Tutorial on Estelle and Early Testing

Richard L. Tenney*†
University of Massachusetts
Boston, MA 02125-3393

Abstract

This paper presents a tutorial introduction to Estelle, a formal description technique developed within ISO for specifying OSI. It explains the Estelle description of the Abracadabra protocol found in *Guidelines for the Application of Estelle, LOTOS, and SDL* and then discusses some initial tests that should be performed for protocols. These tests expose some weaknesses of the Abracadabra protocol as presented.

1 Introduction

Estelle [ISO, IS9074] and LOTOS [ISO, IS8807] are the two Formal Description Techniques developed within the International Organization for Standardization (known as "ISO") Open Systems Interconnection (OSI) project during the 1980's. Estelle is based on communicating finite automata, while LOTOS is based on communicating processes. These were both designed to be used to specify the services and protocols of OSI, but each has found wider application in specifying more general distributed systems as well.

In deciding to base Estelle on extended finite automata, its designers observed that much communications software is written with this model at its base. Thus as events are received, a dispatch table based on the current state is consulted to determine the actions to be performed and the state to enter next. Even before formal descriptions were commonly regarded as necessary, informal descriptions of protocols usually included a state diagram and sometimes even a state table to make their descriptions more precise (see *e.g.* [Postel, 1980].) Even today when protocols are discussed without formal descriptions one often finds finite state descriptions (see *e.g.* [Rose, 1991, page 62] or [Schwartz, 1987, page 349]). Extensive finite state descriptions appear in various ISO OSI protocols (*e.g.* [ISO, IS8073]).

*Although the author was the editor for Estelle [ISO, IS9074] and remains the maintenance editor for that International Standard, the views expressed in this paper are strictly personal views and do not represent an official position of the International Organization for Standardization nor of any of its members.

†The software to produce figure 3 and to do much of the testing reported in section 5 was contributed by Tom Blumer of Phoenix Technologies, Ltd.

Estelle is based on several earlier techniques but also contains several features found in none of them. A summary of the status of formal description techniques at the time the work on Estelle began can be found in [Bochmann and Sunshine, 1980].

This paper is a tutorial on Estelle, with some discussion about how one would begin to experiment with a protocol using Estelle. It is organized into three major portions, the first giving the fundamental notions underlying Estelle, the second covering a rather complete (and, for a tutorial, complex) protocol expressed in Estelle, explaining the language features as they are encountered, and the third presenting some analysis of the protocol.

2 Estelle Fundamentals

Specifications in Estelle comprise systems of structured, extended finite automata communicating through channels. Finite automata are well-established abstract mathematical models of computation devices. (See [Moore, 1964] for a collection of some early papers and a bibliography of additional early papers.)

Any interesting protocol uses some data. Even simple protocols may include sequence numbers that range up to (say) 128. In a finite automaton, each possible value of each variable must be accounted for in all possible configurations. With a connection that undergoes just four states (*e.g.*, Idle, Opening, Established, Closing) and just two sequence numbers modulo 128 (one for transmitting and another for receiving), a pure finite automaton approach would require at least $4 \times 128 \times 128 = 65536$ states. Clearly this is unacceptable. Estelle, by extending finite automata to include variables, reduces this to four states and two variables.

One of the key observations to make is that although the finite automaton descriptions of protocols mentioned above are informative, they are not complete. Without further information, it is not possible either to check or to implement the protocols. Estelle provides a way to add this necessary information.

There are three major components to a system described using Estelle: (*i*) channels, (*ii*) extended finite automata, called *modules*, and (*iii*) structure of the system. We briefly describe each of these in turn. We shall examine these in more detail in section 4, where we shall also introduce the language constructs used to specify these components of Estelle.

2.1 Channels

Channels are thought of as connecting modules. A message (called an *interaction*) placed in one end of a channel ends up in an input queue of the module at the other end of the channel. As discussed below (section 4.6), there are some subtle interactions between this simple concept and the structuring of modules allowed in Estelle. Channels are reliable: any message sent is delivered immediately,[1] once, unchanged, to the correct

[1]Since this is a tutorial, we take the liberty of deviating slightly from the truth — the actual mechanism described in formal semantics of Estelle is quite complicated in order to guarantee certain desirable properties about the interleaving of interactions that arrive at a module from disparate modules, but

recipient. Only those kinds of interactions that are named when the channel is declared are permitted to pass in each direction through a channel.

