UNISPEX - A Unified Model for Protocol Specification and Verification

Son T. Vuong
Donald D. Cowan

CS-82-52

November, 1982

UNISPEX - A UNIFIED MODEL FOR PROTOCOL SPECIFICATION AND VERIFICATION

Son T. Vuong and Donald D. Cowan

Department of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA N2L 3G1

ABSTRACT

We develop a unified model, called UNISPEX, which combines the features of both state transition formalisms and programming language models for protocol specification. A preliminary version of UNISPEX is presented in this report. Its main features are illustrated in a specification example of a simple transport protocol and its verification capabilities are discussed, indicating the suitability of the model to be used in an integrated system for protocol design, validation, verification, and automatic implementation.

November 10, 1982

UNISPEX - A UNIFIED MODEL FOR PROTOCOL SPECIFICATION AND VERIFICATION

Son T. Vuong and Donald D. Cowan

Department of Computer Science
University of Waterloo
Waterloo, Ontario, CANADA N2L 3G1

I. INTRODUCTION

Much of the past research on formal techniques for protocol specification has focussed on either the state transition model [Bochmann78a, Merlin79a, West78a, Zafiropulo80a] or the programming language formalisms [Stenning76a, Brand78a, Hailpern80a, Schwabe81a]. Both approaches have been successfully used to model and verify different aspects of protocols. The state transition method has proven suitable for handling the control aspects of a protocol (such as call setup), while the programming language approach is found to be tractable for dealing with the semantic aspects of a protocol (such as window mechanism). Recently, much attention has been drawn to the development of unified formal description techniques (FDTs) which combine the advantages of both systems. Most of these formal description techniques are, however, oriented toward implementation [Bochmann81a, Blummer81a, Pokraka82a] and they often provide no features for protocol verification.

In this report, we propose a unified model, called UNISPEX (UNIfied SPEcification model), which support both protocol verification (including validation) and implementation. Our model is based on SPEX developed at UCLA [Schwabe81a] and on models described by Blummer and Tenney [Blummer81a] and by Bochmann [Bochmanna, Bochmann82a, Bochmann82b]. However, SPEX does not have an explicit state transition component, and so it does not lend itself naturally to protocol validation. Blummer's and Bochmann's models, on the other hand, contain an explicit state transition component, but their designs are directed toward automatic implementation of protocols, thus lacking support for verification. Our model combines both kinds of models and so, it provides emphasis on verification without sacrificing the essential features for automatic implementation. It is basically an extended finite state model, where each transition corresponds to a program segment with many features similar to SPEX such as *interface* and *channel* variables of abstract data type, which directly supports verification. In addition, the model also contains the specification of *service* primitives and *mapping* events which provides an abstract interface description necessary to characterize the local properties of a communication service provided by a protocol layer. SPEX does not have these features.

It is worthwhile to note that since the unified model contains an explicit state transition component, previous work on protocol validation via reachability analysis can be directly applied to validating protocols specified in this model.

Before discussing the UNISPEX model and its verification capabilities, it is essential to establish some common terminology on protocol specification. We present the model which was first introduced by Zimmermann and adopted for the Open System Interconnection [Zimmermann78a, Zimmermann80a].

According to this model, the communication architecture of a distributed system is structured as a hierarchy of different protocol layers, each one built upon its predecessor. The procedure of each layer is to provide certain services to the higher layers, shielding them from the details of how services are actually implemented. Thus, the specification of each protocol layer consists of service specifications, as viewed by its users and an internal protocol specification, as regarded by its designers.

Users view a protocol as a "black box" or machine, which provides a particular set of services in response to users' commands presented via the user-protocol interfaces (or service primitives), as illustrated in Figure 1. Examples of some basic interfaces (or service primitives) for a transport service are *Connect, Disconnect, Send,* and *Receive.* Naturally , the service primitives should not be executed in an arbitrary order and with arbitrary parameter values. The interface specification must reflect these constraints by defining the allowed sequences of operations directly, or by making use of a "state" of the service which may be changed as a result of some operations. For example, *Send* and *Receive* may only be executed after a successful *Connect.*

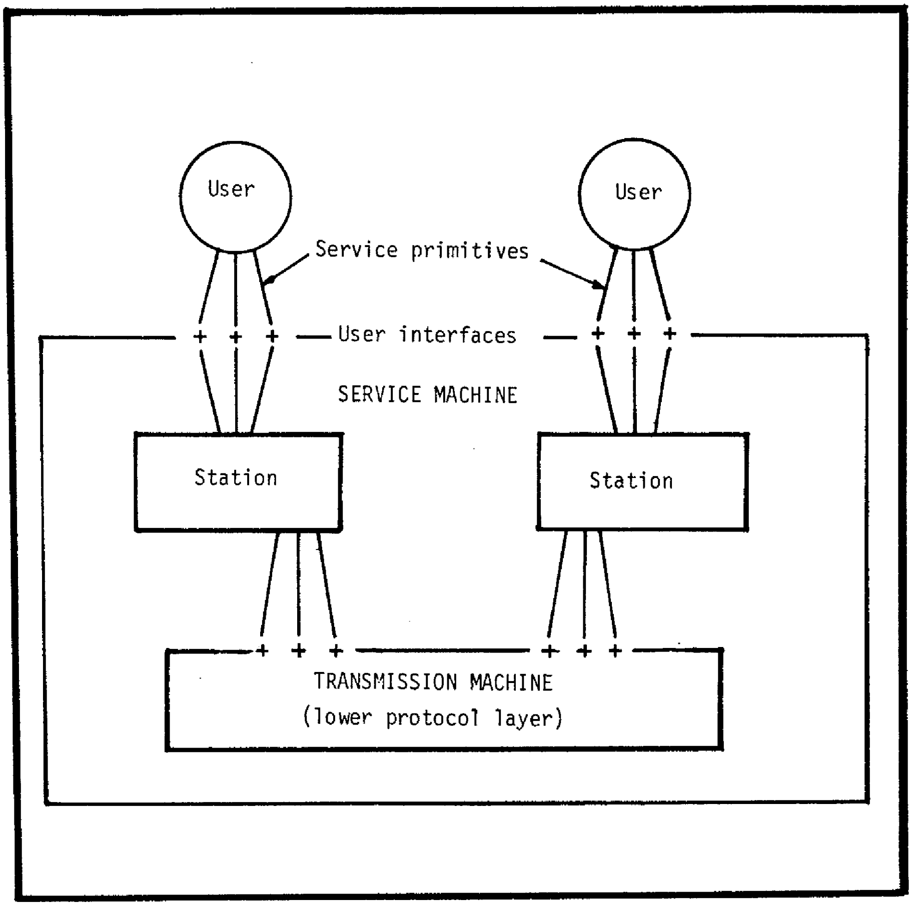

Figure 1   Structure of a protocol layer

The interface specification and the specification of the global properties of a protocol constitute the service specification. An example of a global property of a transport service is that the messages *Received* form an initial subsequence of the messages *Sent,* i.e. messages are transferred in order without corruption, loss or duplication.

