CSP-BASED SPECIFICATIONS FOR NETWORK PROTOCOLS AND SERVICES*

Nien-Chen Liu and Ming T. Liu

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210

ABSTRACT

Formal specifications of network protocols and services can act as a bridge between the designer and the implementor of computer communications networks. During the past years a number of formal specification techniques have been developed to specify either protocols or services, but they cannot be used to specify both in a uniform way. In this paper, we propose a formal technique, based on Communicating Sequential Processes (CSP), that can be used to specify both protocols and services in a uniform way. We demonstrate the applicability of the proposed technique by using it to specify two well-known examples, the Alternating Bit Protocol and a simplified version of the OSI Transport Service. Also presented is a comparison with other formal techniques.

1. INTRODUCTION

Rapid advances in computer communications result in the need for standardization in worldwide information systems. After several years of efforts, the International Standards Organization (ISO) has developed a basic architecture for distributed information systems called the Open Systems Interconnection (OSI) Reference Model [1]. Based on this model, the concepts of protocol specification and service specification have been introduced. They represent two different levels of abstraction in the OSI model [2]. A protocol specification describes the behavior of protocol entities in terms of the interactions between peer entities, and the operations in response to the service requests invoked by the layer above or the service indications issued by the layer below. A service specification, at a higher level of abstraction, defines the allowed sequences of interactions visible at the boundary (logical interface) between users of the service and the underlying service provider.

It has long been recognized that informal descriptions are not adequate to concisely and precisely describe the rules of communications between information systems due to their complexities. A number of formal specification techniques based on finite state machines, Petri nets, programming languages, and more recently, temporal logic [3] and Calculus of Communicating Systems (CCS) [4], have been proposed for modeling communication systems. A formal specification technique is required to have unambiguous constructs so as to provide good readability and to support modularity. Besides these common requirements, it should also be able to specify both protocols and services in a uniform way. Unfortunately, current techniques are only appropriate for specifying either protocols or services but not both. This is because protocols and services, as mentioned above, represent two different levels of abstraction, and one technique cannot be used to specify both in a uniform way. For example, the models based on the extended finite state machine [5, 6] are more suitable to specify protocols whereas the models based on CCS [7, 8] are more suitable to specify services. To facilitate protocol verification, however, formal techniques that can be used to specify both in a uniform way are needed and efforts in developing such new techniques are under way.

In this paper, we present a formal technique, based on the notions of Communicating Sequential Processes (CSP) [9], that can be used to specify protocols and services in a uniform way. We will demonstrate this technique by using it to specify two well-known examples, the Alternating Bit Protocol (ABP) and a simplified version of the OSI Transport Service.

*Research reported here was supported by U.S. Army CECOM, Ft. Monmouth, NJ, under Contract No. DAAAB07-83-K-5542. The views, opinions and/or findings contained in this paper are those of the authors and should not be construed as an official Department of the Army position, policy or decision.
In the next section, the motivations of using the CSP language as the basis of our specifications are presented. In section 3, we specify the Alternating Bit Protocol using CSP notations. In section 4, we present the specification of a simplified version of the OSI Transport Service and the associated modification to CSP. In section 5, a discussion and a comparison with other techniques are presented.

2. THE CSP LANGUAGE

Communicating Sequential Processes (CSP) [9] is a high level concurrent language designed for systems with multiple processors. A CSP program consists of a number of processes that are mutually disjoint in address spaces and communications between processes are accomplished through message passing. In addition, guarded commands are used to describe nondeterministic behavior. It can be seen that those features of CSP provide a suitable basis to model a distributed environment.

After CSP was first introduced by Hoare in [9], it has stimulated subsequent research work in many aspects. The concepts and notations of CSP have greatly influenced the design of concurrent programming languages [10]. Several proof systems for CSP [11, 12, 13] have been proposed, each exploring an alternative way of reasoning the behavior of concurrent programs. Moreover, CSP notations have also been used in the design of VLSI algorithms [14], data base systems [14], and operating systems [15].

We think what makes CSP a distinguished language is its simplicity and effectiveness in modeling distributed systems. As mentioned above, the message passing primitives provide an elegant mechanism for communication and synchronization, and the alternative and repetitive primitives employing guarded commands provide powerful control constructs for nondeterminism. An important philosophy underlying the design of CSP as emphasized by Hoare is that CSP was by no means intended to be a complete programming language and only a minimum set of essential primitives was included. Further extensions or modifications may be required to fit specific purposes.