As interactions are received by a module, they are placed at the end of a queue that can grow arbitrarily long; thus there is always room to add another interaction. Depending on the specification, this queue may be associated with only a single channel, or it may be the module's common queue, which may be shared by several channels. The queues are well-behaved: they neither corrupt nor re-order the data in them.

2.2 Extended Finite Automata

As noted above, finite automata are inadequate succinctly to capture all the details of even most simple protocols. It is thus necessary to extend the notion of a finite automaton. The first step is to add the ability to store values for variables. These variables can be used to store data to be sent, sequence numbers of numbered interactions, partially formed interactions, *etc.*

Although there are several variants of finite automata in the literature, they differ only in some of their details. For our purposes we say that an ordinary finite automaton begins in a specified state and whenever it receives an input it makes a transition from its current state to the next state, possibly making an output as it does so. The choice of transition to make is determined by the specification of the automaton. In general, the choice of transition is non-deterministic, because a well-formed specification of a finite automaton may allow any of several transitions to be used under a given circumstance. However, each time there is a choice to be made, randomly one of the allowable choices will be made.

Estelle modules begin with this same notion and go further by allowing for multiple input opportunities (*interaction points*) and by allowing the choice of transition to depend on the values associated with some of the stored variables. In addition, Estelle modules may have transitions that do not depend on any inputs (*spontaneous transitions*), and these spontaneous transitions may have delays associated with their applicability. This notion of a delay transition is explained in more detail below in section 4.4.1.

An Estelle module must be able to examine and manipulate the values associated with its variables. It must also be able to cause outputs to be sent through any interaction point. Finally, it must also be able to manage the structure and interconnection of the system of modules.

The selection and firing of a transition form a single atomic act, so intermediate values assumed by variables, even if exported to other modules, are never available, and possible alteration of external conditions cannot intervene.

this is too high a level of sophistication to be interesting in a tutorial, so we suppress it. We beg the indulgence of the *cognoscenti*.

2.3 System Structure

An Estelle specification comprises a collection of systems of nested modules. These may be written in such a way that they model several independent systems or so that they model a single, tightly coupled system, or almost any situation between these extremes.

From the outside, it is not possible to tell anything about the internal structure of an Estelle module by its observed behavior. However any module may contain submodules. A module and any of its submodules are referred to as *parent* and *child* respectively. Naturally, the transitive closures of these two relations give rise to *ancestor* and *descendant*. A parent may create and destroy children modules, so the structure of a system may be dynamic. A parent is responsible for the connection of channels to its children, both with each other and to its own internal interaction points.

To facilitate the structuring of modules into submodules there is an attachment mechanism that causes interactions directed from outside a parent module to be processed by one of its child modules without intervention by the parent. This mechanism is distinct from the connection mechanism, but both are forms of *binding*. They will be discussed below in section 4.6.

A parent module is separate from its children modules; they do not share variables, except that a parent has access to those variables of a child module that the child chooses to export. In part to prevent possible race conditions that might occur because of this sharing, a *parent/child priority* is imposed, whereby a child is prevented from making a transition if any of its parent's transitions is enabled.

Some modules may have only initialization transitions that serve to create and bind children. After that they must necessarily remain dormant, because they have no other transitions. Such a module is called *inactive*. By contrast, a module that does have transitions in addition to its initialization is called *active*.

A major use of inactive modules is to set up the overall structure of the system being specified. If all the antecedents of a module are inactive, then the module cannot be removed nor its bindings altered after it is created. Such a module may be designated a *system* module (unless one of its antecedents has already been designated to be a system module). Because of parent/child priority, a system's descendants are sometimes thought of as being tightly coupled, while systems themselves are only loosely coupled. Systems may communicate only by exchanging interactions through channels.