Unlike users, protocol designers are concerned with the internal structure of a protocol layer. They view the protocol as a set of entities (processes, modules, stations or protocol machines) that are local to a user or users, and that communicate among themselves via the services of their lower layer (see Figure 1). An internal protocol specification, also referred to as a design specification, describes:

i) The rules governing the reaction of each entity in response to commands from its users, to messages from other stations, and to internally generated events such as timeouts;

ii) The data types and message formats used by the protocol.

The internal protocol specification can be considered as an "implementation" of the service specification, and as an abstract specification for implementers.

II. UNISPEX - A UNIFIED MODEL FOR PROTOCOL SPECIFICATION

We now present a formalism for specifying protocols. Following the hierarchical model of protocol systems a complete specification of a protocol (or protocol layer) must comprise both a service specification and an internal protocol specification of a set of interconnected *Protocol Machines.* The patterns of interactions between these protocol machines (or peer entities) constitute the layer behavior. In general, a layer may consist of several distinct types of protocol machines, each having its own behavior and possibly several *instances.* The term "protocol machine" is commonly used to refer to an instance of a protocol machine type.

A protocol (or protocol layer) can be completely specified in three parts: *Protocol Machine* part, *Topology* part, and *Properties* part. The Protocol Machine part describes the behavior of each type of protocol machine; the Topology part specifies the set of instances of each protocol machine type and the way the instances are interconnected; and the Properties part presents the desired properties of the interactions between the instances. Each of these specification parts is discussed in the following subsections.

II.1. Protocol Machine Part

A protocol machine has some (internal) *State Variables* and (some external) *Interface* and *Channel Variables;* these variables may be arbitrarily complex data types, which may be defined using algebraic specification methods [Liskov75a, Guttag78a, Guttag78b, Goguen78a]. A protocol machine reacts to a set of specified *Events* by changing some state variables and some external (i.e. interface and channel) variables for each event occurrence. Thus, the behavior of a protocol machine can be described by a finite state machine where the transitions are regarded as events represented by program segments. Each event (or transition) has an enabling predicate associated with it. When an enabling predicate is true, its associated event is enabled and can fire, activating the transition and the execution of the program segment.

We present the specification of a protocol machine type in three sections: Declaration section, Interfaces section and Events section.

II.1.1. Declaration Section

The Declaration section defines the constants, data types, (state, interface, and channel) variables, and initial values of the variables used in the specification of the protocol machine type.

State variables can only be accessed locally within each protocol machine. One of these variables, called "State", represents the major (control) part of the state of a protocol machine, i.e. the state of the finite state machine component. Interface and channel variables, on the other hand, can be accessed from the outside - this is how a protocol machine communicates with the outside world. Interface variables are used for communicating with other layers (via interfaces, as we shall see in a moment) while channel variables allows communications with other protocol machines in the same layer. Each interface or channel variable also has a *direction of flow* associated with it, indicating whether data in that variable flows *into* or *out of* a protocol machine; if no direction is specified, data in that variable flows in both directions.

The *Initial State* of a protocol machine can be specified by giving the values of any variables at system creation time. These include the initial value of "State", and usually empty or zero values for internal, interface, and channel variables.

II.1.2. Interfaces Section

The Interfaces section specifies the service primitive, (lower layer) primitives and mapping events for the protocol machine type. The service primitives characterize the interface to the higher layer while the primitives together with the mapping events constitute the interface to the lower layer. Activation of a service primitive causes some change in interface variable values which, in turn, triggers a protocol event. There may be an enabling predicate associated with a service primitive which must be true for its activation (just as in the case of an event - this is specified by a *when* statement). Similarly, changes in some channel variable values may activate execution of a primitive provided by the lower layer; and this happens via execution of a *mapping event*. A mapping event has an enabling predicate which must be true for its activation, and so, it is also specified by a *when* statement. Basically, we can view the specification of service primitives and of mapping events as sublayers above and below the current layer, respectively, as illustrated in Figure 2. The sublayer above the current layer provides a mapping between the services supported by this layer and the interface variables. Likewise, the sublayer below the current layer serves as a mapping between channel variables and the services provided by the lower layer. These mappings are essential because the interface and channel variables are only abstracted means though which a protocol machine communicates with the other layers. For verification purposes, interface and channel variables are adequate; but for implementation related objectives, these variables are too abstract, and so service primitive effects and mapping events (which maps channel variables to lower layer primitives) are necessary to provide sufficient refinements. These items were not provided in SPEX.

