COMMUNICATION SYSTEM GENERATOR ON LAYERED COMMUNICATING FINITE STATE MACHINE

Yung-Chen Hung and Gen-Huey Chen

Department of Computer Science and Information Engineering,

National Taiwan University, Taipei, Taiwan

Abstract According to the seven-layer OSI (Open System Interconnection) reference model, a communication system can be logically decomposed into layers, each responsible for a specific set of protocol entities. Many approaches have been proposed for the formal specification, validation, and implementation of protocol entities. However, approaches are still lacking for communication systems because the inter-layer synchronization is not clearly defined. This paper presents an approach for automatic communication system development and discusses a semiautomatic implementation of a real-word communication system. We first give a generalized communication system framework (GCSF), which is a universal environment for various communication systems. We then introduce a layered communicating finite state machine (L_CFSM) to specify communication systems. Given an L_CFSM, a communication system generator can automatically produce an executable communication system, based upon the GCSF. The communication system generator is a process of developing communication systems.

We have implemented a communication system generator on DOS and UNIX enviornments, in which a communication system with several real-word protocols from DoD and IEEE 802 protocol standards organizations has been sucessfully developed.

1. Introduction

The objective of developing a communication system generator is to provide a systematic way of designing communication systems so that their correctness can be ensured. According to the seven-layer OSI (Open System Interconnection) reference model, which was proposed by the International Standards Organization (ISO), a communication system can be logically decomposed into layers, each responsible for a specific set of protocol entities. In this paper we concentrate on the layered communication systems. Based on such systems, we first propose a Generalized Communication System Framework (abbreviated to GCSF), which is suitable for a variety of target machines. The GCSF includes queue structure, dispatcher, memory manager, time manager, primitive library, and protocol entities. It provides a universal executable environment for various communication systems.

The problem of specifying communication systems is much more difficult than that for the classical (sequential) systems. The difficulties are due to the necessity of describing several sequential components that may then cooperate and execute in parallel.

The development of a communication system should begin with a formal specification and then proceed with an automatic implementation, in order to attain good reliability and productivity. There are two major requirements for the specification: one is to facilitate the implementation of the communication system and the other is to perform the validation easily.

In general, the existing formal specifications are mainly classified into three categories. The first category is the state-transition model, including finite-state automata (FSA), formal grammars, and Petri nets. The second category is the programming language model, including abstract programs, temporal logic, and abstract data types. The third category are those hybrid models that contain both states and language constructs in the specification of protocols. Extended finite-state machine (EFSM), Estelle, LOTOS, SDL and FAPL are five examples.

It is easy to validate and automate a state-transition model with a small number of states and events. However, an unworkably large number of events and states may be contained in a realistic protocol, thereby creating the so-called *state explosion* problem. Depending on how high level and abstract a language is used, a programming language model may be quite near to an implementation of the protocols. However, efforts to prove the correctness of a program far exceed those required for developing the program, and the proof usually depends heavily on human ingenuity and is hard to automate. The hybrid model attempt to combine the advantages of the above two models, in order to achieve an easy validation and automatic implementation of the communication systems[3].

A Layered Communicating Finite State Machine (abbreviated to L_CFSM) is a language for specifying communication systems with layered architectures. It typically uses a small number of states to capture only the main features of a protocol, with each state being augmented with context variables and primitives. The L_CFSM consists of two parts: the state-transition part and the program part. The state-transition part captures the control aspect of the protocol, while variables and data are easily handled by the program part. A major feature of the L_CFSM is that it can directly download the specification to a target machine established with a GCSF, therefore setting up an executable layered communication system without any human involvement. Besides, the L_CFSM also provides upward and downward multiplexing. The L_CFSM has the advantages of both the state-transition and the programming language models.

Progress has been made in creating an integrated set of tools for automatic protocol design. These tools are useful in the specification, validation and implementation of protocols, and they are dedicated to implementing a protocol entity [4], [5], [6]. The objective of this paper is to design a communication system generator such that a layered communication system, not only a protocol entity, can be automatically implemented from an L_CFSM.

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

92CH3132-8/0000-0466 $3.00 © 1992 IEEE

(1) Istall the GCSF into the given target machine,
(2) Input the L_CFSM of the communication system,
(3) Transform protocol entities into event matrices and action matrices,
(4) Validate the communication system to ensure they are logically error-free, and
(5) Download event matrices and action matrices to the target machine.

The communcation system is established after step (5). Note that the validation of a communication system is to ensure their interactions are complete, live, deadlock-free, and free from overflow. 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 protocol entities are waiting for each other forever. The freedom from overflow guarantees the number of packets in a queue does not exceed its capacity. Many effective methods [4] have been proposed for protocol validation. Some of them are suitable for the state-transition model; they also do well for our model.

