SERVICE SPECIFICATION AND PROTOCOL CONSTRUCTION FOR THE TRANSPORT LAYER[1]

Sandra L. Murphy and A. Udaya Shankar
Department of Computer Science
University of Maryland
College Park, Maryland 20742

ABSTRACT

In a computer network, the transport layer uses the service offered by the network layer and in turn offers its users the transport service of reliable connection management and data transfer. We provide a formal specification of the transport service in terms of an event-driven system and safety and progress properties. We construct three verified transport protocols that offer the transport service. The first transport protocol assumes a perfect network service, the second assumes loss-only network service, and the third assumes loss, reordering and duplication network service.

Our transport service specifications are very realistic. Each user can be closed, listening, active opening, passive opening, open, or closing. A local incarnation number uniquely identifies every active opening and listening duration. Users can issue requests for connection, listening, closing, data send, etc. The transport layer issues indications for successful or unsuccessful connection, closing, data reception, etc. A connection is established only if one user requested the connection and the other was listening, or both requested the connection. A user receives data only from the appropriate incarnation of the distant user, and receives it insequence, without loss or duplication. Progress properties ensure that every outstanding user request is eventually responded to by an appropriate transport indication.

Our protocols are constructed by stepwise refinement of the transport service. The construction method automatically generates a verification that the protocols satisfy the transport service. One distinctive feature of our protocol construction is that the events and verification of the data transfer function is directly obtained from any one of the numerous verified single-incarnation data transfer protocols already presented in the literature.

[1]Work supported by National Science Foundation Grant No. ECS 85-02113.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/ or specific permission.

© 1988 ACM 0-89791-279-9/88/008/0088 $1.50

1. INTRODUCTION

In a computer network, the communication protocols are organized into hierarchical layers. The *service specification* of a protocol layer defines the desired behavior of the interface between that protocol layer and the protocol layer above it. A formal service specification is necessary for guiding software engineers in building applications, for interfacing protocols of the same layer, and for verification of protocol standards and implementations. The service specification must not only be complete, but it must be compact and convenient to use [VISS].

The transport layer lies between the network layer and a layer of users, as illustrated in Figure 1. It uses the services offered by the network layer and in turn offers the transport services of reliable connection management and data transfer to the users.

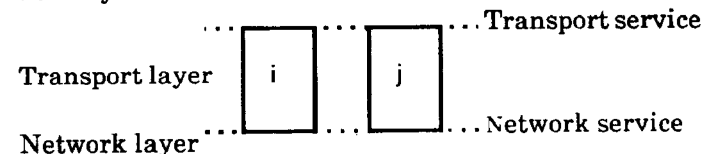

Figure 1: Transport layer interfaces

Data transfer is the function of immediate interest to the users. Connection management enters the picture because in practice, a user desires to exchange data with other users only periodically. Between such periods, the user does not want to be aware of the other users on the network. We refer to such a user state as *closed.* Each time the user leaves the closed state, it is said to acquire a *new incarnation.* Typically, a closed user is not allocated communication resources such as port numbers, buffers, channels, etc. These resources are freshly acquired each time the user becomes active. This is significant for efficiency and crash recovery.

When a user wants a connection established with a distant user, it is the responsibility of the connection management function to determine whether the distant user is willing to accept a connection, and if so to establish message communication paths between the users that are free of messages from past incarnations. The data transfer function builds upon this *connection management service* to offer the *data transfer service*,

which is that data of the current incarnations is delivered insequence, without any loss, duplication, or reordering.

Service specification approaches

The first step in specifying the transport service is to define the events at the user interface, i.e., the user requests and protocol indications. A sequence of such interface event occurrences represents an instance of interaction between the users and the protocol. The second step in specifying the transport service is to define the set of interface event sequences allowed at the interface. A simple connection management service can be specified by the events $ConnReq_i$, $ConnInd_i$, and $RejInd_i$ at site i, the events Acc_j and Rej_j at site j, and the allowed sequences $< ConnReq_i$, Acc_j, $ConnInd_i >$ and $< ConnReq_i$, Rej_j, $ConnRej_i >$.

In general, an interface event sequence is of infinite length, because services are usually nonterminating. Because of nondeterministic choice, there are usually an infinite number of event sequences allowed. Thus, service specification by enumeration as in the above example is not possible. Furthermore, events usually have parameters (e.g., incarnation numbers, data values, etc.), and the event sequences must satisfy constraints regarding these parameter values.

Several approaches have been proposed to specify the set of event sequences allowed. One approach is to use grammars from the theory of formal languages, such as attribute grammars, context-free grammars, regular expressions, etc. [HAAS]. A second approach is to specify a finite state machine or Petri Net that generates the set of allowed event sequences. Such specifications have the advantage that they can be efficiently analyzed by automated means, but suffer from the disadvantage of not being powerful enough to handle realistic services without being too cumbersome, i.e., the infamous state space explosion. Typically, with such specifications, one proves only general properties such as boundedness (the number of messages in the channel is finite), absence of unspecified receptions, and absence of deadlock. A third approach is to use extended state machines [BILL, BRINK]. These models do not suffer from the disadvantage of the second approach, but neither do they have its advantage. Also, state machines by themselves cannot specify progress properties that require fairness requirements, such as a message that is sent repeatedly into a channel is eventually delivered.

Our approach is to use *safety* and *progress properties*, in conjunction with an extended state machine specified by state variables and events. The safety properties in this report will be restricted to invariant properties, which are statements of the form A *is invariant* where A is a predicate in the state variables. A is invariant means that at every instant the values of the state variables satisfy A. Progress properties are statements about values eventually achieved by state variables. We shall consider progress properties of the form A *leads-to* B, where A and B are predicates in the system state variables. A *leads-to* B means that if the system state satisfies A, then the system will eventually enter a state where B holds. This is a small fragment of the more general linear time temporal logic [MANN].

Our transport service specification

We provide formal specifications of transport services between every pair of users, i and j (see Figure 1). Our transport service specifications are symmetric. Each user can be closed, listening, active or passive opening, open, or closing. A local incarnation number uniquely identifies every active opening and listening duration. Users can issue connection requests, listen requests, end listen requests, close requests, and data send requests. The transport layer can issue connection indication, reject indication (unsuccessful completion of a connection request), close indication, and data receive indication.