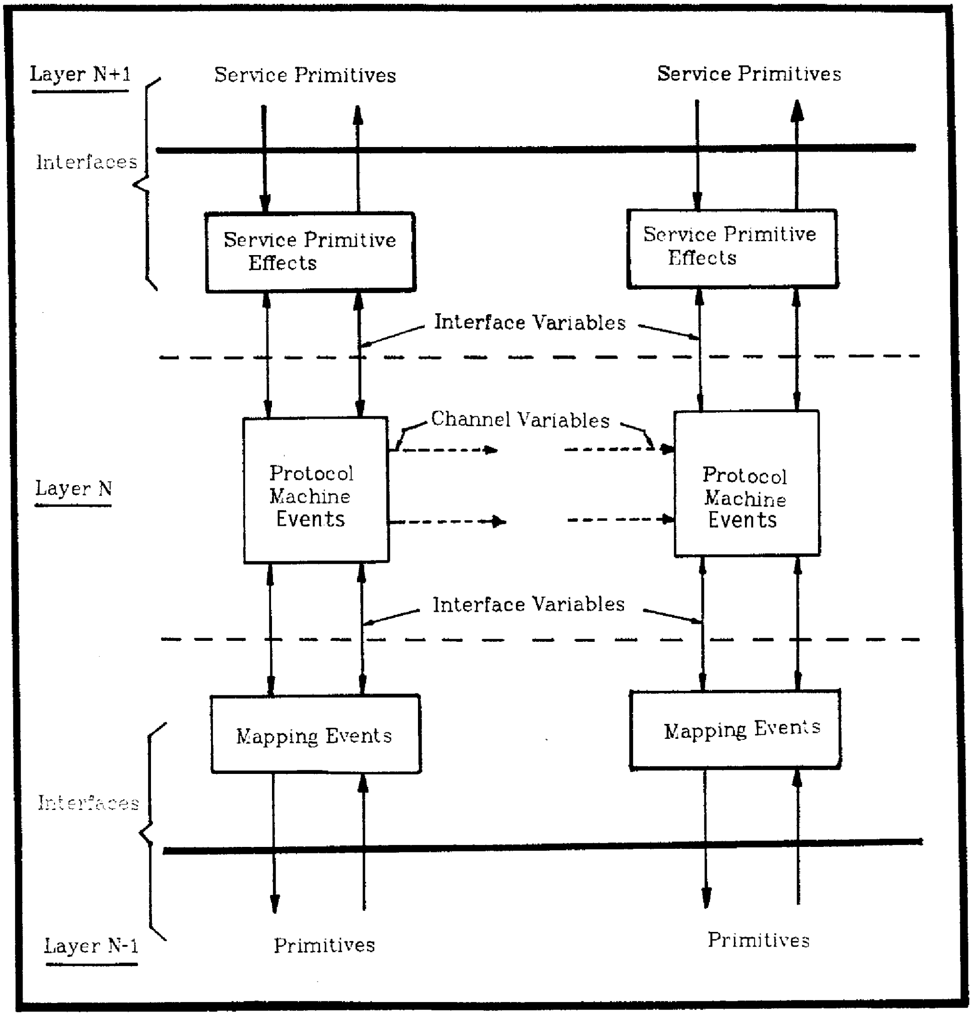

Figure 2   Components of a protocol layer

II.1.3. Events Section

The Events section describes the behavior of the protocol machine type by specifying all possible events which may occur in the course of interactions between this protocol machine and other protocol machine(s). Both the state transition and the program component of the protocol machine are specified explicitly within each event. Each event has, thus, an event name, an event code, origin and destination states, an enabling predicate, and an effect.

Event name is a unique name used to reference an event.

Event code, written as a plus or minus sign followed by a message name, indicates whether the event is a transmission or reception of a message. A minus sign "-" indicates message transmission and a plus sign "+ ", message reception - the same notation has been used for labeling transitions in the finite state machine model. Here, we have essentially restricted the scope of the events to that which permits the execution of a single message transmission or reception. A sequence of message transmissions and/or receptions is, therefore, considered as a sequence of events rather than a single event. This restriction allows basically a one-to-one mapping between an event in UNISPEX and a transition in the finite state machine model. In fact, the corresponding transition is labeled with the event code. Two special event codes, +OTHER and +ANY, are also used, and they require different mappings to state transitions. Furthermore, more than one alternate message name can be specified in an event code, meaning that one of these message types can be received (or transmitted) for the associated event, e.g. event code "+X,+Y" indicates reception of either message X or message Y.

Origin and *destination states* are the states (e.g. values of "State") prior to and after the execution of the event.

Enabling predicate is a boolean function of state, interface and channel variables, which must be true to enable its associated event. The origin state is also considered as part of an enabling predicate, i.e. "State" must be this origin state before the event is enabled. Enabled events may fire at any time.

The *effect* of an event comprises all changes in any variable values incurred by the occurrence of the event. These changes are considered to happen simultaneously, i.e. every event is atomic so that all changes for an event are computed from the old values of the variables *before* the event occurs. The order of these computations for an event is, thus, immaterial. The syntax of an event specification is given below, where a *when* statement is used to specify the enabling predicate and the effect of the associated event.

```
Events
[
    <event name> / <event code> / <origin> → <destination> :
    when (<enabling predicate>)
     [ <effect>]
    .
    .
]
```

Since interface and channel variables are externally visible, it is possible for a higher layer to change the value of some interface variable (via executing some service primitive) to enable an event in the protocol machine. Likewise, an event executed in one protocol machine may alter the value of some channel variable in another protocol machine, thereby enabling an event in the other protocol machine. This is effectively how protocol machines exchange data and synchronize their activities.

State variables, on the other hand, are not visible externally, and so, they can be regarded as *history* variables, which accumulate information about the computation.

II.2. Topology Part

The Topology part specifies the instances of each protocol machine type and how these instances are connected. The latter is achieved by allowing interface variables at each instance to be connected to interface variables at other instances. This means that connected interface variables are shared variables between the corresponding instances. This concept and the notation used are borrowed from SPEX.

Since channel variables are realized by interfaces with the lower layer, simultaneous accesses to shared (channel) variables by two events at different protocol machine are arbitrated by some lower-layer mechanism that nondeterministically chooses one event to be executed first. In general, events at different protocol machines may be enabled simultaneously, with their enabling predicates involving some shared variables. However, in this case, only one event will actually fire because events are atomic. Then all enabling predicates are recomputed to determine which events are enabled at that time.

Naturally, connected channel variables must be of the same data type and their direction of data flow must be consistent in the sense that and outgoing channel variable of one protocol machine must be connected to an incoming channel variable of another protocol machine. Furthermore, initial values of channel variables must be consistent across their interconnections.

II.3. Properties Part

The last part of the specification of a protocol layer is the *Properties* specification. A set of predicates are given to describe the layer's desired behavior. Normally, these predicates relate the values of the variables in different protocol machines, thus describing the global properties of the layer. Local properties can also be specified by giving predicates that involve only variables at one protocol machine. Furthermore, we can view local or global properties involving variable "State" as being attached to different states in a protocol machine.

The specified properties are actually *Asserted Invariants.* They hold for all possible system behaviors (i.e. all sequences of events in the system) and must be proven by the specifier in order to verify that the specification does indeed capture her/his intuitive understanding of the behavior of the system. Proofs for invariants are typically by induction over all possible sequences of events in the system.