2. Generalized Communication System Framework (GCSF)

In this section, we take an overview on the GCSF (see Figure 2), which is a universal framework for a variety of communication systems and is suitable for heterogeneous communication environments. In the GCSF, a set of modules are proposed for inter-layer synchronization, inter-layer flow control, memory management, time management, event interpretation, and protocol entity driving. The GCSF also provides multi-user, multi-medium, multi-stream, and multiplex capacity. In Figure 2, we assume that l is the number of communication media, k is the number of users, and $r(i)$ is the number of protocol entities in layer i. The framework mainly consists of unidirectional FIFO queues, a dispatcher, a memory manager, a time manager, protocol entities, predicate library, and primitive library. The function of each module is described as follows.

Queue Structure

The unidirectional FIFO queues are used for inter-module, inter-layer, and inter-computer synchronization.

Each protocol entity can send (receive) packets into (from) its outgoing (incoming) queues. Each outgoing queue of a protocol entity is an incoming queues of another protocol entity. In the GCSF, each protocol entity has two incoming queues: one for receiving packets from upper layer protocol entities, and the other for receiving packets from lower layer protocol entities. All protocol entities share a pair of queues for communication with the time manager, and the packets from (into) the time manager pass through the same service queue. Note that the proposed GCSF can be easily extended to handling multiple priority queues, in which multiple incoming queues are required between layers.

Dispatcher

As mentioned above, each protocol entity is represented by a separate process. Therefore, how to switch the CPU between these processes is important. If the layered communication system runs in a single-user environment (e.g., DOS), a dispatcher is needed to allocate the CPU among these processes. On the other hand, if the layered communication system runs in a multi-user environment (e.g., UNIX), a new dispatcher is still needed because the original operating system scheduling criteria do not fit the communication system. The dispatcher always allocates the CPU to the process with the highest priority. The priorities of processes are usually determined by the number of packets in the incoming queues, the urgency of packets, and whether timeout events are received. In particular, when a timeout event occurs to a protocol entity and the queue from the time manager to the protocol entity is not empty, the protocol entity has the highest priority.

The dispatcher is a central component in a communication system. When it gets the CPU to run, it switches the CPU to the highest-priority protocol entity, which then removes a packet from the most urgent incoming queue, called *dequeue*, processes them, inserts some packets into outgoing queues, called *enqueue*, and finally returns the CPU to the dispatcher. By switching the CPU among the protocol entities, the dispatcher can make the communication system more productive and the inter-layer flow more smooth.

Memory Manager

For convenient use of the communication system, we provide a uniform logical view of memory utilization. The memory manager defines a fixed-size block (16, 256, 512, or 1024 bytes) 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 when it is invoked will allocate (free) a logical memory unit. The enqueue routine and dequeue routine 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 communication system, some time-related mechanisms such as retransmission, timeout and piggyback are necessary. The proposed framework provides a time manager for the following purposes: start a timer, stop a timer, keep track of timers, and send timeout signals to protocol entities. The interface between the time manager and the protocol entities is a pair of queues, through which start and stop events are transmitted from protocol entities to the time manager and timeout events are transmitted in reverse direction.

Protocol Entities

As mentioned in [1], a protocol entity may be divided into two parts: non-logical part and logical part.

(1) *Non-logical part.* Some tasks in a protocol entity such as formatting, encoding, decoding, error-checking, syntax-ckecking, fragmentation, reassembly and synchronizing can be considered "non-logical". Likewise, the predicates for the event interpretation can also be considered "non-logical". Their implementation may require special considerations concerning the target machine. Such non-logical components are specified by the programming language model in the L_CFSM. In our implementation, a predicate library and a primitive library are provided to support non-logical functions of protocol entities, and an action matrix is used to drive the primitives.

(2) *Logical part.* The remaining part of a protocol entity is considered "logical". It comprises the "brain" of the protocol entity and is specified by the state-transition

model in the L_CFSM. We use the state transition model to describe the behavior of the protocol entity. Each state of the state-transition model corresponds to a different control state of the protocol entity. Each transition of the model is related to an input event, or an output event, or a time event that enables the transition. In our implementation the state-transition model is represented by an event matrix.

In the proposed communication system generator, the event matrix and the action matrix can be automatically generated from the L_CFSM by a protocol entity compiler, which will be described in Section 3.

3. Layered Communication Finite State Machine (L_CFSM)

In this section, we introduce the L_CFSM, which is used to describe the communication systems. The communication systems are assumed having p nodes, q layers in each node, and $r(ij)$ protocol entities in the jth layer of node i.

