underline{D}''_{1(k-1)}, \underline{D}'_{21}, \ldots, \underline{D}'_{2(k-1)}, \underline{D}''_{21}, \ldots, \underline{D}''_{2(k-1)}) \mid$$

$\forall x_{ijw}$ satisfying that $PA_{ij,S}(x_{ijw}) = 1$ and $EM_{ij}(s_{ij}, x_{ijw}) \neq null$,
(1) $\underline{s}_{ij} = EM_{ij}(s_{ij}, x_{ijw})$,
(2) Case 1. x_{ijw} is a receiving event.

$y_{ijw} \cdot \underline{D}_{pi} = D_{pi}$ if $y_{ijw} \in M_{pi}$,
$y_{ijw} \cdot \underline{D}'_{ij} = D'_{ij}$ if $y_{ijw} \in M'_{ij}$,
$y_{ijw} \cdot \underline{D}''_{i(j-1)} = D''_{i(j-1)}$ if $y_{ijw} \in M''_{i(j-1)}$,

Case 2. x_{ijw} is a sending event.

$\underline{D}_{ip} = D_{ip} \cdot y_{ijw}$ if $y_{ijw} \in M_{ip}$,
$\underline{D}'_{i(j-1)} = D'_{i(j-1)} \cdot y_{ijw}$ if $y_{ijw} \in M'_{i(j-1)}$,
$\underline{D}''_{ij} = D''_{ij} \cdot y_{ijw}$ if $y_{ijw} \in M''_{ij}$,

where $\cdot$ denotes the concatenation of two packet sequences,

(3) The other components of $\underline{S}$ are the same as the corresponding ones of S.}

$$NT_{Pij}(S) = EM_{ij} \oplus PA_{ij,S}$$
$$= \{(s_{ij}, x_{ijw}) | EM_{ij}(s_{ij}, x_{ijw}) \neq null \text{ and } PA_{ij,S}(x_{ijw}) = 1 \text{ and if } x_{ijw} \text{ is a receiving event,}$$
$$y_{ijw} \cdot \underline{D}_{pi} = D_{pi} \quad \text{if } y_{ijw} \in M_{pi},$$
$$y_{ijw} \cdot \underline{D}'_{ij} = D'_{ij} \quad \text{if } y_{ijw} \in M'_{ij},$$
$$y_{ijw} \cdot \underline{D}''_{i(j-1)} = D''_{i(j-1)} \quad \text{if } y_{ijw} \in M''_{i(j-1)}.\}$$

Let $NS(S)$ be the set of all possible next global states that can be induced from S by a certain event. Since it is assumed that no two events occur simultaneously, $NS(S)$ can be determined as follows:

$$NS(S) = NS_{P11}(S) \cup NS_{P12}(S) \cup \cdots \cup NS_{P1k}(S) \cup NS_{P21}(S) \cup NS_{P22}(S) \cup \cdots \cup NS_{P2k}(S).$$

For example, $NS(S) = NS_A(S) \cup NS_B(S)$ for the example of Figure 8*a*. Assume $n > 1$. If $S = (0, 0, \varepsilon, \varepsilon)$, then

$$NS_A(S) = NS_A(0, 0, \varepsilon, \varepsilon) = EM_A \otimes PA_{A,S}$$
$$= \begin{pmatrix} 1 & - & 1 & - \\ - & 2 & 2 & 3 \\ - & - & 0 & - \\ - & - & 2 & - \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$
$$= \{(1, 0, 1, \varepsilon)\},$$
$$NS_B(S) = NS_B(0, 0, \varepsilon, \varepsilon) = EM_B \otimes PA_{B,S}$$
$$= \begin{pmatrix} 1 & - & 2 & - \\ - & - & 2 & - \\ - & - & 3 & 0 \\ - & 2 & - & - \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$
$$= \{(0, 1, \varepsilon, 3)\},$$
$$NS(S) = NS(0, 0, \varepsilon, \varepsilon) = NS_A(0, 0, \varepsilon, \varepsilon) \cup NS_B(0, 0, \varepsilon, \varepsilon)$$
$$= \{(1, 0, 1, \varepsilon), (0, 1, \varepsilon, 3)\},$$
$$NT_A(S) = NT_A(0, 0, \varepsilon, \varepsilon) = EM_A \oplus PA_{A,S}$$
$$= \begin{pmatrix} 1 & - & 1 & - \\ - & 2 & 2 & 3 \\ - & - & 0 & - \\ - & - & 2 & - \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$
$$= \{(0, -1)\},$$
$$NT_B(S) = NT_B(0, 0, \varepsilon, \varepsilon) = EM_B \oplus PA_{B,S}$$
$$= \begin{pmatrix} 1 & - & 2 & - \\ - & - & 2 & - \\ - & - & 3 & 0 \\ - & 2 & - & - \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 0 \\ 0 \end{pmatrix}$$
$$= \{(0, -3)\},$$