Sibling modules may either run in a synchronous parallel fashion or in an interleaved parallel fashion. The choice is indicated by the attribute of the parent, which must be either a *process* or an *activity*, respectively. The children of an activity must themselves all be activities, while the children of a process may be either processes or activities. As the default is to inherit this attribute from the parent, it is necessary to ensure that any active module has an attributed ancestor. Systems themselves are thus either system activities or system processes.

There is no guarantee of fairness in Estelle semantics, so that if two sibling activities are always able to make a transition, one of them may always fire and the other never fire. Similarly, if two transitions with the same priority are always simultaneously enabled, one of them may always fire and the other one may never fire. It is therefore the task of the specifier to guarantee that the system specified performs as required even in these extreme cases.

3 Abracadabra Example

To make the discussion of Estelle more concrete, we shall make use of an example. Our example will be a version of the alternating bit protocol [Bartlett *et al.*, 1969], a data transfer protocol that uses a single-bit sequence number (that alternates between 0 and 1, hence the name). This is often used as a didactic protocol (see *e.g.*[Merlin, 1979], [Blumer and Tenney, 1982] and [Tarnay, 1991]). It is interesting to note that its original published description is a pair of symmetric finite automata. By itself, however, it is too simple to exhibit many interesting features of modern protocols, so we use a version that adds retransmission on timeouts and simple connection and disconnection procedures. The resultant protocol is described fully in *Guidelines for the Application of Estelle, LOTOS, and SDL* [ISO, TR10167, clause 10], where it is called "Abracadabra". Although it is simpler than most actual protocols, it is nevertheless complex enough to be interesting. An updated version of this protocol and description appears in [Turner, 1993].

3.1 Abracadabra Service

In many respects Abracadabra is like a data link or a transport protocol: it provides a reliable, connection-oriented service between a pair of users. A full data link or transport protocol would have to handle addressing, communication failures, multiplexing, management functions, *etc.*, but the structure of its specification could easily follow that of the Abracadabra specification below. The difference in specification would mainly be a difference in amount not in kind.

A simple service diagram for Abracadabra is shown in figure 1. This diagram shows only the most fundamental uses of the protocol: a simple connection, data transfer, and disconnection. In such diagrams, time increases from top to bottom. Note that only connection is a confirmed service, meaning that the user who initiated the ConReq ultimately receives an explicit ConConf as a response. For the other services, the user simply trusts that the appropriate actions take place.

There are many things that the service diagram does not show. For example, not shown is the fact that the service is completely symmetric. In the interest of simplicity, the service diagram also ignores less likely occurrences, like two ConReq's being issued simultaneously by the two Users. Rather than burden the reader with increasingly complex service diagrams, it is common to put much of the information about abnormal and unusual behavior into the protocol description.