Definition 1: An *L_CFSM* is a fourteen-tuple

$(<S_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<A_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<V_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<H_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<I_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<EP_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<O_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<M_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<M'_{ijk}>_{i=1,...,p,j=1,...,(q-1),k=1,...,r(ij)},$
$<C_{ijk}>_{i=1,...,p,j=1,...,q,k=1,...,r(ij)},$
$<C'_{ijk}>_{i=1,...,p,j=1,...,(q-1),k=1,...,r(ij)},$
$TC,\quad TC',\quad SUCC),$

where

S_{ijk} is the state of the kth protocol entity (denoted by P_{ijk}) of the jth layer in node i,

A_{ijk} is the set of primitives executed by P_{ijk},

V_{ijk} is the data structure for representing the state vector of P_{ijk},

H_{ijk} is the data structure for representing the header of packets in P_{ijk},

I_{ijk} is the data structure for representing the interface of packets in P_{ijk},

EP_{ijk} is the set of predicates for P_{ijk},

O_{ijk} is the initial state of P_{ijk},

M_{ijk} is the set of packets which P_{ijk} can receive from the upper layer through a specified queue, denoted by Q_{ijk},

M'_{ijk} is the set of packets which P_{ijk} can receive from the lower layer through a specified queue, denoted by Q'_{ijk},

C_{ijk} is the capacity of Q_{ijk},

C'_{ijk} is the capacity of Q'_{ijk},

TC is the capacity of the queue, denoted by TQ, from protocol entities to the time manager,

TC' is the capacity of the queue, denoted by TQ', from the time manager to protocol entities, and

$SUCC$ is the set of transitions that are represented by a partial function from $S_{ijk}\times E_{ijk}$ to $S_{ijk}\times 2^{A_{ijk}}$, where E_{ijk} is the set of events that can occur to protocol entity P_{ijk}.

The states of protocol entities and the contents of queues will change during the execution of the communication system. In the following, we define the global state of the communication system.

Initially, each protocol entity P_{ijk} is in state O_{ijk} and all the queues are empty. The communication system can generate all possible *interaction paths* staring from the initial global state through some intermediate global states, called *reachable global states*, to the final global states. The generated graph is often called *state transition diagram*. In a state transition diagram, a new global state will be generated from the current global state if a sending event, or a receiving event, or a time event occur to a protocol entity.

The function of protocol validation is to detect potential logical errors in an L_CFSM. There are four types of logical errors: deadlock, unspecified reception, nonexecutable transition, and queue overflow. A deadlock is a global state in which only receiving events or timeout event are defined, but all the queues are empty. Thus, if the 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 or timeout event are defined, not all the queues are empty, but the first packet in each nonempty queue is not specified in the receiving events or timeout event. In other words, no 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 occurence) in a protocol entity that never occurs during the execution and it exists due to some inconsistencies or syntatic errors in the design of the L_CFSM. A queue overflow is a reachable global state in which the number of packets to be put into a queue exceeds its capacity.

Many effective methods [8] have been proposed for protocol validation. Some of them are suitable for the communication system specified by the state-transition model, they can also do well for our model. In particular, a matrix approach combining reachability analysis, which has been proposed in [6], is a good way for the protocol validation of the L_CFSM.

4. Protocol Entity Compiler

As discussed in the previous section, the logical part of a protocol entity is modeled as a finite state transition diagram. The finite state transition diagram can be represented by two matrices: event matrix and action matrix. These two matricies are good for understanding the meaning of the protocol entity, detecting logical errors and driving the protocol entity by the GCSF. A protocol entity compiler is responsible for transforming the logical part of a protocol entity in the L_CFSM into event and action matrices.

After the protocol entitis are transformed into matrices,

protocol validation can be performed on the event matrices [6], instead of on protocol entities themselves. If the communication system is logically error-free, then these matrices are downloaded to the GCSF that has been installed in the taget machine. After that, the communication system is well-established. Otherwise, the L_CFSM should be modified.

In the next section, how to drive protocol entities and determine what events occur to protocol entities are described.

5. Experience with the Communication System Generator

The communication system generator described in this paper has been used to generate a variety of communication systems, including protocols for the data link, network, transport, session and application layers of the ISO/OSI reference model as summarized in Figure 3. These communication systems were specified by the L_CFSM. 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 4. In Figure 4, Intel 80286 and 80386 are hosts, and Intel 80186 and 82586 are front-end processors (Intel 82586 chips 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, which belong 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 inter-layer synchronization, the migration was easily achieved.

The most complex protocol entity in our implementation is TCP (Transmission Control Protocol) of DoD (Department of Defence). 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. The detailed listings of states, events and action primitives from our TCP implementation are shown in [7].