where an entry with "-" denotes a null value and an "ε" denotes an empty sequence. If $S = (1, 0, 1, \varepsilon)$, then

$$NS_B(S) = NS_B(1, 0, 1, \varepsilon) = EM_B \otimes PA_{B,S}$$
$$= \begin{pmatrix} 1 & - & 2 & - \\ - & - & 2 & - \\ - & - & 3 & 0 \\ - & 2 & - & - \end{pmatrix} \otimes \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$
$$= \{(1, 1, 1, 3), (1, 2, \varepsilon, \varepsilon)\}, \text{ and}$$
$$NT_B(S) = NT_B(1, 0, 1, \varepsilon) = EM_B \oplus PA_{B,S}$$
$$= \begin{pmatrix} 1 & - & 2 & - \\ - & - & 2 & - \\ - & - & 3 & 0 \\ - & 2 & - & - \end{pmatrix} \oplus \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$
$$= \{(0, -3), (0, +1)\}.$$

With the aid of $NS(S)$ and $NT_{pij}(S)$, we now proceed with the reachability analysis to generate all reachable global states and executable transitions. The algorithm for the reachability analysis is as follows:

```
step 1: G = G' = {S0};
        FOR i = 1 TO 2, j = 1 TO k DO
          R_Pij = { };
step 2: REPEAT
          L = {S' | S' ∈ NS(S) for all S ∈ G'};
          FOR i = 1 TO 2, j = 1 TO k DO
            BEGIN
              R'_Pij = {(s, x) | (s, x) ∈ NT_Pij(S)
                       for all S ∈ G'};
              R_Pij = R_Pij ∪ R'_Pij;
            END;
          G' = L - G;
          G = G + G';
        UNTIL G' is empty.
```

In the above algorithm, G represents the set of all reachable global states starting from the initial global state S_0, and R_{Pij} represents the set of all executable transitions in protocol entity P_{ij}. The sets of G and R_{Pij} contain all the necessary information for detecting the logical errors of the layered communication system.

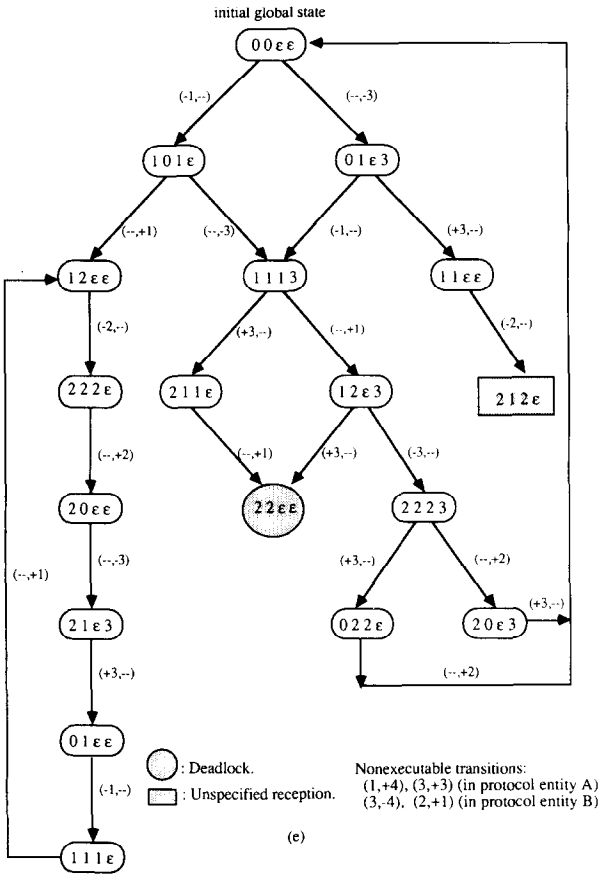

Figure 8. *e*, State transition diagram.

The proposed algorithm is to generate all the reachable global states and the executable transitions by augmenting G with G' and R_{Pij} with R'_{Pij} in each iteration, where G' and R'_{Pij} are the sets of newly generated reachable global states and executable transitions, respectively. The sets G' and R'_{Pij} are determined by finding the sets $NS(S)$ and $NT_{Pij}(S)$, respectively, for each reachable global state S that has just been generated in the last iteration.