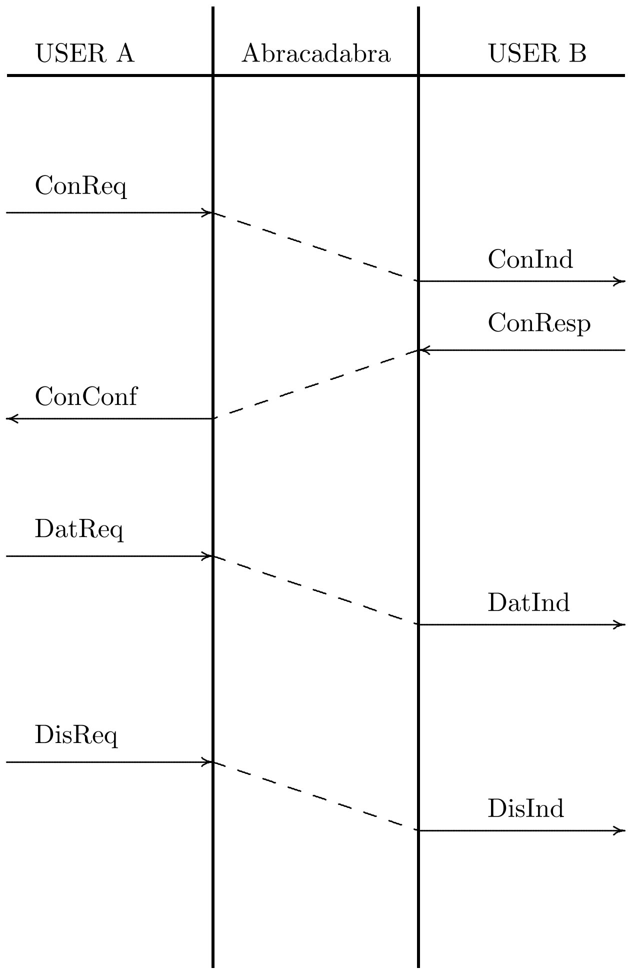

Figure 1: Abracadabra Service Diagram

3.2 Abracadabra Protocol

The protocol to implement the Abracadabra service is relatively straightforward. We assume that there are two users, A and B, and that each of them communicates with an Abracadabra protocol entity (which we will call a *Station*) that implements the service. These two stations are considered to be peers. They communicate by sending *Protocol Data Unit*s (*PDU*s) through a full-duplex, unreliable communications medium that may lose or delay messages. However, we assume the medium will never corrupt, reorder, duplicate, or originate messages. This system is shown in figure 2.

User A requests a connection with User B by initiating a ConReq. Station A sends a CR (*Connect Request*) PDU to Station B. When this arrives, Station B issues a ConInd to User B, who (assuming a willingness to connect) replies with a ConResp to Station B. Station B sends a CC (*Connect Confirm*) back to Station A, which issues a ConConf to User A. After this, either user may send data. A user's DatReq is conveyed as part of a DT (*DaTa*) PDU, and a DisCon is conveyed as a DR (*Disconnect Request*) PDU.

If everything always worked as desired, this would be the end of it!

However, even the connection phase of the exchange can become complicated. What is to happen if the CR is lost due to a fault in the communications medium? What if the CC is lost? What if User B is not willing to accept the connection. What if User A decides to abandon the connection before it has been completed? What if User A tries to send data before being informed that the connection exits?

Furthermore, the User trusts that data are transferred without requiring an explicit confirmation, but the communications medium is not assumed reliable, so the Stations must arrange some kind of confirmation between themselves and recovery mechanisms for the cases where messages are lost by the medium.

Rather than explain the full protocol in English, it will be more instructive to describe it in Estelle, with English explanations. This is done in the next section. We remark that the Estelle description closely follows the English description of [ISO, TR10167], both in content and in order.

4 Estelle Specification

To express the portions of an Estelle specification that rely on traditional programming (*e.g.*, manipulation of variables and flow of control), Estelle uses a language that is based on level 0 ISO Pascal [ISO, IS7185]. Estelle enhances Pascal in various ways. The integers and real numbers of Estelle are the usual mathematical ones; implementation details like a maximum largest integer and precision of real numbers are not considered. Also, functions are allowed to return arbitrary types.[2] The `goto` statement is restricted to make it act like a return. Since ordinary file input and output are not part of communications specification, all those features of Pascal that relate to I/O were removed from Estelle. Finally, the keyword `specification` was substituted for `program`.

[2]But note that the syntax of an expression was not changed, so the returned value of a function can be used in only very simple ways.