The following safety properties are specified for the service: A connection is established only if one user requested the connection and the other was listening, or both requested the connection (balanced opening). A user receives data only from the current incarnation of the distant user, and receives it insequence, without loss or duplication. The following progress properties are specified for the service: A connection request by a user will eventually result in a connection, provided that the remote user is listening or is itself actively opening. The states of active opening, passive opening, and closing are transient. Data sent by an open user is eventually received, unless either of the users request closing of the connection.

Our connection termination service is similar to that offered by ISO TP, rather than that offered by TCP where each entity independently closes its outgoing data connection. This choice was made only for the sake of brevity. Other than this, our service specification is equivalent to (what we believe is) the intended service of the Internet TCP [DoD], and subsumes the service of ISO TP/Class 4 [ISO1] (see below).

We believe that an approach based on safety and progress properties provides the correct level of abstraction for real-life protocol services, such as the transport service. It can provide compact mathematical representations of protocol concepts (e.g., connections being uncorrupted by past incarnations, eventual delivery of current incarnation data). It allows a customer to examine service specifications and decide without massive analysis whether or not technical and contractual obligations have been met.

Our construction of verified protocols

A protocol is specified by a communicating state machine at each entity consisting of state variables and events, and fairness requirements for certain events. The events model the interaction with the user and the transmission and reception of messages.

The protocol's behavior is defined by the set of event sequences generated by the interactions of the protocol state machines at each entity. Each protocol event

corresponds to an interface event or a finite sequence of interface events. Thus, each protocol event sequence corresponds to an interface event sequence. Verification consists of proving that the set of interface event sequences corresponding to the protocol event sequences satisfies the service specification.

Starting from the transport service specifications, we construct by stepwise refinement three protocols that offer the transport services for three different network services. The construction method automatically generates a verification that the protocols satisfy the transport services.

The first protocol assumes a perfect network service that delivers messages insequence without loss. The protocol is very simple, using a 2-way handshake for opening and for closing. The handshakes are required because the distant user can be closed. The incarnation numbers need not be implemented. The second protocol constructed assumes a network service that may lose messages in transit. It requires a 3-way handshake [SUN] for opening, and a a 2-way handshake for closing. The third protocol constructed assumes a network service that may lose, reorder and duplicate messages in transit. This protocol is obtained from the second protocol by having the incarnation numbers to be strictly increasing.

A natural consequence of construction is that the logical structure of our third protocol is simpler than that of TCP or ISO TP. In particular, correct connection establishment is achieved with simplicity by having separate connection request messages for active and passive opening. (The connection management function of this protocol is similar to the connection management protocol presented in [MUR1].)

Incorporation of data transfer protocols: There are several verified data transfer protocols that guarantee correct, insequence delivery of data for the various types of network services [SHAN1, SHAN2, STEN]. However these data transfer protocols are *single-incarnation* protocols: i.e., they assume that the pair of users are initialized correctly and always remain open with respect to each other. We show how any of these single-incarnation data transfer protocols can be formally incorporated with connection management protocols to yield *multi-incarnation* data transfer protocols: i.e., protocols that correctly transfer data between current incarnations. The safety and progress requirements desired of the multi-incarnation protocol *automatically* follow from the safety and progress properties established for the original single-incarnation protocol! This appears to be the first such nontrivial use of compositional progress proofs.

Related work

The best-known transport protocols today are the Internet TCP [TCP] and ISO TP [ISO1]. The protocol specification of TCP [TCP] describes in English the actions to be taken for each message reception or user request. These event processing descriptions alone are more than 25 (double-spaced) pages long. The military standard for TCP [DoD] translates these English descriptions into state transition tables to eliminate ambiguity, but the result is more than 60 pages long. In neither case is the service that is provided to the user explicitly or comprehensively specified. We believe that the intended service of TCP is exactly that offered by our protocol, with a minor difference in connection termination, as mentioned above.

While there have been verifications of fragments of TCP with respect to some desired properties, there has been no complete verification. In fact, TCP has numerous problems. Many of these problems do not show up often in practice because of the infrequent occurrence of events such as simultaneous opening (of the same socket pair), message duplication or reordering by the channels, etc. The lack of service specifications means that one cannot decide the correctness of certain behaviors of the protocol. For example, in TCP, one active opening entity and one listening entity may be prevented from establishing a connection by the presence of connection request messages from previous connection attempts. There is no way to tell if this was the original intention of the protocol designers.

The ISO TP [ISO1] includes four classes of protocols depending on the type of network service available. We are concerned with Class 4 which considers network services that may lose, reorder, or duplicate. The specification for ISO TP is somewhat more formal than that for TCP. In [ISO1], the actions to be taken on receipt of each message or user request are specified both by an English description and by a state table. However, a caveat in the state tables states that "in the event of a discrepancy between the description in these tables and that contained in the text, the text takes precedence". These state tables are quite lengthy, requiring more than 20 pages to describe the protocol actions. ISO is presently developing a more formal description in ESTELLE [ISO2] of the protocol actions.

A separate document [CCITT] specifies the intended service of ISO TP. It specifies the set of interface event sequences at each site individually. omitting the precedence relationships between interface events at different sites. This is inadequate for most purposes. In our trivial example above (Section 1.1), we would specify the allowed sequences $<ConnReq_i, ConnInd_i>$ and $<ConnReq_i, RejInd_i>$ at site i, and the allowed sequences $<Acc_j>$ and $<Rej_j>$ at site j. One cannot infer from this that $<ConnReq_i, Rej_j, ConnInd_i>$ is not allowed.

The specification in [CCITT] is incomplete and does not relate a local event to a specific occurrence of a distant event, nor does it relate the parameters of the events. Only some examples of global interface event sequences are given, using time sequence diagrams.

The ISO transport service appears to have several disadvantageous features when compared with the service offered by TCP or our protocol. The ISO TP specification does not guarantee eventual data transfer, in that a user may believe a connection is established, when the distant user is actually closed. From the ISO service specification, it appears that this is intended, but

one cannot be sure. Also, the problem of simultaneous opening by both users must be handled by the users themselves; the service specifies that two connections will be established.