In the next section the UNISPEXification (specification using UNISPEX) of a simple transport protocol is presented to illustrate the ideas of the specification model.

III.3. A TRANSPORT PROTOCOL AS AN EXAMPLE

The protocol example considered is based on a simple transport protocol described by Schwabe[Schwabe81a] which provides the service of reliable full-duplex data transfer with flow control between two stations. The lower layer, i.e. the network layer, is assumed to provide virtual circuit service, guaranteeing that messages are delivered in order between sender and receiver without error, loss, or duplication. So, the transport protocol is relatively simple - the subnetwork is in fact doing most of the work. X.25 basically supports such virtual circuit service.

The flow control in each direction of data transfer for a transport connection is provided via a simple credit (or window) mechanism, i.e. the receiver gives explicit credits to the sender each time it has enough buffer to receive new messages.

For the sake of simplicity, only a single transport connection is considered, assuming that the interactions (events) specified always refer to a particular transport (and network) connection not explicitly identified. The transport connection between two stations must be established before data transfer can actually occur. Either station can initiate the connection and when the data transfer session is complete either station can initiate the disconnection.

Figure 3 shows the UNISPEXification of the transport protocol.

The **Protocol Machine** statement indicates the beginning of the definition of a *Protocol Machine Type.* There is only one type of protocol machines in this protocol, namely *TransportStation.*

The **Needs** statement is a shorthand naming the data types which are used in the definition of this protocol machine type, but which are defined separately (in a library).

The **State Variables** at each protocol machine are:

State: represents the (control) state of the finite state machine component of the protocol machine;

Sent, Received: are ghost variables to record the histories of all messages sent to

```
Protocol Machine (TransportStation)

    Declaration

        Needs [Integer, Boolean, Side, Message, QueueOfMessage]

        Type Request = [CR, CC, data, DR, DC, null]

        Variables
        [
            State::
                State: [0..5],
                Sent, Received: QueueOfMessage;
            Interface::
                MaxRec: Integer,
                Accepted: Boolean,
                Credit: Integer,
                MsgToSend, MsgReceived: Message,
                Interface: [idle, conn, connpending, discpending],
                Cmd: [conn, disc, send, receive, null];
            Channel::
                CreditToRec, CreditToSend: Integer
                InReq, OutReq: Request,
                InChannel, OutChannel: QueueOfMessage;
        ]

        Initial State
        [
            State := 0 and
            Interface := idle and
            Cmd := null and
            InReq := OutReq := null and
            Accepted := false and
            Sent := Receive := empty and
            MsgToSend := MsgReceived := empty and
            CreditToSend := 0
        ]

    Interfaces

        Service Primitives
          [
            Connect (SizeOfBuf: Integer)
                When (Interface = idle)
                [
                    Interface := connpending;
                    Cmd := conn;
                    MaxRec := SizeOfBuf
                ]
            Disconnect()
                When (Interface != discpending)
                [
                    Interface := discpending;
                Cmd := disc
            ]
        Listen (SizeOfBuf: Integer)
            When (Accepted = false)
            [
                Accepted := true;
                MaxRec := SizeOfBuf
            ]
        DoNotListen()
            When (Accepted)
            [
                Accept := false
            ]
        Send (DataMsg: Message)
            When (Interface = conn and Cmd = null)
            [
                Cmd := send;
                MsgToSend := DataMsg
            ]
        Receive (DataMsg: Message)
            When (Interface = conn and Cmd = null)
            [
                Cmd := receive;
                When (Cmd = null)[DataMsg := MsgReceived]
            ]
    ]
```

Primitives

```
    [
        ToNet (Type: Request; CreditS, CreditR: Integer; Msg: Message);
        FromNet (Type: Request; CreditS, CreditR: Integer; Msg: Message)
    ]
```

Mapping Events

```
    [
        MsgSend
            When (OutReq != null or Credit != 0)
            [
                ToNet (OutReq, CreditToSend, Credit, MsgToSend);
                MsgToSend := empty;
                OutReq := null;
                Credit := 0
            ]
        MsgReceive
            When (InReq = null
                    and FromNet (Type, CreditS, CreditR, Msg))
            [
                InReq := Type;
                CreditToRec := CreditS;
                CreditToSend := CreditToSend + CreditR;
                MsgReceived := Msg
            ]
            ]
    Events
      [
        ConnReq /-CR/ S0 → S1:
                When (Cmd = conn and OutReq = null)
                [
                   Cmd := null;
                   OutReq := CR;
                   CreditToRec := MaxRec;
                   Credit := MaxRec
                ]
        ConnRemConf /+CC, +CR/ S1 → S3:
                When (InReq = (CC or CR))
                [
                   Interface := conn;
                   InReq := null
                ]
        ConnRemReq /+CR/ S0 → S2:
                When (InReq = CR and Accepted)
                [
                   Interface := connpending;
                   InReq := null
                ]
        ConnConf /-CC/ S2 → S3:
                When (OutReq = null)
                [
                   CreditToRec := MaxRec;
                   Credit := MaxRec;
                   Interface := conn;
                   OutReq := CC
                ]
        DataSend /-data/ S3 → S3:
                When (Cmd = send and CreditToSend > 0
                                        and OutRequest = null)
                [
                   Sent := Add(Sent, MsgToSend);
                   OutChannel := Add(Outcchannel,MsgToSend);
                   OutReq := data;
                   CreditToSend := CreditToSend -1;
                   Cmd := null
                ]
        DataRec /+data/ S3 → S3:
                When (Cmd = receive and InReq = data)
                [
                   Received := Add(Received, MsgReceived);
                   InChannel := Remove(Inchannel);
                   MsgReceived := Front(InChannel);
                   CreditToRec := CreditToRec + 1;
                   Credit := Credit + 1;
                   Cmd := null;
                   InReq := null
```

```
              ]
          DiscReq /-DR/ (S0,S1,S2,S3) → S4:
                  When (Cmd = disc)
                  [
                     Cmd := null;
                     OutReq := DR
                  ]
          DiscRemConf /+DC, +DR/ S4 → S0:
                  When (InReq = (DC or DR))
                  [
                     Interface := idle;
                     InReq := null
                  ]
          DiscRemReq /+DR/ (S0,S1,S2,S3) → S5:
                  When (InReq = DR and Accepted)
                  [
                     Interface := dispending;
                     InReq := null
                  ]
          DiscConf /-DC/ S5 → S0:
                  When (OutReq = null)
                  [
                     Interface := idle;
                     OutReq := DC
                  ]
      ]
```