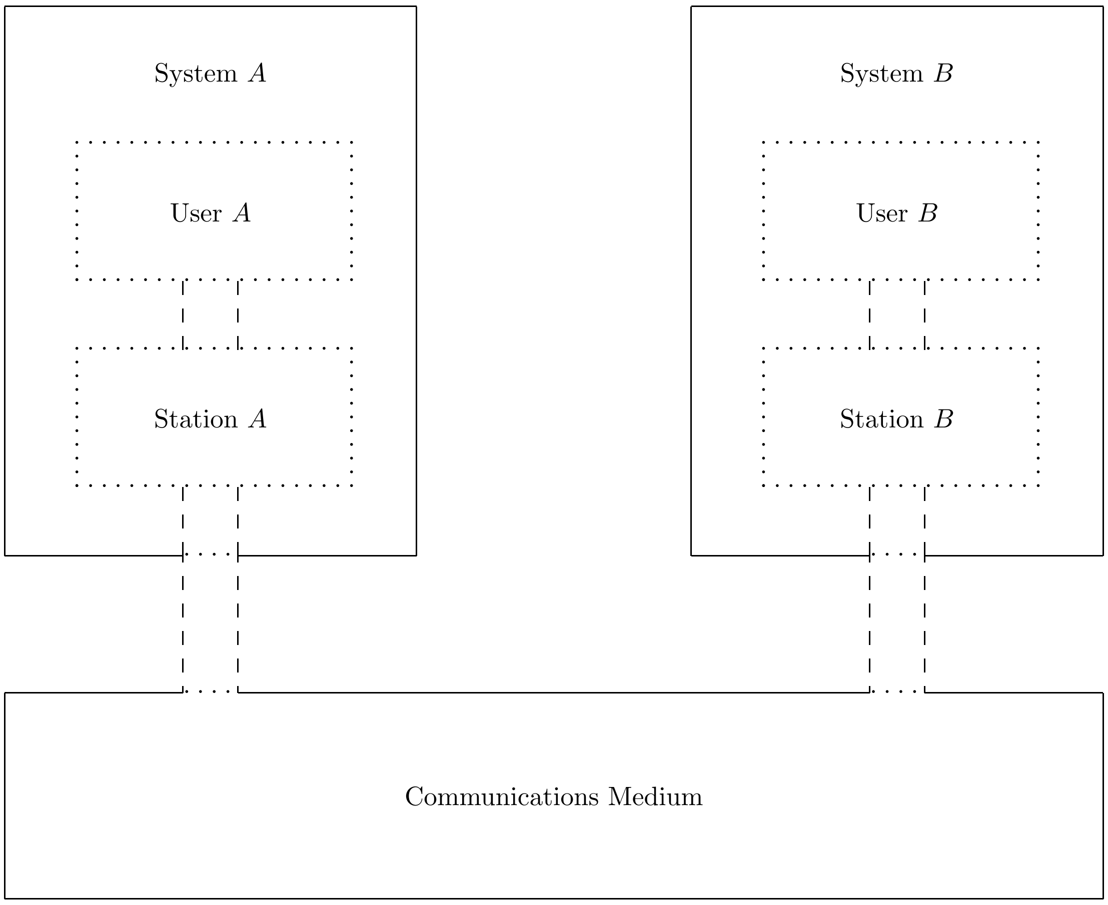

Figure 2: Simple Abracadabra System

Other language constructs were created to deal with the aspects of OSI specification. These will be discussed as needed in going through the Abracadabra protocol specification.

The version of the Abracadabra protocol given here was taken with only a few cosmetic changes from [ISO, TR10167, clause 10.3.4]. The entire description is appended to this paper.

4.1 Global Parameters

We begin with the beginning of the specification.[3] One of the interesting things to note is that the two communicating systems are completely symmetric: there is but one description of them and two instantiations. This is in contrast to the original description of the simpler alternating bit protocol in [Bartlett *et al.*, 1969] where two different automata are given.

```
1    specification Abracadabra;
2
3      default individual queue;
4      timescale seconds;
```

Line *1* merely names the specification, much like the `program` statement of a Pascal program. There is no semantic meaning attached to the identifier `Abracadabra` except to preclude its use elsewhere in the specification.

The default queueing discipline given at the specification level, as in line *3*, applies to each interaction point in each module that does not have its own queueing discipline explicitly given. If no specification-wide default is given, then each interaction point must have an explicit discipline. The two choices are `common queue` and `individual queue`. There may be at most one common queue in each module, but there may be many individual queues.

The `timescale`, line *4*, is intended to work with delay transitions (see section 4.4.1), but it is technically a comment, with no semantic meaning. However, as an aside, we note that the formal meaning of an Estelle specification may be thought of as a tree of global instantaneous descriptions, where paths through the tree represent possible execution sequences. Outside constraints remove branches from this tree, and the time scale can provide such a constraint.

```
6      const
7          N = any integer;  { number of transmission attempts }
8          P = any integer;  { delay amount for timers }
```

Two constant parameters characterize any instance of the Abracadabra protocol. The first of these, `N`, the maximum number of times a PDU may be sent before the Station gives up, is specified in line *7*. The second, `P`, the amount of time (in seconds, subject to the discussion of `timescale` above) between retransmission attempts, is given in line *8*.