While CSP was not intended to be a realistic programming language, we think it is exactly this reason which makes CSP a good basis for a specification language. In our approach to specifying protocols and services, we use the basic structures of CSP as a backbone and make either some extensions or modifications such that a variety of communication protocols and services with distinct properties can be specified in a uniform yet flexible way. In this paper we assume the reader is familiar with the CSP language [9]. Also, we adopt the already widely accepted extension which allows output commands to appear in guards.

3. ALTERNATING BIT PROTOCOL

There are several versions of the Alternating Bit Protocol in the literature. The version used in this paper is taken from [5]. It provides a one way data transmission over a half-duplex transmission medium which occasionally may corrupt or lose messages. As shown in Figure 1, two protocol entities, Sender and Receiver, together with a transmission medium and a timer, form a communication system for User1 and User2. When Sender gets a data block from User1, it attaches a single bit sequence number to the data block and transmits this frame one or more times until the reception of a correct acknowledgement. A timer is used to detect any loss of frame or acknowledgement occurred during transmission. It is assumed that the time-out interval is properly set such that time-out will only occur after a transmission loss has happened. On the other side, Receiver keeps checking the sequence number of each coming frame. If a frame has the expected sequence number, Receiver then delivers the data block of that frame to User2 and sends an acknowledgement back to Sender. Otherwise, it just responds with sending the last acknowledgement.

![Figure 1: The Structure of the ABP communication system](image)

The specification of ABP is shown in Figure 2. Each system component is modeled by a process and the whole communication system is described by all the processes running in parallel. From this simple example, we can see the nicest thing about using CSP is that it provides a very natural way to describe the properties of a protocol in both semantic and control aspects. The semantic aspect usually involves changing values of variables, whereas the control aspect involves the interactions between system components. It has been widely recognized that the programming language model is suitable to deal with semantic properties whereas the finite state machine model is suitable to express control properties. In
order to take advantage of both approaches, extended finite state machine models [5, 6] have been proposed, which enhance a finite state automaton by associating a program segment (called action or operation) with each transition. A transition may occur from a major state to another when an interaction happens and its associated enabling predicate is satisfied. The program segment associated with that transition is executed and may change values of the context variables or initiate another interaction.

By carefully examining the APB example above, the reader may notice that the CSP specification, which is a programming language model, captures many features of the extended finite state machine model. A comparison between the extended finite state machine (EFSM) model and the CSP language model is given below.

1. An interaction between two system components corresponds to a communication between two processes. For example, sending a frame from the sender to the medium is modeled by the matching communication pair:

   **Sender**:: [ ... Medium[frame] ... ]
   **Medium**:: [ ... Sender[frame] ... ]

2. The major states of a system component in EFSM specification are implicitly reflected by the control points (or locations) just before the input commands of the process modeling that component in CSP specification. Note that if there is a repetitive or alternative command with all its guards containing input commands, the control point at the beginning of that command is analogous to a major state from which one of several transitions may occur depending on which interaction is to happen. For example, the major states of the Sender are the control point at which it waits for a data block from User1 and the control point at which it may receive either an acknowledgement or a timeout signal.

3. A transition from a major state to a major state, and the action associated with it in EFSM specification is analogous to the execution flow from a control point to a control point, and to all the assignment and output commands executed between these two control points in CSP specification, respectively. For example, when the Sender gets a data block from User1, it will prepare a frame with a proper sequence number, send this frame, trigger the Timer and wait at the next control point.
4. Associating enabling predicates with transitions in EFSM specification is analogous to using boolean guards to guide the execution of the desired action and reaching the next control point in CSP specification. For example, upon receiving an acknowledgement, the sender will either retransmit the last frame or wait for another data block from User1, depending on whether the received acknowledgement is the expected one or not.

4. A SIMPLIFIED OSI TRANSPORT SERVICE

In this section we demonstrate how to use our technique to specify the OSI Transport Service. The complete OSI Transport Service (TS) Definition is described in [16]. We consider only a simplified version in which parameters of service primitives are ignored except the TS user-data parameter associated with the data transfer primitives, and in which only normal data transfer service is provided. We also assume that a Connection_request issued by a calling TS user (a TS user who initiates a Transport_connection establishment request) will always result in a Connection_indication to the called TS user (a TS user with whom a calling TS user wishes to establish a Transport_connection). Consequently, the simplified Transport Service consists of the following service primitives (see Table 1), each of which is invoked in one of the three phases - connection establishment, data transfer, and connection release.

The common operations in each phase can be illustrated by Figure 3 and explained below. Here, we assume that User1 is the calling TS user and User2 is the called TS user.