Consider again the example of Figure 8*a*. Let e_A and e_B denote the events that may occur to protocol entities A and B, respectively, when the global state is $S = (a, b, X, Y)$. Also, denote $S = (a, b, X, Y)$ as the next global state of S (see Figure 8*d*). Figure 8*e* shows the corresponding state transition diagram of the reachability analysis with queue capacities equal to one ($n = 1$), where each node uniquely represents a reachable global state and each arch represents an event occurrence (transition).

After generating G and R_{Pij}, we can detect the logical errors of the layered communication system as follows. (Let $S = (s_{11}, \ldots, s_{1k}, s_{21}, \ldots, s_{2k}, D_{12}, D_{21}, D'_{11}, \ldots, D'_{1(k-1)}, D''_{11}, \ldots, D''_{1(k-1)}, D'_{21}, \ldots, D'_{2(k-1)}, D''_{21}, \ldots, D''_{2(k-1)} \in G$, and $x = (s_{ij}, x_{ijw})$ is a transition in protocol entity P_{ij}.

(1) S is a deadlock state if

$$NS(S) = \{\ \} \text{ and}$$

$$D_{12} = D_{21} = D'_{11} = \cdots = D'_{1(k-1)} = D''_{11} = \cdots = D''_{1(k-1)} = D'_{21} = \cdots = D'_{2(k-1)} = D''_{21} = \cdots = D''_{2(k-l)} = \varepsilon.$$

(2) S is an unspecified reception state if

$$NS(S) = \{\ \} \text{ and}$$

for all y_{ijw} satisfying $y_{ijw} \cdot \underline{D}_{pi} = D_{pi}$ or

$$y_{ijw} \cdot \underline{D}'_{ij} = D'_{ij} \text{ or}$$

$$y_{ijw} \cdot \underline{D}''_{i(j-1)} = D''_{i(j-1)},$$

$EM_{ij}(s_{ij}, x_{ijw})$ is not defined.

(3) x is a nonexecutable transition if

$$EM_{ij}(s_{ij}, x_{ijw}) \neq null \text{ and } (s_{ij}, x_{ijw}) \notin R_{Pij}.$$

6. EXPERIENCE WITH THE LAYERED COMMUNICATION SYSTEM GENERATOR

The layered communication system generator described here has been used to generate a variety of layered communication systems, including protocols for the data link, network, transport session, and application layers of the ISO/OSI reference model as summarized in Figure 9. These layered communication systems were specified by the L_CFSMs. We have not only implemented these real-world protocols automatically, but also migrated these layered communication systems into several target machines, which are summarized in Figure 10. In Figure 10, Intel 80286 and 80386 are

Layer	Protocol Entity
7.application	FTP, SMTP, TELNET
6.presentation	
API	TLI, SOCKET
5.session	TCP, UDP
4.transport	
3.network	IP, ARP, RARP, ICMP
2.datalink	LLC (IEEE 802.2)
1.physical	IEEE 802.3

Figure 9. Layered communication systems generated by the layered communication system generator.

Layer	target 1	target 2	target 3	target 4
7.application	Intel 80286 (DOS)	Intel 80386 (UNIX)	Intel 80286 (DOS)	Intel 80386 (UNIX)
6.presentation				
API			Intel 80186	Intel 80186
5.session				
4.transport				
3.network				
2.datalink				
1.physical	Intel 82586			

Figure 10. Environments where the layered communication systems are developed.

hosts and Intel 80186 and 82586 are front-end processors (Intel 82586 chip is devoted to IEEE 802.3). DOS and UNIX were run in Intel 80286 and 80386, respectively. Protocol entities were first developed in the host, and then either all of them stayed in the host or some of them, belonging to the application interface (API), session transport, network, and datalink layers, were migrated into the Intel 80186 front-end processor (i.e., the intelligent network card). Since queues were used for interlayer synchronization, the migration was easily achieved.

The most complex protocol entity in our implementation is the transmission control protocol (TCP) of Department of Defence (DoD). TCP performs the function of establishing sessions between user processes on the internet and ensures reliable communications by implementing error recovery procedures on an end-to-end basis [13]. The detailed listings of states, events, and action primitives from our TCP implementation are shown in Appendices A and B. The event and action matrices are shown in Appendices C and D. In our implementation, there are 11 states, 17 upper layer events, 4 timer events, 13 lower layer events, and 19 hand-code primitives (i.e., the nonlogical parts). Each primitive has about 30 lines of C codes.

7. DISCUSSION AND CONCLUSION

On the basis of a generalized communication system framework, we have proposed a layered communication system generator. This framework contains queues, a switcher, memory manager, time manager, protocol entity generator, and hand-code primitive library. The proposed layered communication system generator can automatically generate layered communication systems (except for the nonlogical parts). The layered communication systems were specified by the L_CFSMs, in which each protocol entity was again specified by an MFSM. The protocol entity generator transformed each MFSM into an event matrix and an action matrix. The behavior of each protocol entity was expressed by these two matrices. We have also presented a matrix approach to the validation of the protocol entities.