More formal attempts to specify the global set of allowable event sequences for the TCP and ISO TP services have appeared in the literature [LOGR, BILL, BRNK, SIDH]. These use finite state machine models [LOGR, SIDH] or extended state machine models without safety or progress assertions [BILL , BRNK]. They do not specify progress properties that involve fairness. They model simplified services such as a one-time connection only [SIDH, BILL, LOGR]. They either do not model event parameters or do not ensure that the parameters are correctly transferred [SIDH, BILL, LOGR].

Organization of the report

In Section 2, we specify the transport service interface. In Section 3, we specify the three types of network service interfaces. In Section 4, we summarize our heuristic for constructing protocols from service specifications. In Section 5, we specify how multi-incarnation data transfer protocols can be constructed from single-incarnation data transfer protocols. In Sections 6, 7, and 8, we construct the transport protocols for perfect network service. for loss-only network service. and for loss, reordering and duplication network service.

2. TRANSPORT SERVICE SPECIFICATION

For every pair of users, i and j, we specify the transport service by an event-driven system and safety and progress requirements. An event driven system consists of a set of state variables along with their initial values, a set of events. Each event of the system is specified by an *enabling condition* and an *action*. An event may occur only when its enabling condition is true. The action specifies an update to the state variables when the event occurs.

We specify a service interface (whether transport or network) by an event-driven system together with some safety and progress assertions [LAM]. The events represent the service requests and indications. The safety assertions for the service interface are specified as invariant predicates in the state variables. We allow an interface event at i to have read-access to state variables at j, although it can only update state variables local to i. Note that the enabling conditions and actions of events actually correspond to safety constraints. We could have expressed these in the form of safety assertions too, but chose not to do so merely for reasons of convenience.

Interface state variables

For the pair i and j, we have the following history of the interface event occurrences (the events are defined below):

H_{ij} :Sequence of entries of the form (e , *param* , *olin* , *odin* , *nlin* , *ndin*), one for each interface event occurrence. Here, e is the name of the event, *param* equals the parameters of the event (if any), *olin* and *odin* equal the values of Lin_i and Din_i before the event occurred, and *nlin* and *ndin* equal the values of Lin_i and Din_i after the event occurred.

We say j **did** (*ename* , *param* , *olin* , *odin* , *nlin* , *ndin*) to mean ($ename_j$, *param* , *olin* , *odin* , *nlin* , *ndin*) $\in H_{ij}$. Similarly, we use **didnot** instead of **did** to mean the tupel is not in H_{ij} . When we refer to a tuple in the domain of entries of H_{ij} where a component in the tuple can have any of its allowed values, we shall omit that component in our reference. For example, i **did** (*ConnectReq* , *nlin* $=Lin_i$) is true if and only if H_{ij} contains some tuple which has $ConnectReq_i$ as the value for the e field, and Lin_i as the value for the *nlin* field, and any legal values for the other fields. We will use this convention with tuples from other domains also.

We now list the state variables at i . The state variables for j are similar, except that i is replaced by j . Below l and d range over nonnegative integers.

$State_i$: Initialized to *closed* .
{*closed* , *listening* , *aopening* , *popening* , *open* , *closing* }

Lin_i : integer. Initialized to 0.
The local incarnation number. Uniquely identifies every active opening and listen duration of i .

Din_i : integer $\cup$ {*null* }. Initialized to *null* .
The perception of the local incarnation number of j . Equals *null* whenever i is closed or listening.

$Source_i(l , d)$. Initialized to *empty* .
Sequence of *data* blocks sent by user i while $(Lin_i , Din_i) = (l , d)$.

$Sink_i(l , d)$. Initialized to *empty* .
Sequence of *data* blocks received by user i while $(Lin_i , Din_i) = (l , d)$.

Interface events

Recall that interface events represent user requests and transport layer indications. The interface events at i are specified formally in Table 1. The events for j are obtained by interchanging i and j .

We now discuss the notation used in specifying these interface events. We have used the notation *UpdateH* (e , *param*) to express the update to H_{ij} as a result of the occurrence of event e with parameters *param* . For example, *UpdateH* ($ConnectInd_i$, *udn*) denotes the following action:

$H_{ij} \leftarrow H_{ij}$ @
($ConnectInd_i$, *udn* , *olin* $=l_1$, *nlin* $=l_2$, *odin* $=d_1$, *ndin* $=d_2$)

where l_1 and d_1 are the values of Lin_i and Din_i before the event occurrence, and l_2 and d_2 are the values of Lin_i and Din_i after the event occurrence. Note that $l_1=l_2$ for all events but $ConnectReq_i$ and $ListenReq_i$.

New (Lin_i) denotes a function that returns an integer that has not yet appeared in the *olin* field in any entry of H_{ij} .

We now informally discuss the interface events. The events $ListenReq_i$ and $ConnectReq_i$ begin a new incarnation. The event $ListenReq_i$ indicates that user i is willing to establish a connection, if asked. The event $ConnectReq_i$ indicates that user i actively desires to establish a connection.

The events $CloseReq_i$ and $EndListenReq_i$ indicate user i's desire to end an incarnation. The event $CloseReq_i$ indicates that user i wishes to terminate an established connection. The event $EndListenReq_i$ indicates that user i is no longer willing to establish a connection.

The events $AttemptInd_i$ and $ResumeListenInd_i$ concern an attempt to establish a connection when the user is listening. They are needed to indicate entry and exit from the passive opening state. Passive opening is a necessary state between listening and open (in the case of imperfect network services). $AttemptInd_i$ indicates that an attempt to establish a connection is in progress, and the state becomes passive opening. $ResumeListenInd_i$ indicates that the attempt failed, and the state returns to listening. The attempt will fail if it was not initiated by the current incarnation of user j, or if the incarnation of user j corresponding to Din_i has never been or is no longer actively desiring to establish a connection, or has terminated a connection to a past incarnation of user i.

Event $RejectRecvInd_i$ indicates that the attempt to establish a connection for an active opening user has been unsuccessful. The user then becomes closed. Event $RejectSentInd_i$ indicates the rejection of a distant connection attempt when user i is closed. This event is needed in specifying the progress of opening (see L_{4-7} below).