Topology

```
       [
         Instances::
              Station: Array(side) of TransportStation;
         Connections::
              Station(i).OutReq <--> Station(OppositeSide(i)).InReq;
              Station(i).CreditToSend <--> Station(OppositeSide(i)).CreditToSend;
              Station(i).OutChannel <--> Station(OppositeSide(i)).InChannel
       ]
```

Properties

```
       [
           CreditToSend ≥ 0 and CreditToReceive ≥ 0;
           Sent = Append(Received,InChannel);
           Length(InChannel) ≤ MaxRec
       ]
```

Figure 3 A simple transport protocol in UNISPEX

or received from the other station, respectively, by this station - they are used for conveniently stating the properties of the protocol.

The **Interface Variables** at each protocol machine are:

Accepted: indicates whether this station is willing to accept a connection or disconnection request from another station;

MaxRec: is an integer indicating the maximum number of messages that this station can receive from the remote station before delivering to its user (i.e. the window or the maximum number of messages that the remote station can send before receiving an acknowledgement or credit) - it is initialized during connection establishment to the size of the of the receive buffer, as given by the user;

Credit: is an integer indicating the number of additional credits or extra messages that this station can receive - it is incremented by one each time a message is delivered to the user and is reset to zero after its value is conveyed to the remote side;

MsgReceived, MsgToSend: are message received and message to be sent, respectively, by this protocol machine;

Interface: indicates the state of the interface (connection) between the protocol machine and its user, which can be one of {idle, conn, connpending, discpending};

Cmd: is a buffer used by the user to indicate a connect, disconnect, send, or receive request to the protocol machine, thus triggering an event in the protocol machine - it can be one of {conn, disc, send, receive, null}.

The **Channel Variables** known for this protocol machine are:

CreditToRec, CreditToSend: are integers indicating the number of messages that this protocol machine is capable of receiving and transmitting, respectively;

InReq, OutReq: are buffers used to indicate the reception or transmission, respectively, of a (control or data) message by this side to the other side (protocol machine) - they can be one of {CR, CC, data, DR, DC, null}, where

CR = Connect Request, CC = Connect Confirmation, DR = Disconnect Request, and DC = Disconnect Confirmation;

InChannel, OutChannel: are queues of messages that are to be received and that have been sent, respectively, by this station, representing the full-duplex channel connecting this station to the other station.

The protocol provides to the higher layer a set of **Services Primitives,** which can be invoked by the user in the form of procedure calls. Execution of a service primitive causes some change in interface variables, which triggers a protocol machine event.

The *Connect* primitive tries to establish a transport connection. In our simple model, it has a parameter, *SizeOfBuf,* which gives the size of the receive buffer, i.e. the maximum number of messages the user is capable of receiving. (In more general cases, this call would have two additional parameters, a local and a remote transport address, and each transport address may participate in more than one transport connections.)

The *Disconnect* primitive is to terminate a transport connection, leading the interface (and the protocol machine) back to the idle state.

The *Listen* and *DoNotListen* primitives announce the user's willingness and unwillingness, respectively, to accept a connection request from the remote end.

Send primitive is invoked by the user to transmit a data message on the transport connection. This primitive and the *Received* primitive can only be activated when the state of the interface is connected, i.e. the station is in the data transfer state.

The *Receive* primitive indicates the user's desire to accept a data message on the established transport connection.

The interface of a transport station to the network layer is via two primitives *ToNet* and *FromNet,* each having four parameters: the packet type, chosen from the set {CR, CC, data, DR, DC}; the number of sending credits remained; the number of additional receiving credits; and the message data, which is "empty" when the packet is not a data packet.

The **Mapping Events** *MsgSend* and *MsgReceive* provide a mapping between the channel variables and the network primitives. *MsgSend* maps channel variable *OutReq* and interface variable *MsgToSend* to primitive *ToNet,* and *MsgReceive* maps primitive *FromNet* to channel variable *InReq* and interface variable *MsgReceived.*

The **Events** known at each protocol machine are:

ConnReq

which is enabled by a connect request from the user. Its effect is to activate the transmission of a Connect Request (CR) packet.

ConnRemConf

which is enabled by the reception of a Connect Confirmation (CC) or Connect Request (CR) from the remote user. Its effect is to lead the station to the *DataTransfer* state (state 3).

ConnRemReq

which is enabled when a connect request is received from the remote user and the local user is willing to accept the connect request. Its effect is to lead the station the *RemoteReq* state (state 2).

ConnConf

which is enabled when the station is in the *RemoteReq* state. Its effect is to transmit a connection confirmation (CC).

DataSend

which is enabled when the connection has been established and there are a send request from the user and enough credits for the message to be sent. Its effect is to transmit a data message to the remote side and decrement the number of credits for sending data messages.

DataRec

which is enabled when the connection has been established and there are a receive request from the user and a data message available at the station to be delivered to the user. Its effect is to deliver the received message to the user and allows an extra receiving credit.

DiscReq

which is enabled whenever the user issues a disconnect request. Its effect is to transmit a Disconnect Request (DR) packet.

DiscRemConf

which is enabled by the reception of a Disconnect Confirmation (DC) or Disconnect Request (DR) from the remote user. Its effect is to lead the station back to the *Idle* state (state 0).

DiscRemReq

which is enabled when a disconnect request is received from the remote user and the local user is willing to accept the disconnect request. Its effect is to lead the station the *RemoteDisc* state (state 5).

DiscConf

which is enabled when the station is in the *RemoteDisc* state. Its effect is to transmit a Disconnect Confirmation (DC) packet.

The **Topology** part indicates that the protocol layer consists of two stations (data type *Side* has two values), each being an instance of the TransportStation protocol machine type. Furthermore, the channel variables *OutReq, Credit-ToSend,* and *OutChannel* at each side are connected to the channel variables *InReq, CreditToRec,* and *In*Channel at the opposite side, respectively.