We have physically established the generalized communication system framework on target machines (Intel 80286 and 80386) to validate the proposed layered communication system generator. Several layered communication systems with real-world protocols from DoD and IEEE 802 standards organizations have been successfully developed by the proposed layered communication system generator.

In summary, the proposed layered communication system generator has the following features:

1. Automatic generation of layered communication systems, except for the nonlogical parts (the hand-code primitives).
2. Enhanced ability of specifying layered communication systems.
3. Enhanced reliability of layered communication systems.
4. Easy maintenance of layered communication systems.
5. Easy migration of layered communication systems.
6. Considerable reduction in the overhead that arises from the implementation of layered communication systems.

In addition, the proposed layered communication system generator can be easily adapted to multiple stream connections between computers and the multiple priority scheme [14]. For the former, we only need to add an additional index to the components of the L_CFSM (for example, S_{ij} now becomes S_{ijk}), to specify layered communication systems. For the latter, we need to provide multiple priority queues between layers such that the multiple priority scheme can be supported. Each packet will be put into the priority queue with the same priority level.

Finally, our future research directions include automation and generalization of the nonlogical parts, a matrix approach to handling the state explosion problem [15], and a new strategy for the switcher to improve the productivity of layered communication systems.

ACKNOWLEDGMENT

We thank the anonymous referees for their helpful suggestions and comments. We also thank Professors Yu-Chin Hsu and Maw-Sheng Chern for their careful reading of the manuscript.

APPENDIX A: TCP States and Events

Code	State name
S1	CLOSED
S2	LISTEN
S3	SYN SENT
S4	SYN RECVD
S5	ESTAB
S6	CLOSEWAIT
S7	LASTACK
S8	FINWAIT1
S9	FINWAIT2
S10	CLOSING
S11	TIMEWAIT

Figure A1. Listing of TCP states.

Code	Event name
U1	unspecified passive open
U2	fully specified passive open
U3	active open
U4	active open with data
U5	send data
U6	close
U7	abort

Figure A2. Listing of TCP events from upper layer entity.

Code	Event name
T1	retransimition timeout
T2	upper layer protocol timeout
T3	timewait timeout
T4	acknowledgement timeout

Figure A3. Listing of TCP events from TIME MANAGER.

Code	Event name
L1	synchronization
L2	acknowledgement
L3	security not match
L4	reset
L5	synchronization, acknowledgement
L6	invalid acknowledgement
L7	acknowledgement of synchronization
L8	security, precedence not match
L9	final, acknowledgement of synchronization
L10	out of sequence no.
L11	acknowledgement of final
L12	final, acknowledgement
L13	final

Figure A4. Listing of TCP events from lower layer entity.

APPENDIX B: Listing of TCP Action Primitives

Code	Short name	Full format in TCP
A	accept	T_accept(sv,dT_blk)
B	dispatch	T_dispatch(sv,dT_blk)
C	generate SYN	T_genSYN(sv)
D	open	T_open(sv,rT_blk)
E	open fail	T_openfail(sv)
F	open success	Td_xnsfer(sv)
G	partial reset	T_partRST(sv)
H	record SYN	T_recSYN(sv,dT_blk)
I	reset	T_RST(sv)
J	reset self	T_RSTslf(sv)
K	restart time wait	T_rstarttw(sv)
L	retransmit	T_rxmt(sv)
M	save send	T_savesnd(sv,rT_blk)
N	send ack	T_sndack(sv)
O	send final	T_sndFIN(sv)
P	set final	T_recFIN(sv)
Q	start time wait	T_starttw(sv)
R	update sliding window	T_UPDsw(sv,dT_blk)
S	stop upper layer receive	T_stopULPrec(sv)

sv : state vector of TCP
dT_blk : the packet from lower layer entity
rT_blk : the packet from upper layer entity

APPENDIX C: TCP Event Matrix

	U1	U2	U3	U4	U5	U6	U7	T1	T2	T3	T4	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13
S1	S2	S2	S3	S3																				
S2						S1	S1					S4	S2	S2										
S3					S3	S1	S1		S1			S4		S3	S1	S5	S3							
S4					S4	S8	S1	S4	S1		S4	S1			S1		S4	S5	S1					
S5					S5	S8	S1	S5	S1		S5	S1	S5		S1		S5		S1	S6	S5			
S6					S6	S7	S1	S6	S1			S1	S6		S1		S6		S1		S6			
S7							S1	S7	S1			S1	S7		S1		S7		S1		S7	S1		
S8						S8	S1	S8	S1		S8	S1	S8		S1		S8		S1		S8	S9	S11	S10
S9						S9	S1	S9	S1		S8	S1	S9		S1		S9		S1		S9			S11
S10						S10	S1	S10	S1			S1	S10		S1		S10		S1		S10	S11		S10
S11						S11	S1			S1		S1			S1		S11		S1		S11		S11	