A successful connection attempt is indicated by the event $ConnectInd_i$. From $ConnectInd_i$, we observe that user i becomes open only under the following conditions: (a) any one user is listening and the other is active opening, or (b) both the users are active opening. Condition (b) corresponds to a balanced opening situation (like the service offered by TCP [DoD]). Without this, we would be required to treat as two users any user who wished to both request connections and also respond to connection requests (like the service offered by ISO TP [ISO1]).

Observe also that $ConnectInd_i$ ensures that user i establishes a connection only with the current incarnation of the user j. This rules out inadvertently establishing connections with "ghosts" (incarnations that are no longer active). It also rules out "delphic" protocols which could predict the distant user's future state well enough to generate beforehand any messages required to establish a connection.

Finally, event $CloseInd_i$ indicates the termination of an established connection. From $CloseInd_i$, we observe that user i closes his end of the connection only if (a) user i open and user j is closing, (b) both users are closing, (c) user i is closing and user j has closed, or (d) user j became closing before user i could become open. Note that if data is in transit when a close request is made, there is no guarantee that it will be received before the connection is closed (see L_7 below). If such a guarantee is required, the users must wait for data to be suitably acknowledged before issuing a close request. (This is like the service offered by ISO TP.)

Safety requirements

The safety properties required of the interface are listed in Table 2. S_1 states that user i only receives data blocks that were sent by the incarnation of user j indicated by i's distant incarnation number. Further, the data blocks are received in the sequence that they were sent. S_2 states that when user i is open, user j is at least aware that the connection attempt is in progress. $S_{\bar{1}}$ and $S_{\bar{2}}$ are the properties for user j corresponding to S_1 and S_2.

S_3 is an obvious desired property. More fundamentally, we shall see in Section 4 that it is required in order for the progress of a single-incarnation data transfer protocol to guarantee the data transfer progress property L_7 below.

For any assertion S_i, we use $S_{\bar{i}}$ to denote S_i with the variable subscripts i and j interchanged. Because our service is symmetric, for every assertion S_i, we also have the corresponding symmetric assertion $S_{\bar{i}}$. Henceforth, we will usually not explicitly write the assertion $S_{\bar{i}}$. Note that $S_{\bar{3}}$ is the same as S_3. We shall also use the notation $S_{i,j}$ to denote $S_i \wedge S_j$.

Recall that the transport service has, in addition to the safety constraints stated in Table 2, the safety properties that have been encoded into the enabling conditions and actions of the interface events.

Progress requirements

The progress properties required of the interface are listed in Table 3. (As per convention, we have not explicitly listed the $L_{\bar{i}}$'s.) Progress properties L_1, L_2 and L_3 state that active opening, passive opening, and closing are transient states.

Progress property L_4 states that if both users are active opening, and neither user has already rejected the connection attempt, then at least one of the users becomes open. There are three points to note here. First, either user could have rejected the connection attempt: it could have been closed when the connection attempt was made (but subsequently decided to request a connection). Second, even if neither user has rejected the connection attempt, we cannot guarantee that both users will be open at the same time: the user who first becomes open can always immediately issue a close request, before the other user becomes open. Third, the above two behaviors hold even if the network service is perfect!

Progress properties L_5 and L_6 are similar to L_4, for the cases where j is listening or passive opening, and open, respectively. Note that in L_5, a distant user who is listening may end listening before an attempt indication occurs.

Progress property L_7 states that data that is sent in the current connection incarnation will be received eventually, unless either user issues a close request.

3. NETWORK SERVICE SPECIFICATION

The network layer provides communication services between i and j. To model this, we have the following interface state variables:

$\mathbf{z}_i$ Sequence of messages in transit from entity i to entity j, initially empty.

$\mathbf{z}_j$ Sequence of messages in transit from entity j to entity i, initially empty.

We have the following network service interface events, where @ denotes concatenation:

$Send_i(m) \equiv$ **when** $True$ **do** $\mathbf{z}_i \leftarrow \mathbf{z}_i @m$

$Rec_i(m) \equiv$ **when** $\mathbf{z}_j \neq <>$ **do** $[m \leftarrow head(\mathbf{z}_i); \mathbf{z}_i \leftarrow tail(\mathbf{z}_i)]$

In the case of imperfect network interfaces, we also have network error interface events that can delete, reorder and duplicate messages in $\mathbf{z}_i$ (depending on the type of the imperfection).

Finally, we have the following progress properties for the various network services. The rules assume that entity i is always ready to receive whatever message is at the head of $\mathbf{z}_j$, and vice versa.

Fairness rule for perfect network service:
If message m is in $\mathbf{z}_i$, then entity j eventually receives the message.

Fairness rule for imperfect network service:
If message m is repeatedly sent into $\mathbf{z}_i$, then entity j will eventually receive the message.

4. MULTI-INCARNATION DATA TRANSFER PROTOCOLS

As mentioned earlier, there are several examples of verified data transfer protocols that guarantee safe and live data transfer between two users that are initialized correctly and are always open. In this section, we show how to transform such a single-incarnation data transfer protocol into a multi-incarnation data transfer protocol.

Single-incarnation protocol

Recall that a protocol is defined by a set of state variables and a set of events. Each event can only access the state variables local to a site, either i or j. In a single-incarnation data protocol, entity i has a set of state variables, $\mathbf{v}_i$. These include $Source_i$ and $Sink_i$, which record the data blocks sent and received by user i since initialization.

We shall consider the entities to exchange messages of the type $(DATA, dtparam)$, where $dtparam$ denotes various parameters such as data, sequence numbers (in the case of imperfect network interfaces [SHAN2, STEN]), window sizes (in the case of flow control [SHAN1, SHAN2]), etc. The exact nature of the parameter $dtparam$ depends on the protocol and the type of network service being assumed.