- The connection establishment phase as shown in Fig.3(a) is started by User1 who issues a Connection_request primitive to the underlying Service Provider. The Service Provider then issues a Connection_indication primitive to User2. If User2 is willing to accept this connection, a Connection_response primitive is given to the Service Provider who in turn issues a Connection_confirmation to User1.

- The data transfer phase as shown in Fig.3(b) is started on User2's side as soon as the Connection_response primitive is issued whereas it is started on User1's side only when the Connection_confirmation primitive occurs. In this phase, each user can issue a sequence of Normal Data_request primitives such that the TS user-data are conceptually queued in the Service Provider. The Service Provider, at the same time, dequeues this sequence by issuing Normal Data_indication primitives to the other side.

- The connection release phase as shown in Fig.3(c) is entered on each side when the user issues a Disconnect_request primitive or receives a Disconnect_indication either due to the inability of the Service Provider to maintain the connection or due to the initiation of a Disconnect_request by the other user.

An important feature of the Transport Service is that a connection release is permitted at any time regardless of the current phase. On each side, once a Disconnect_request or Disconnect_indication occurs, no further primitives may be performed. In order to deal with this situation, a modification to the original CSP language is made, as explained in the following paragraph.

In CSP, a repetitive command terminates when all its guards fail. A guard fails if either its boolean expression is false or the process named in its I/O command has terminated. As a result, the execution of a repetitive command may terminate due to the termination of other processes. This semantic property has been considered a powerful feature and is called distributed termination [17]. While this distributed termination feature is confined to the repetitive command, we shall extend it to the process level as follows. If an I/O command is in the outermost scope of a process (i.e. it is neither in the guard of a guarded command nor nested in any alternative or repetitive command) and if the process named in this command has already terminated when it is executed, then the I/O command along with the remaining part of the process are skipped and the whole process terminates immediately.
For example:

\[
\begin{align*}
A &::= \{ \text{B?go() } \rightarrow \text{skip} \\
& \quad \{ \text{C?go() } \rightarrow \text{skip} \}
\end{align*}
\]

\[
\begin{align*}
B &::= \text{A?go();}
& \vdots
\end{align*}
\]

\[
\begin{align*}
C &::= \text{A?go();}
& \vdots
\end{align*}
\]

Process A consists of only an alternative command in which one of the guarded commands is to be executed. If the first one is chosen, Process B will receive a go() signal and continue the execution of the remaining commands. Process C, on the other hand, terminates without executing any command since the input command A?go() is in the outermost scope and Process A has terminated. A similar statement can be given for the case when the second guarded command in Process A is chosen.

By this extension, a process may terminate due to the termination of another process and we shall make use of this feature to model the behavior of the connection release phase defined in the Transport Service.

Using the above extension, the CSP-based specification of the simplified Transport Service is given in Figure 4. It consists of 8 processes, each playing a different but integral role in its own right. Processes User1 and User2 model the behavior of the calling TS user and the called TS user respectively, while the other 6 processes taken together model the behavior of the Service Provider. As implied by the names of the processes, process Connection is to model the connection establishment phase, processes Dataqueue1 and Dataqueue2 are to model the two independent data flows in the data transfer phase, and the remaining three processes Disconnect1, Disconnect2, and DisconnectP are to model the connection release phase which can be initiated by the calling TS user, the called TS user, or the Service Provider, respectively.

To model the occurrence and the direction of propagation of a primitive, a matching I/O command pair is employed. From Figure 4, the reader may notice that all the communications are conducted between either User1 or User2 and the other 6 processes. This is not by chance but simply because all the service primitives only occur at the boundary (logical interface) between the service users and the Service Provider.

In order to highlight the control structures of those processes, all the declarations of variables are omitted. As they are quite clear from the context, the identifiers named "end" in processes User1 and User2 are boolean variables and their initial values are set to false. The parameter "data" is a variable whose value can be any positive integral number of octets. We also assume that processes Dataqueue1 and Dataqueue2 are associated with the abstract data objects Q1 and Q2 respectively. They are two ordinary queues and the elements are of the same type as variable
In the second aspect, the sequences of primitives in each phase are described by either one or a set of processes as explained below.

connection establishment phase:

Process Connection consisting of 4 sequential I/O commands specifies the sequence of primitives required for a successful connection establishment.

data transfer phase:

Processes Dataqueue1 and Dataqueue2 represent the behavior of two independent data flows in data transfer phase. For each process, a succession of data requests from the sending user result in the accumulation of data at one end of the queue and, at the same time, a succession of data indications to the receiving user result in the removal of data from the other end of the queue.

connection release phase:

Processes Disconnect1 and Disconnect2 describe the cases in which a Disconnect request is issued from a user leading to a Disconnect indication to the other user. Process
within a layer in response to the interactions
Another example is the case in which User1 issues
waiting for the Cres() signal from User2. Using
Dreq() signal. At this moment, process Connection
may refuse the connection request by sending a
and Disconnect2 to terminate instead of trying to
User2.