The **Properties** part asserts three invariant properties. The first property simply says that the number of sending credits and the number of receiving credits are nonnegative. The second property states that all messages sent by this side to the opposite side have been received in order except for possibly those last messages which are still in transit in the channel. The last property asserts that the number of messages in transit in a channel does not exceed the window size (i.e. the maximum number of credits allowed). The two last properties can be combined to state that the messages sent by each side do not lag the messages received by the opposite side by more than the window size.

IV. VERIFICATION OF PROTOCOLS MODELED IN UNISPEX

Protocol verification consists of validation against general syntactic properties and proof of correctness of the semantic properties. In our previous work we have dealt with validation of protocols via reachability analysis, and decomposition [Vuong81a, Vuong82a, Vuong82b, Vuong83a]. Our approach as well as many other validation approaches [Zafiropulo80a] require that protocols be specified in terms of finite state diagrams. The UNISPEXification of a protocol contains an explicit state transition component which can be extracted from the specification and translated easily into the state diagram form. The state diagram representation gives an overview (the control structure) of protocols and allows the applications of those validation techniques to verify the general protocol properties. We now discuss the systematic translation of linear UNISPEXifications into the familiar state diagram representation.

IV.1. Translation into State Diagrams

Via <origin>, <destination>, and <event code>, an UNISPEX event can be directly translated into a transition (arrow or arc) of the corresponding state diagram. The origin and destination states of the event are the origin and destination states of the transition, and the event code is the label of the transition, indicating a message reception or transmission. When <origin> of an event is a list of states rather than a single state, this event is mapped into several transitions (or arcs) with the same label, namely <event code>. Each of these transitions is drawn from one of those origin states to the destination state.

Figure 4 shows the state diagram for the UNSPEXification of the transport protocol in Figure 3. Notice that event *DiscReq* is translated into four arcs with label -DR, drawn from states 0,1,2,3 to state 4. Similarly, event *DiscRemReq* is mapped into four arcs with label +DR, drawn also from states 0,1,2,3 to state 5. A list of states for <origin> is a convenient notation in UNISPEX. It allows hierarchical state diagram specification such as the one used in the CCITT X.25 and X.75 specification. So, two separate hierarchical state diagrams can be drawn for the transport protocol: one for the connection termination phase, containing the last 4 events; and the other for the connection establishment phase, containing the first 6 events. Furthermore, the nested structure can be easily recognized through the hierarchical specification via <origin> as list of states. Thus, protocol decomposition based on nested structured can be directly applied to a UNISPEXification to yield two (internal and external) components, as shown in Figure 5 for the transport protocol example.

Other extensions with special values being assigned to <origin>, <destination>, and <event code> also require special treatment. If <origin> is ANY, the associated event will be translated into several transitions, one for each state. IF <destination> is SAME, the destination state will be the same as the origin state. Thus, {ANY $\rightarrow$ SAME} will be translated into several self-loop transitions, one for each state of the protocol machine.

It is worthwhile to note that errors detected in the validation of a state diagram specification may not necessarily all be errors in the original UNISPEX version. In fact, since context information of an event is ignored during the translation to a state diagram, an event which may be disabled under certain conditions, may be allowed to occur in the state diagram representation. Likewise, if we provide the event priority capability in UNISPEX, i.e. each event has a priority attribute so that simultaneously enabled events can be arbitrated according to their priority, then it is possible that *one* of several events" in UNISPEX gets translated inaccurately to *any* of several transitions" in the state diagram representation.

IV.2. Proof of Correctness

Once a protocol (layer) has been designed and specified, the protocol designer may seek a proof of correctness in order to increase her/his confidence that the specification does indeed capture her/his understanding of the protocol. This proof is done by induction on the sequences of events of the entire system and can be carried out in a top-down manner so that a global property is proved by assuming that the local properties at each protocol machine hold. Each local property, in turn, can be understood from the assertions on each service primitive, which may be shown from the assertions associated with each protocol event. At the lowest level, given an entry assertion for an event, its exit