The events of entity i are as follows:

$SI.DataSendReq_i(data) \equiv$
when ES_i **do** $[Source_i \leftarrow Source_i @data ; UpdateS_i]$

$SI.DataRecvInd_i(data) \equiv$
when ER_i **do** $[Sink_i \leftarrow Sink_i @data ; UpdateR_i]$

$SI.Send\ DATA_i \equiv$
when ESD_i **do** $[Send_i(DATA, dtparam); UpdateSD_i]$

$SI.Rec\ DATA_i \equiv$
when $ERD_i \wedge Rec_j(DATA, dtparam)$ **do** $[UpdateRD_i]$

ES_i, ER_i, ESD_i, and ERD_i are predicates in $\mathbf{v}_i$. $UpdateS_i$, $UpdateR_i$, and $UpdateSD_i$ are actions in $\mathbf{v}_i$. In the $SI.Send\ DATA_i$ event, $dtparam$ is a function of $\mathbf{v}_i$. In the $SI.Rec\ DATA_i$ event, $UpdateRD_i$ is an action in $\mathbf{v}_i$ and $dtparam$. For protocols of this form, the following safety and progress properties have been proven [SHAN2]:

$S_{dt} \equiv Sink_i$ prefix–of $Source_j$

$L_{dt} \equiv Source_i = \mathbf{m}$ *leads-to* $Sink_j = \mathbf{m}$

Multi-incarnation protocols

We obtain multi-incarnation data transfer protocols from the single-incarnation protocols as follows. Entity i has the set of variables $\mathbf{v}_i(l, d)$, for every pair of incarnation numbers l and d. This includes $Source_i(l, d)$ and $Sink_i(l, d)$. In practice, the set of variables $\mathbf{v}_i(l, d)$ is auxiliary. Only one set of variables $\mathbf{v}_i$ need be implemented, corresponding to $\mathbf{v}_i(Lin_i, Din_i)$; this is initialized each time entity i becomes open.

The data messages exchanged by the entities are now of the form $(DATA, sin, rin, dtparam)$, where sin and rin are the values of Lin_i and Din_i when the message was sent.

Table 4 specifies the events of the multi-incarnation protocol in terms of the single-incarnation protocol events. For any enabling condition or action α, the notation $\alpha[Lin_i, Din_i]$ denotes α with $\mathbf{v}_i$ replaced by $\mathbf{v}_i(Lin_i, Din_i)$. Observe that this multi-incarnation protocol is a refinement of the transport service specification. The events of Table 4 are also refinements of the single-incarnation events listed above.

We now relate the properties of the single-incarnation protocol to the properties of the multi-incarnation protocol. We have the following results (proof in [MUR2]):

Theorem 1.

If the single-incarnation protocol satisfies S_{dt} and L_{dt} and the multi-incarnation protocol satisfies S_3, then the multi-incarnation protocol satisfies S_1 in the case of imperfect network service,
$State_i = open \wedge State_j = open \Rightarrow S_1$ in the case of perfect network service, and L_7 for any network service.

From $State_i = open \wedge State_j = open \Rightarrow S_1$, it is simple to derive that S_1 is invariant in the case of perfect network service. Because S_{dt} and L_{dt} hold for the

single-incarnation protocol, it follows from Theorem 1 and the invariance of S_3 that S_1 and L_7 hold for the multi-incarnation protocol.

This completes the construction of the data transfer function of our transport protocols. The rest of our paper essentially deals with constructing a connection management protocol that satisfies the safety properties $S_{2,3}$ and the progress properties L_{1-6}. Given such a connection management protocol, the transport protocol would be defined by the state variables and events of the connection management and the data transfer protocols.

5. SUMMARY OF CONSTRUCTION HEURISTIC

We construct our transport protocols from the upper interface specifications by the method of stepwise refinement described in [SHAN3]. At any point during the construction, we have the following: A partially constructed protocol system, complete with state variables and events; a set of safety and progress requirements on the protocol; and a marking that identifies the extent to which the requirements are satisfied by the protocol system.

The protocol system satisfies the following properties with respect to the transport and network service specifications. First, the state variables of the protocol form a superset of the state variables of the transport service specification. Thus, there is a projection mapping from each state of the protocol system to a state of the service, referred to as its image at the service interface.

Second, each protocol event e_P must be a *refinement* of some transport service events, which means that if e_P can take the implementation from state s_1 to s_2, then there is a transport service event e_U that can take the upper interface from state t_1 to t_2, where t_i is the image of s_i. This condition can be relaxed by introducing a safety requirement S, in which case the condition has to be satisfied only for each (s_1, s_2) pair such that s_1 and s_2 satisfy S. We will have to prove that such safety requirements introduced are in fact safety properties of the implementation. A special case of event refinement is that e_P has a null image (i.e., t_1 equals t_2).

Third, the events of the protocol can include in their enabling conditions and actions the network service interface events, in addition to the state variables mentioned above. However, each protocol event can only access state variables local to its site. In order to achieve this, some of the variables of the transport service specification are declared to be auxiliary (e.g., a history variable).

At the start of the construction, the protocol system, safety requirements and progress requirements are obtained respectively from the event-driven system, safety assertions and progress assertions of the transport service specification. The marking is empty. At each step of the construction, the protocol system and the set of requirements are refined and the marking is increased. The construction ends when the protocol system satisfies all the requirements.

6. CONSTRUCTION ASSUMING PERFECT NETWORK SERVICE

We now construct the connection management protocol assuming perfect network service. Once this is done, the transport protocol is just the union of these sets of events with those of the data transfer protocol in Table 4. The connection management messages sent by entity i are of the form (M, sin, rin) where sin and rin are the values of Lin_i and Din_i when the message was sent. In the case of perfect network service, we will see that the sin and rin fields can be made auxiliary, and need not be implemented.

A protocol event may only reference local variables. Thus, our first refinement step is to replace all terms of the form j **did** (e) in the service events by an appropriate message reception. We introduce the following messages: $(CR, sin, null)$ which requests a connection; $(CRACK, sin, rin)$ which accepts a connection request; (REJ, sin, rin) which rejects a connection request; $(DATA, sin, rin, data)$ which transfers data; (DR, sin, rin) which requests connection closing; and $(DRACK, sin, rin)$ which acknowledges the connection closing.

By replacing terms of the form j **did** (e) in the service events by the appropriate Rec_j constructs, we obtain the events of the protocol for perfect network service. (Below, we illustrate this with the $ConnectInd_i$ event.)