CSP specification, it means that once a user
process has terminated due to the sending of a
Dreq() or the receiving of a Dind(), all the other
processes wishing to communicate with it so as to
complete the remaining commands should terminate
instead of waiting indefinitely. This explains the
reason why we made an extension to the
original notion of distributed termination. For
instance, after receiving a Cind() signal, User2
may refuse the connection request by sending a
Dreq() signal. At this moment, process Connection
has already finished the first two commands and is
waiting for the Cres() signal from User2. Using
our extended notion of distributed termination, it
just ignores the remaining two commands and
terminates since User2 has already terminated.
Another example is the case in which User1 issues
a Dreq() to Disconnect1 and, at the same time,
DisconnectP issues a Dind() to User2. Again, the
termination of User2 will cause both Disconnect1
and Disconnect2 to terminate instead of trying to
communicate with the already terminated process
User2.

5. DISCUSSION AND COMPARISON

In this paper, we have used a CSP-based
specification technique to specify a protocol and
a service. The former is the Alternating Bit
Protocol and the latter is a simplified version of
the OSI Transport Service. It shows that the CSP
language model can be applied to both protocol and
service specifications in a uniform way.

As mentioned before, a protocol specification
describes the behavior of the protocol entities
within a layer in response to the interactions
with other peer entities or the two neighbor
layers. It emphasizes the interpretation of the
control information and the actions invoked by
service primitives. In view of this aspect, the
control structures in CSP, especially the guarded
command, can be employed to model these kinds of
behavior in a natural and concise manner. It has
long been criticized that a programming language
model usually has the shortcoming of specifying
unnecessary or implementation-oriented details.
While this is true for the Fortran or Pascal-like
languages, we think CSP is powerful enough to keep
the specifications at an appropriate abstract
level.

A service specification, representing a more
abstract view, defines the services provided by
the underlying service provider to the service
users. It emphasizes the allowed sequences of
service primitives at a boundary (logical
interface) between two layers. To deal with this
aspect, we view a CSP specification as a
"communication sequence generator". Therefore, it
is not surprising to find that, in the simplified
Transport Service specification, each process
spends almost all the time in communicating with
other processes. Also, unlike in the protocol
specification, only a small number of variables
are needed, except that some boolean variables
(e.g. ‘end’) are used to guide the control flow
and some data objects (e.g. Q1 and Q2) are used to
model the inherent property in the defined
service. In specifying a service, by virtue of the
rendezvous communication mechanism, the occurrence
of a service primitive can be modeled by a CSP
matching I/O command pair and all-com processes
communicate with each other in an arranged order.
The reader should note that while all the allowed
sequences of primitives are defined by the global
communication sequences, each individual process
in the specification represents a separate
functional unit and this modularity is not usually
obtainable by other approaches using sequencing
expressions.

We also mentioned at the beginning of this paper
that the approach based on the extended finite
state machine is considered to be suitable for
protocol specification whereas the approach based
on Calculus of Communicating Systems is considered
to be suitable for service specification. This
fact brings up a natural question: what are the
relations between the CSP language model and the
other two approaches mentioned above? In section
3, we have made a comparison between CSP language
model and the extended finite state machine model.
We conclude that the constructs of CSP can be used
to accomplish the same expressive power as the
extended finite state machine model does. On the
other hand, some comparisons between CSP and CCS
have been investigated in [4, 18]. Although the
relation between these two languages has not been
understood thoroughly, there is a strong evidence
showing that they have a close relation with each
other in both syntactic and semantic aspects.

As stated in [19], both the extended finite
state machine model and the temporal ordering
language model (which is based on CCS), have been
adopted as two standard formal description
techniques, and have been used to accomplish
interworking between these two techniques is
still underway. We hope this paper can give
some insight into the understanding of relations
between different formal specification techniques.

ACKNOWLEDGEMENT

The authors wish to thank Neelamegam
Soundararajan for his many helpful discussions and
comments while this work was in progress.
REFERENCES