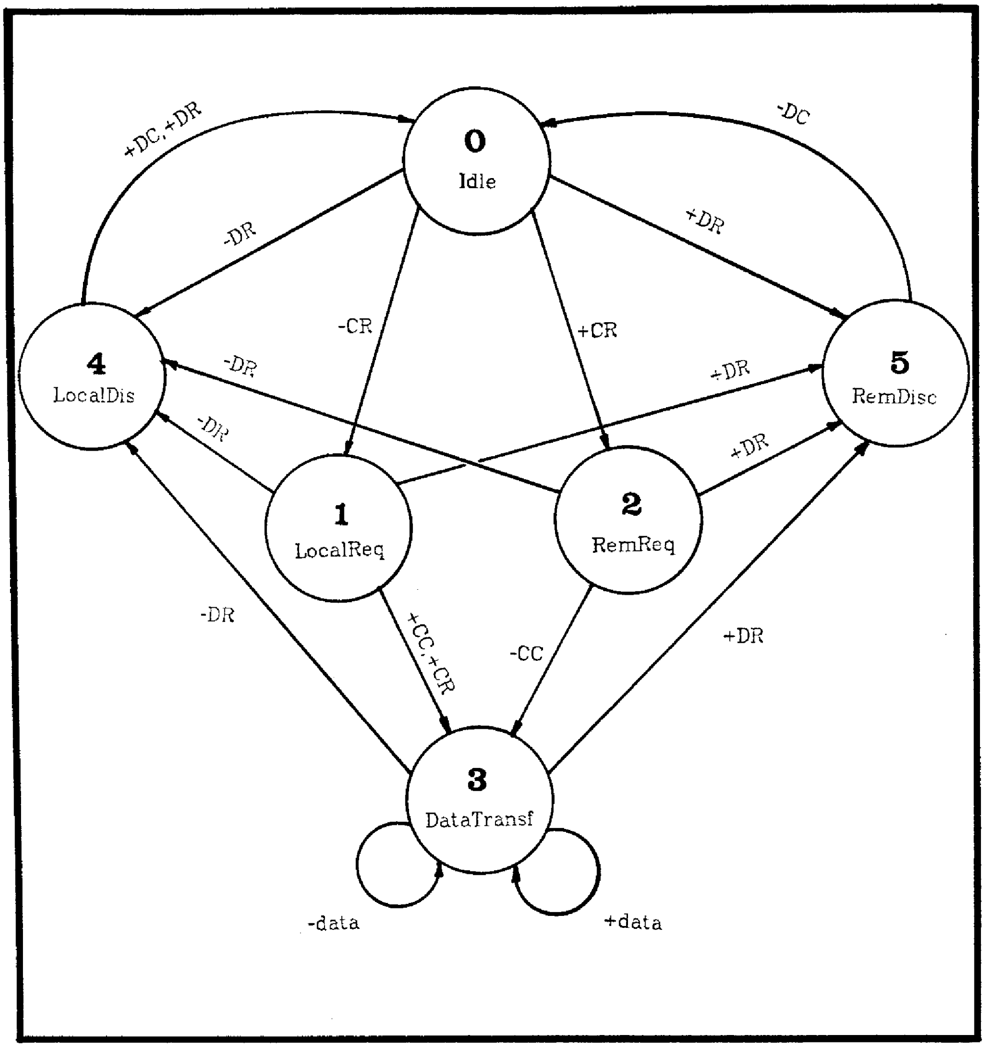

Figure 4   Finite state diagram for the transport protocol in figure VI.2

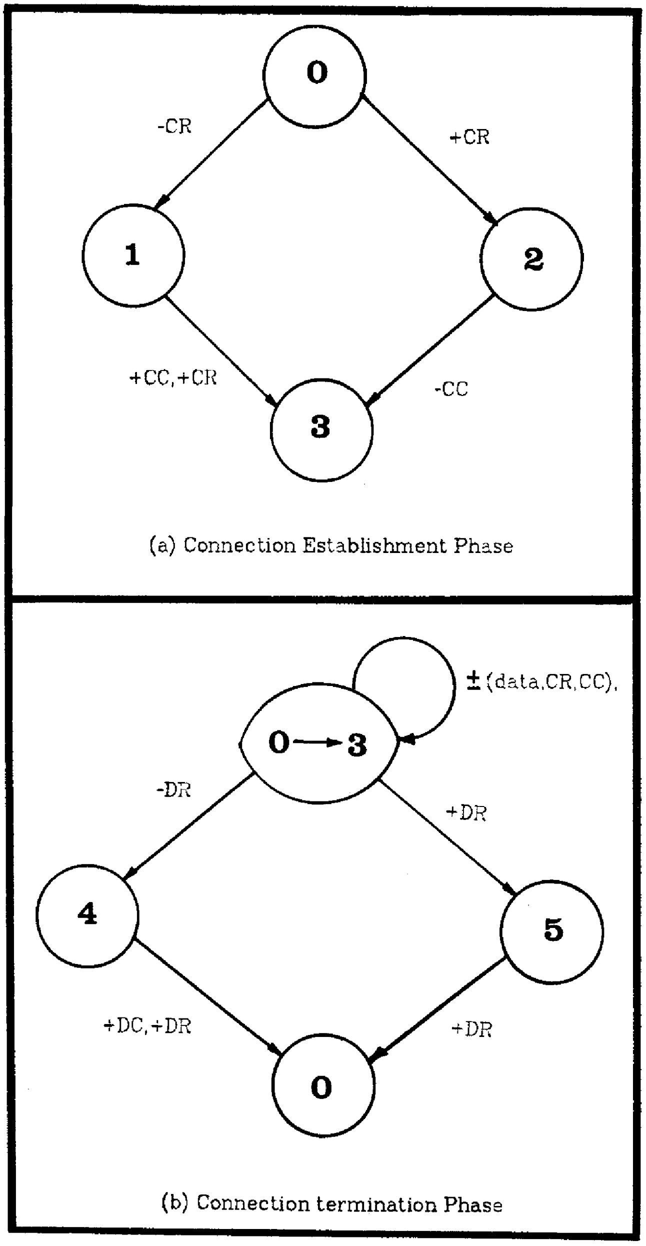
0
-CR
+CR
1
2
+CC,+CR
-CC
3
(a) Connection Establishment Phase
±(data,CR,CC),
0⟶3
-DR
+DR
4
5
+DC,+DR
+DR
0
(b) Connection termination Phase

Figure 5   Decomposed finite state diagrams for the transport protocol

assertion (after the event is executed) can be proven from the entry assertion, the enabling predicate, and the effect of the event. This can be done by hand simulation or reliably by the machine. Several systems have been develop to aid the verification process. Usually, a verification condition generator is used to generate theorems which must be proved in order to verify the specification. The proofs often require ingenious assertions introduced at intermediate steps so that an interactive theorem prover based on deduction and extensions for subgoaling, matching, and rewriting would be useful to aid proving theorem.

Schwabe [Schwabe81b] gives the rules to translate any given protocol SPEXification into an algebraic data type specification which can be verified using the rewriting rule capability of the AFFIRM system. Similar rules can be defined for UNISPEX to allow the verification of a protocol UNISPEXification to be carried out using existing automated systems such as AFFIRM [Thompson81a] INA JO [Eggert80a], GYPSY [Good81a], and SPECIAL [Silverberg80a]. A brief comparative description of these systems can be found in [Cheheyl81a, Sidhu81a].

In addition to protocol verification, i.e. verification of global properties of protocols that we have discussed, another kind of desirable verification is service verification (or satisfaction), which involves proving that the *protocol* specification and the *service* specification behave the same way when observed through their interface variables. So, a user may design her/his system using the service specification with the confidence that when the service is realized by the protocol in the actual system, there will be no undesired or unexpected behavior.

The third kind of verification is to show that a certain program satisfies a protocol specification. This kind of verification is essentially the same as service verification. The difference is that a different lower level language is used for specifying the program in contrast to the same language being used for both service and protocol specifications.

V. SUMMARY

Aiming at an integrated system for protocol specification, synthesis, verification, and automatic implementation, we have introduced a unified model which uses both finite state transition diagrams and programs for specifying protocols. We have presented the features of the specification model and illustrated these features in a specification example of a transport protocol. Validation and verification issues were also discussed.

Although further experience with modeling real-life protocols and exposure to different existing computer tools for protocol verification are needed in order to develop such an integrated system and to define completely the features of the model, we believe that most features of the unified model presented herein, are of general validity and applicability.