Consider the service event $ConnectInd_i$ when $State_i = aopening$. This has the term j **did** $(ConnectReq)$. Replacing this term by $Rec_j(CR, sin, rin)$, we get the event

when $Rec_j(CR, sin, rin) \wedge State_i = aopening$ **do**
$[State_i \leftarrow open ; Din_i \leftarrow sin ; UpdateH(ConnectInd_i, sin)]$

We introduce the following safety requirements to ensure that this event is a refinement of $ConnectInd_i$.

S_4: $State_i = open \Rightarrow Din_i = Lin_j$
S_5: $(CR, x, y) \in \mathbf{z}_j \Rightarrow j$ **did** $(ConnectReq, nlin = x)$

Similarly refining the service events, in [MUR2] we construct a protocol for perfect network service. The protocol is a simple 2-way handshake for connection establishment -- either $CR_i - CRACK_j$, $CR_i - CR_j$, or $CR_i - REJ_j$. Connection termination is also a 2-way handshake -- either $DR_i - DRACK_j$ or $DR_i - DR_j$. The protocol makes no decisions based on the values of Lin_i and Din_i. The data transfer event specifications in Table 4 can also be simplified by eliminating references to Lin_i and Din_i. Therefore, these are auxiliary variables that need not be implemented.

The proof that this protocol satisfies the safety and progress assertions of the service and the safety assertions needed for refinement is given in [MUR2].

7. CONSTRUCTION ASSUMING LOSSY NETWORK SERVICE

We construct a protocol for lossy network service by refining the protocol constructed for perfect network service. In the construction of a protocol for perfect net-

work service, we could rely on the network to deliver any message sent. Hence, each message was sent exactly once. In constructing a protocol for lossy network service, it is necessary to retransmit messages. Because of retransmissions, a listening (or active opening) user who receives a *CR* cannot be certain that a previous copy of the *CR* was not rejected while the user was in his previous state of closed. Such a reject could cause the distant entity to close. It is necessary therefore to request confirmation that the distant user has not closed. This introduces a third message, making a 3-way handshake: CR_i-$CONFIRM_j$-$CRACK_i$, or CR_i-$CONFIRM_j$-REJ_i. This message also must be retransmitted to make sure that it is received and the *CRACK* or *REJ* response is returned.

Now, however, the (*CONFIRM*, *sin* , *rin*) message may be received from user j when Lin_j is no longer *sin* . This could happen much as with the *CR* -- several *CONFIRM*'s are sent, one is rejected by the distant user, the local user closes, the distant user becomes active opening and receives one of the other copies. But this *CONFIRM* is not a response to the latest *CR* transmitted by the local user. Thus, the local user should not become open on receiving a *CONFIRM*; there is no guarantee that it is from the current distant user or that the distant user is not closed. Some information should be included by the distant user in the *CONFIRM* which could be checked as being current. But the only current information the distant user might know that the local user could check is the local *Lin* .

The solution is to require that the protocol return in the *CONFIRM* the sender (*sin*) of the *CR* being confirmed. The most obvious way to do this is to include $Din_i \leftarrow sin$ in the event that receives a *CR*, send (*CONFIRM*, Lin_i, Din_i), and check $rin = Lin_i$ in the event that receives a *CONFIRM*. For similar reasons, the *rin* on a received *DR* or *REJ* should be checked against *Lin* .

Note that the *DRACK* sent in response to a *DR* cannot be retransmitted forever, because the user may become active opening or listening. In case the *DRACK* gets lost, then, a closing user is allowed to close if a *CR* or *REJ* is received.

Finally, observe that it is possible for a listening user to receive a *CONFIRM* when the distant user has received a *CR* from some previous incarnation. If the *CONFIRM* was from an active opening entity, the protocol should respond, in such a way that the connection becomes established. This is mandated by the progress assertion that says a listening user and an active opening user should eventually establish a connection. But if the *CONFIRM* was from a passive opening user, establishing a connection would violate the specification of the *ConnectInd* event. The progress assertion that says the passive opening state is transient mandates that some response be made. Hence, a listening user must be able to distinguish between a *CONFIRM* from an active opening user and one from a passive opening user. The *CONFIRM* message is therefore split into two -- *CRAO* (which combines the function of *CR* and *CONFIRM* for active opening users) and *CRPO* (the *CONFIRM* for a passive opening user). A new message is also introduced called *RESET* which is sent by a listening or passive opening user on receipt of a *CRPO* and which will, when received, cause a passive opening user to resume listening.

At this point, we have completed the construction for a loss only lower interface. We now proceed with the construction for the loss/recorder/duplicate interface.

8. CONSTRUCTION FOR A LOSS, REORDER AND DUPLICATION NETWORK SERVICE.

We construct a protocol for loss/reorder/duplicate network service by refining the protocol constructed for lossy network service. With a loss/reorder/duplicate network, it is possible for messages to arrive in any order. The protocol behavior must be altered to ensure that the safety and progress requirements still hold. Observe that it is possible that a *REJ* could arrive after a *CRAO*. This could happen if the network delivered an old *CRAO* to user i before a current *REJ*. It could also happen that user j could receive a connection request from user i and send a *REJ*, then change state and send a *CRAO* of its own that gets delivered first. In both cases the *rin* of the *REJ* is Lin_i. If a *REJ* that is received after a *CRAO* is accepted, then in the second case user i would be closed and user j could open, which violates the safety assertion S_2. If such a *REJ* is ignored, then in the first case user j could be permanently in a transient state. The problem is that the protocol does not know which of these two messages is the more current. The solution chosen is to insist that incarnation numbers be strictly increasing for each user. Note that Din_i always takes its value from the *sin* field of some received message. Also, when a message is sent, $rin = Din_i$ or $rin = sin$ of some recently received message. We therefore introduce the following safety assertion about messages in transit:

$B_1 \equiv (M, sin, rin) \in \mathbf{z}_i \Rightarrow sin \leq Lin_i \wedge rin \leq Lin_j$

Note that B_1 does not say that incarnation numbers must be increasing for all time, but that the user may be assured that the messages in the channel have this property. In other words, it is possible to reuse incarnation numbers that have not been used for a sufficiently long time.