6. Discussion and Conclusion

In this paper, on the basis of the GCSF, we have proposed a communication system generator. The GCSF contains queues, a dispatcher, a memory manager, a time manager, a protocol entity complier and a primitive library. The proposed communication system generator can automatically generate communication systems (except for the non-logical parts). The communication systems are specified by the L_CFSM. The protocol entity complier can transform the logical parts of the protocol entities into the event and action matrices. The behavior of each protocol entity is expressed by these two matrices.

We have physically established the GCSF on several target machines (Intel 80286 and Intel 80386) to validate the proposed 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 communication system generator.

In summary, the proposed communication system generator possesses the following features:

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

References

1. S. Aggarwal and R. P. Kurshan, Automated implementation from formal specification, in *Proceeding 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 Transactions on Software Engineering*, vol. 14, no. 3, March 1988, pp. 281-300.
3. D. Brand and P. Zafiropulo, On communicating finite-state machines, *Journal of the Association for Computing Machinary*, vol. 30, no.2, April 1983, pp. 323-342.
4. C. H. Chow, M. G. Gonda, and S. S. Lam, On constructing multi-phase communication protocols, in *Proceeding of I. F. I. P. International Workshop on Protocol Specification, Testing, and Verification*, 1985, pp. 57-68.
5. H. M. Deitel, *An introduction to operating systems*, Addison-Wesley publishing company, 1984.
6. Y. C. Hung and G. H. Chen, A layered communication system generator, in *Proceeding of IEEE Conference on Communications Software: Communication for Distributed Application & System*, May 1991, pp. 115-130.
7. Y. C. Hung and G. H. Chen, Communication system generator on layered communicating finite state machine, Tech. Rep., Department of Computer Science and Information Engineering, National Taiwan University, Taiwan, 1991.
8. M. T. Liu, Protocol engineering, *Advances in Computers*, vol. 29, Academic Press, San Diego, 1989, pp. 79-195.

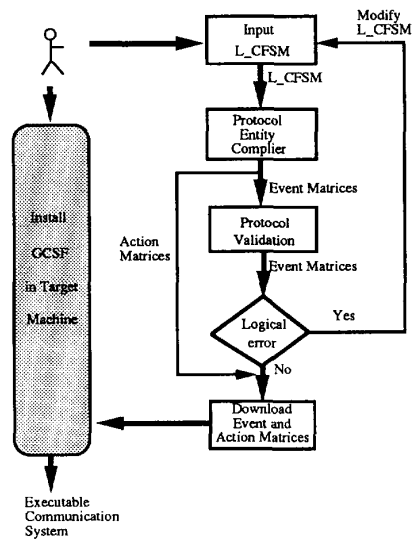

Figure 1. An overview of the communication system generator.

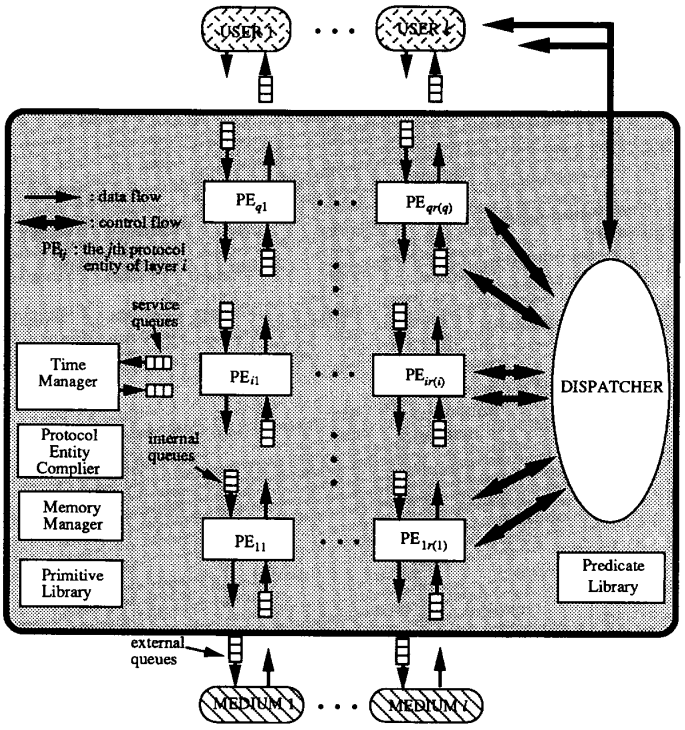

Figure 2. A Generalized Communication System Framework (GCSF).

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	X.25
2. datalink	LLC (IEEE 802.2)	LAP B
1. physical	IEEE 802.3	X.21 bis

: have been developed. : will be developed.

Figure 3. The communication systems generated by the 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 4. The environments where the communication systems are developed.