REFERENCES

[Blummer81a]

Blummer, T. P. and Tenney, R. L., "An Automated Formal Specification Technique For Protocols," *Protocol Testing - Towards Proof?, INAG/NPL Workshop Proceedings,* (May 1981).

[Bochmann82a]

Bochmann, G. V., Cerny, E., Gagne, M., Jard, C., Leveille, A., Lacaille, C., Maksud, M., Raghunathan, K. S., and Sarikaya, B., *Some Experience with the Use of Formal Specifications,* Pubication #433, Dept. d'Informatique et de recherche operationnele, U. de Montreal, Montreal, P. Q. (1982)

[Bochmann78a]

Bochmann, G. V., "Finite State Description of Communication Protocols," *Computer Networks,* pp. 361-372 (October 1978).

[Bochmann81a]

Bochmann, G. V., "The Use of Formal Description Techniques for OSI Protocols," *Proc. National Telecom. Conf.,* pp. F8.6.1-6 (December 1981).

[Bochmann82a]

Bochmann, G. V., "Examples of Transport Protocol Specifications," International Organization for Standardization, Enschede (April 1982).

[Bochmann82b]

Bochmann, G. V., "Formalized Specification and Analysis of a Virtual File System," Hahn-Meitner-Institut HMI-B 367, Berlin (February 1982).

[Brand78a]

Brand, D. and Joyner, W. H., "Verification of Protocols Using Symbolic Execution," *Computer Networks* **2** pp. 352-360 (October 1978).

[Cheheyl81a]

Cheheyl, M. H., Gasser, G., Huff, G. A., and Millen, J. K., "Verifying Security," *ACM Computing Surveys* **13**(3) pp. 279-339 (September 1981).

[Eggert80a]

Eggert, P. R., "Overview of the Ina Jo Specification Language," Technical Report SP-4082, System Development Corporation, Santa Monica, California (October 1980).

[Goguen78a]

Goguen, J. A., Thatcher, J. W., and Wagner, E. G., *An Initial Algebra Approach to the Specification, Correctness, and Implementation of abstract data types,* Prentice Hall (1978).

[Good81a]

Good, D. I. and Devito, B. L., *Using the Gypsy Methology.* October 1981.

[Guttag78a]

Guttag, J. V., Horowitz, E., and Musser, D. R., "Abstract Data Types and Software Validation," *Communications of the ACM 21,* pp. 1048-1064 (December 1978).

[Guttag78b]

Guttag, J. V. and Horning, J. J., "The Algebraic Specification of Abstract Data Types," *Acta Informatica,* (10) pp. 27-52 (1978).

[Hailpern80a]

Hailpern, B. and Owicki, S., "Verifying Network Protocols Using Temporal Logic," *NBS Trends and Applications Symposium,* pp. 28-35 (May 29, 1980).

[Liskov75a]

Liskov, B. and Zilles, S., "Specification Techniques for Data Abstractions," *IEEE Trans. Software Engineering,* (March 1975).

[Merlin79a]

Merlin, P. M., "Specification and Validation of Protocols," *IEEE Transactions on Communications* **COM-27**(11) pp. 1671-1680 (November 1979).

[Pokraka82a]

Pokraka, E.N. and Vuong, S.T., "A Hybrid Model for Protocol Specification and Verification," *Proc. Conf. of the Canadian Information Processing Society (CIPS),* (May 1982).

[Schwabe81b]

Schwabe, D., "Formal Specification and Verification of a Connection Establishment Protocol," *Proceedings, Seventh Data Communications Symp.,* p. Mexico City (October 1981).

[Schwabe81a]

Schwabe, D., "Formal Techniques for the Specification and Verification of Protocols," Ph. D. Thesis, University of California, Los Angeles (1981).

[Sidhu81a]

Sidhu, D. P., "Toward Constructing Verifiable Communication Protocols," *In Protocol Testing - Toward Proof?, INWG/NPL, Workshop Proceedings,* pp. 75-141 (May 1981).

[Silverberg80a]

Silverberg, B. A., "An Overview of the SRI Hierarchical Development Methodology," Technical Report CSL-116, SRI Project 1015, Contract N0039-79-C-0, SRI International, California (July 1980).

[Stenning76a]

Stenning, N. V., "A Data Transfer Protocol," *Computer Networks,* (1) pp. 99-110 (1976).

[Thompson81a]

Thompson, D. H., Gerhart, S. L., Erickson, R. W., Lee, S., and Bates, R. L., *The Affirm Reference Library,* Information Science Institute, University of Southern California (1981).

[Vuong81a]

Vuong, S. T. and Cowan, D. D., "Automated Protocol Validation via Resynthesis: The CCITT X.75 Packet Level Recommendation as an Example," Research Report CS 80-39, Computer Science Department, University of Waterloo, Canada (revised) (May 1981).

[Vuong82a]

Vuong, S. T. and Cowan, D. D., "A Decomposition Method for the Validation of Structured Protocols," *Proc. INFOCOM Conference,* (April 1982).

[Vuong82b]

Vuong, S. T. and Cowan, D. D., "Protocol Validation via Reachability Analysis," *Proc. COMPCON Fall 82,* (September 1982).

[Vuong83a]

Vuong, S.T. and Cowan, D.D., "Reachability Analysis of Protocols with FIFO Channels," *Proc. ACM SIGCOMM '83 Symp. on Communications Architecture and Protocols,* (March 1983).

[West78a]

West, C. H., "General Technique for Communication Protocol Validation," IBM J. Research Development (July 1978).

[Zafiropulo80a]

Zafiropulo, P., West, C.H., Cowan, D.D., and Brand, D., "Toward Analyzing and Synthesizing Protocols," *IEEE Trans Communications* **COM-28**(4) pp. 651-660 (April 1980).

[Good78a]

Good, D. I., and Cohen, R. M., "Verifiable Communications Processing in GYPSY," *IEEE Trans Communications,* (1978).

[Zimmermann78a]

Zimmermann, H. and Naffah, N., "On Open System Architecture," *Proc. ICCC,* pp. 669-674 (September 1978).

[Zimmermann80a]

Zimmermann, H., "OSI Reference Model - The OSI Model of Architecture for Open Systems Interconnection," *IEEE Trans. on Commununications* **COM-28**(4) pp. 425-432 (April 1980).