APPENDIX D: TCP Action Matrix

	U1	U2	U3	U4	U5	U6	U7	T1	T2	T3	T4	L1	L2	L3	L4	L5	L6	L7	L8	L9	L10	L11	L12	L13
S1	D	D	DC	DC																				
S2						JS	J					HC	I	I										
S3					M	JS	J		E			HN		I	E	HFN	I							
S4					M	OS	IJ	L	IE		N	IE			E		I	ARF	IG					
S5					B	OS	IJ	L	IE		N	EI	AR		E		N		IE	ARPN	N			
S6					B	OS	IJ	L	IE			IJ	SN		J		N		IE		N			
S7							J	L	IE			IJ	SN		J		N		IE		N	J		
S8						S	IJ	L	IE		N	IJ	AR		J		N		IE		N	AR	ARPNQ	ARPN
S9						S	IJ	L	IE		N	IJ	A		J		N		IE		N			SPNQ
S10						S	IJ	L	IE			IJ	R		J		N		IE		N	Q		RN
S11						S	J			J		IJ			J		N		IJ		N		NK	

The execution sequence of the primitives in each entry are from left to right and from top to bottom.

(e.g. The execution sequence of the primitives in entry (S8,L12) is A, R, P, N, Q.)

REFERENCES

1. S. Aggarwal and R. P. Kurshan, Automated implementation from formal specification, in *Proceedings of I.F.I.P. International Workshop on Protocol Specification, Testing, and Verification*, 1985, pp. 127-136.
2. D. P. Anderson, Automated Protocol Implementation with RTAG, *IEEE Trans. Software Eng.* 14, 281-300 (1988).
3. T. P. Blumer and D. P. Sidhu, Mechanical Verification and Automatic Implementation of Communication Protocols, *IEEE Trans. Software Eng.* SE-12, 827-843 (1986).
4. G. Jholzmann, The Pandora System: An Interactive System for the Design of Data Communication Protocols, *Comp. Net.* 8, 71-79 (1984).
5. C. V. Ramamoorthy, C. T. Dong, and Y. Usuda, An Implementation of an Automated Protocol Synthesizer (APS) and Its Application to the X.21 Protocol, *IEEE Trans. Software Eng.* SE-11, 886-908 (1985).
6. N. Shiratori, K. Takahashi, and S. Noguchi, A Software

Design Method and Its Application to Protocol and Communication Software Development, *Comp. Net. ISDN Syst.* 15, 245-267 (1988).

7. S. T. Vaong, D. D. Hui, and D. D. Cowan, VALIRA—a tool for protocol validation via reachability analysis, in *Proceedings of 7th International Symposium on Protocol Specification, Testing, and Verification*, 1988, pp. 35-41.
8. P. Zafiropulo, C. West, H. Rudin, D. D. Cowan, and D. Brand, Towards Analyzing and Synthesizing Protocols, *IEEE Trans. Commun.* COM-28, 651-661 (1980).
9. T. T. Lee and M. Y. Lai, A Relational Algebraic Approach to Protocol Verification, *IEEE Trans. Software Eng.* 14, 184-193 (1988).
10. D. Brand and P. Zafiropulo, On Communicating Finite-State Machines, *J. ACM* 30, 323-342 (1983).
11. C. H. Chow, M. G. Gonda, and S. S. Lam, On constructing multi-phase communication protocols, in *Proceedings of I.F.I.P. International Workshop on Protocol Specification, Testing, and Verification*, 1985, pp. 57-68.
12. A. Finkel, A new class on analyzable CFSMs with unbounded FIFO channels, in *Proceedings of 8th International Symposium on Protocol Specification, Testing, and Verification*, 1988, pp. 283-294.
13. *DDN Protocol Handbook – DoD Military Standard Protocols*, vol. 1, DNN Network Information Center, Menlo Park, California, 1985.
14. *UNIX System V Release 4 – Programmer's Guide: STREAMS*, UNIX Software Operation, Prentice-Hall, Englewood Cliffs, New Jersey, 1990.
15. M. T. Liu, Protocol Engineering, *Adv. Comp.* 29, 79-195 (1989).