In general, when a message (M, sin, rin) is received, if $sin < Din_i$ the message may be ignored. Hence, if $Din_j = Lin_i$ it will not get set to a new value, unless that value is *null* . Note that because of the three way handshake, we can now strengthen S_3 by changing its antecedent from $State_i = open \wedge State_j = open$ to $State_i = open$.

The events of the final protocol are given in [MUR2], as is a proof that this protocol is a refinement of the service specification, and satisfies the service safety and progress requirements. Some of the details of the proof are presented in [SHAN4]. The final protocol and proof are very similar to the ones presented in [MUR1]. For brevity, we have not given the proof here.

9. CONCLUSION

We defined a formal abstract specification of the transport service interface. We constructed transport protocols for three types of network service, using the method of stepwise refinement. This demonstrates the usefulness of service specifications that are sufficiently abstract to apply to different environments; and the stepwise refinement heuristic in constructing protocols.

The construction brings out some interesting relationships between the protocol and the network services, such as the need for a 3-way handshake with a loss-only network service, and the need for strictly increasing incarnation numbers with a loss, reordering and duplication network service.

We have incorporated existing data transfer protocols into the construction in such a way that their safety and progress properties automatically hold. This appears to be the first such nontrivial use of compositional progress proofs.

REFERENCES

[BILL] J. Billington, "Specification of the Transport Service Using Numerical Petri Nets", *Protocol Specification, Testing, and Verifications II*, 1982, ed C. Sunshine, North-Holland, 1982.

[BRNK] E. Brinksma and G. Karjoth, "A Specification of the Transport Service in LOTOS", *Protocol Specification, Testing, and Verifications, IV*, 1984, North-Holland, 1985.

[CCITT] CCITT, *Transport Service Definition for Open Systems Interconnection (OSI) for CCITT Applications*, International Telegraph and Telephone Consultative Committee Recommendation X.214, 1985.

[DIJK] E. W. Dijkstra, *A Discipline of Programming*, Prentice Hall, 1975.

[DoD] *Transmission Control Protocol*, DDN Protocol Handbook: DoD Military Standard Protocols, DDN Network Information Center, SRI, MILSTD1778, Aug 1983.

[HAAS] O. Haas, "Formal protocol specification based on attribute grammars," *Protocol Specification, Testing, and Verifications V*, 1985, ed M. Diaz, North-Holland, 1985.

[ISO1] International Organization for Standardization, *Information Processing Systems* – Open Systems Interconnection – Transport Protocol Specification, ISO DIS 8073, 1985.

[ISO2] International Organization for Standardization, *A Formal Description of ISO 8073 in ESTELLE*, Working draft, ISO TC 87/SC 6/WG 4 N14, Mar 1985.

[LAM] S. S. Lam and A. U. Shankar, "Specifying an Implementation to Satisfy Interface Specifications: A State Transition Approach," presented at the 26th Annual Lake Arrowhead workshop, Sept. 16-18, 1987.

[LOGR] L. Logrippo, D. Simon, and H. Ural, "Executable description of the OSI Transport Service in Prolog," *Protocol Specification, Testing, and Verifications, IV*, 1984, North-Holland, 1985.

[MANN] Z. Manna and A. Pnueli, "Adequate Proof Principles for Invariance and Liveness Properties of Concurrent Programs," *Science of Computer Programming*, Vol. 4, 1984.

[MUR1] S. L. Murphy and A. U. Shankar, "A verified connection management protocol for the transport layer," *ACM SIGCOMM '87 Workshop*, Stowe, Vermont, August 1987.

[MUR2] S. L. Murphy and A. U. Shankar, "Service Specification and Protocol Construction for the Transport Layer," CS-TR-2033, Dept. of Computer Science, University of Maryland, May 1988.

[SHAN1] A. U. Shankar, "A Verified Sliding Window Protocol with Variable Flow Control", *Proc. ACM SIGCOMM* '86, Aug 1986.

[SHAN2] A. U. Shankar, "Verified data transfer protocols with variable flow control", CS-TR-1746, Dept. of Computer Science, University of Maryland, Mar 1987, conditionally accepted in *ACM TOCS*.

[SHAN3] A. U. Shankar and S. S.. Lam, "A Stepwise Refinement Heuristic for Protocol Construction", CS-TR-1812, Dept. of Computer Science, University of Maryland, Mar 1987.

[SHAN4] A. U. Shankar, "A Connection Management Protocol for the Transport Layer and Its Verification", submitted for publication.

[SIDH] D.P. Sidhu and C.S. Crall, "Executable Logic Specifications for Protocol Service Interfaces", *IEEE Trans. on Soft. Engr.* Vol 14-1, Jan 1988.

[STEN] Stenning, N. V., "A data transfer protocol," *Computer Networks*, Vol. 1, pp. 99-110, September 1976.

[SUN] C. A. Sunshine and Y. K. Dalal, "Connection Management in Transport Protocols", *Computer Networks*, Vol.2(6), Dec 1978.

[TCP] *Transmission Control Protocol*, Request for Comments 793, Network Information Center, SRI International, 1981.

[VISS] Ch. A. Vissers, L. Logrippo, "The importance of the service concept in the design of data communications protocols," *Protocol Specification, Testing, and Verifications V*, 1985, ed M. Diaz, North-Holland, 1985.

Table 1: Transport service interface events at i

$ListenReq_i \equiv$
  when $State_i = closed$ **do** $[Lin_i \leftarrow New(Lin_i);\ State_i \leftarrow listening;$
  $UpdateH(ListenReq_i)]$

$ConnectReq_i \equiv$
  when $State_i = closed$ **do** $[Lin_i \leftarrow New(Lin_i);\ State_i \leftarrow aopening;$
  $UpdateH(ConnectReq_i)]$

$CloseReq_i \equiv$
  when $State_i = open$ **do** $[State_i \leftarrow closing;\ UpdateH(CloseReq_i)]$

$EndListenReq_i \equiv$
  when $State_i = listening$ **do** $[State_i \leftarrow closed;\ UpdateH(EndListenReq_i)]$

$DataSendReq_i(data) \equiv$
  when $State_i = open$ **do** $[Source_i(Lin_i, Din_i) \leftarrow Source_i(Lin_i, Din_i)@data;$
  $UpdateH(DataSendReq_i, data)]$

$AttemptInd_i(d) \equiv$
  when $State_i \in \{listening, popening\} \wedge j$ **did** $(ConnectReq, nlin = d)$
  do $[Din_i \leftarrow d;\ State_i \leftarrow popening;\ UpdateH(AttemptInd_i, d)]$

$ResumeListenInd_i(Din_i) \equiv$
  when $State_i = popening \wedge [Din_i \neq Lin_j \vee j$ **didnot** $(ConnectReq, olin = Din_i) \vee$
  j **did** $(RejectSentInd, olin = Din_i) \vee j$ **did** $(CloseReq, olin = Din_i, odin \neq Lin_i)]$
  do $[Din_i \leftarrow null;\ State_i \leftarrow listening;\ UpdateH(ResumeListenInd_i, Din_i)]$

$RejectSentInd_i(d) \equiv$
  when $State_i = closed \wedge [j$ **did** $(ConnectReq, nlin = d) \vee j$ **did** $(ListenReq, nlin = d)]$
  do $UpdateH(RejectSentInd_i, d)$

$RejectRecvInd_i(d) \equiv$
  when $State_i = aopening \wedge j$ **did** $(RejectSentInd, olin = d)$
  do $[Din_i \leftarrow null;\ State_i \leftarrow closed;\ UpdateH(RejectRecvInd_i, d)]$

$ConnectInd_i(d) \equiv$
  when $d = Lin_j \wedge$
  $[(State_i = aopening \wedge j$ **did** $(ConnectReq, nlin = d)) \vee$
  $(State_i = aopening \wedge j$ **did** $(AttemptInd, olin = d)) \vee$
  $(State_i = popening \wedge j$ **did** $(ConnectReq, nlin = d))]$
  do $[Din_i \leftarrow d;\ State_i \leftarrow open;\ UpdateH(ConnectInd_i, d)]$

$CloseInd_i(d) \equiv$
  when $[(State_i = open \wedge j$ **did** $(CloseReq, olin = d)) \vee$
  $(State_i = closing \wedge j$ **did** $(CloseReq, olin = d)) \vee$
  $(State_i = closing \wedge j$ **did** $(CloseInd, olin = d)) \vee$
  $(State_i \in \{aopening, popening\} \wedge j$ **did** $(CloseReq, olin = d, odin = Lin_i))]$
  do $[Din_i \leftarrow null;\ State_i \leftarrow closed;\ UpdateH(CloseInd_i, d)]$

$DataRecvInd_i(data) \equiv$
  when $State_i = open$
  do $[Sink_i(Lin_i, Din_i) \leftarrow Sink_i(Lin_i, Din_i)@data;\ UpdateH(DataRecvInd_i, data)]$

Table 2: Safety requirements for transport service interface

$S_1 \equiv Sink_i(Lin_i, Din_i)$ prefix-of $Source_j(Din_i, Lin_i)$

$S_{\bar{1}} \equiv Sink_i(Lin_j, Din_j)$ prefix-of $Source_i(Din_j, Lin_j)$

$S_2 \equiv State_i = open \Rightarrow State_j \in \{aopening, popening, open, closing\}$

$S_{\bar{2}} \equiv State_j = open \Rightarrow State_i \in \{aopening, popening, open, closing\}$

$S_3 \equiv State_i = open \wedge State_j = open \Rightarrow (Lin_i, Din_i) = (Din_j, Lin_j)$

Table 3: Liveness requirements for transport service interface

$L_1 \equiv State_i = aopening$ *leads-to* $State_i \in \{open, closed\}$

$L_2 \equiv State_i = popening$ *leads-to* $State_i \in \{listening, open, closed\}$

$L_3 \equiv State_i = closing$ *leads-to* $State_i = closed$

$L_4 \equiv State_i = aopening \wedge State_j = aopening$
  $\wedge\ j$ **didnot** $(RejectSentInd, param = Lin_i)$
  $\wedge\ i$ **didnot** $(RejectSentInd, param = Lin_j)$
  leads-to $State_i = open \vee State_j = open$

$L_5 \equiv State_i = aopening \wedge State_j \in \{listening, popening\}$
  $\wedge\ j$ **didnot** $(RejectSentInd, param = Lin_i)$ *leads-to*
  $State_i = open \vee State_j = open \vee j$ **did** $(EndListenReq, olin = Lin_j)$

$L_6 \equiv State_i \in \{aopening, popening\} \wedge State_j = open$
  leads-to $State_i = open \vee j$ **did** $(CloseReq, olin = Lin_j)$

$L_7 \equiv State_i = open \wedge State_j = open \wedge Source_i(Lin_i, Din_i) = \mathbf{m}$
  leads-to $Sink_j(Din_i, Lin_i) = \mathbf{m}$
  $\vee\ i$ **did** $(CloseReq, olin = Lin_i) \vee j$ **did** $(CloseReq, olin = Lin_j)$

Table 4: Multi-incarnation data transfer protocol events at i

$MI.DataSendReq_i(data) \equiv$
  when $State_i = open \wedge ES_i[Lin_i, Din_i]$ **do**
  $[Source_i(Lin_i, Din_i) \leftarrow Source_i(Lin_i, Din_i)@data;\ UpdateS_i[Lin_i, Din_i];$
  $UpdateH(DataSendReq_i, data)]$

$MI.DataRecvInd_i(data) \equiv$
  when $State_i = open \wedge ER_i[Lin_i, Din_i]$ **do**
  $[Sink_i(Lin_i, Din_i) \leftarrow Sink_i(Lin_i, Din_i)@data;\ UpdateR_i[Lin_i, Din_i];$
  $UpdateH(DataRecvInd_i, data)]$

$MI.Send\ DATA_i \equiv$
  when $State_i = open \wedge ESD_i[Lin_i, Din_i]$
  do $[Send_i(DATA, Lin_i, Din_i, dtparam);\ UpdateSD_i[Lin_i, Din_i]]$

$MI.Rec\ DATA_i \equiv$
  when $ERD_i[Lin_i, Din_i] \wedge Rec_j(DATA, sin, rin, dtparam)$ **do**
  if $State_i = open \wedge (sin, rin) = (Din_i, Lin_i)$ **then** $UpdateRD_i[Lin_i, Din_i]$
  else *skip*