Although the specification here forces the values ultimately associated with `N` and `P` to be

[3]The line numbers in italics at the beginning of the lines are not part of the specification itself, but are added to facilitate talking about the specification.

integers, Estelle does not have a way to restrict the value of these constants here, so they could even take negative values. However in line *580*, where the body of the specification is initialized, the transition cannot fire unless `N` and `P` are both positive. This unfortunately large distance between the declarations and the checking of their values is one of the costs of basing Estelle on Pascal. Again in passing we note that the formal semantics of Estelle require that some value be assigned to each constant that is declared using `any`, so that the specification actually determines a collection of specifications, one for each possible value of `N` and `P`.

Lines *7* and *8* both contain comments. These may be inserted at any place that a space could be inserted into the text. The alternate forms available in Pascal, using `(*` for { and `*)` for }, are also acceptable in Estelle.

Also in keeping with Pascal conventions, the representation (upper- versus lower-case, font, *etc.*) of a character is insignificant except within character-string constants, and spacing on the page, while potentially helpful to human readers, is of no importance to the meaning of the specification.

```
10      type
11          SeqType = 0..1;         { sequence number type }
12          UserDataType = ...;
```

In addition to ordinary Pascal types, Estelle allows an unspecified type, designated by three dots (`...`), as in line *12*. Just as for unspecified constants, this gives rise to a collection of specifications, one for each possible type. The type `...` is usually used to designate something that has no real bearing on the functioning of the protocol,[4] and frequently its use is combined with functions declared to be `primitive` as a way of leaving implementation details out of the specification. Thus a buffer might be declared to be of type `...` and the routines to insert and extract information from the buffer might be declared `primitive`.

4.2 Channels

```
22      channel USAP(user, provider);
23          by user:
24              ConReq;
25              ConResp;
26              DatReq(UserData : UserDataType);
27              DisReq;
28          by provider:
29              ConInd;
30              ConConf;
31              DatInd(UserData : UserDataType);
32              DisInd;
```

The channel declaration, lines *22–32*, gives the name of the channel, `USAP`, together with

[4]It is always somewhat amusing that the user's data have nothing to do with the protocol; yet this is as it should be.

the names associated with the two roles associated with the channel. When two modules are connected through a `USAP` channel, one must assume the role of `user`, and the other the role of `provider`. The `user` can initiate `ConReq`, `ConResp`, `DatReq` and `DisReq` interactions that will be received by the `provider`, and the `provider` can initiate `ConInd`, `ConConf`, `DatInd`, `DisInd` interactions that will be received by the `user`. The `DatReq` and `DatInd` interactions each have a parameter called `UserData`, which is of type `UserDataType`.

The `USAP` channel declaration should be compared with the service diagram in figure 1 as these two contain some of the same information. Both indicate the direction (*i.e.*, which side is the initiator and which the recipient) of each interaction. The channel declaration provides additional information, because it shows which interactions admit parameters. In particular the `DatReq` and `DatInd` interactions each carry `UserData`. However, a service diagram provides information that is not contained in a channel declaration, because it shows causal relationships.

```
34      channel PeerCode(peer, coder);
35          by peer, coder:
36              CR;
37              CC;
38              DT(Seq : SeqType; UserData : UserDataType);
39              AK(Seq : SeqType);
40              DR;
41              DC;
```

The `PeerCode` channel is symmetric, and either role may initiate and receive any of its interactions. We have already discussed the `CR`, `CC`, `DT`, and `DR` PDUs. The other two (`AK` and `DR`) are used to indicate receipt of the `DT` and `DR`, respectively. Note that the `DT` and `AK` PDUs carry sequence numbers.

4.3 Module Headers

```
49      module User systemprocess;
50          ip U : USAP(user);
51      end;
```

Modules are described in two parts, commonly referred to as a *header* and a *body*. The header conveys the information available outside the module, while the body describes actions of the module. lines *49–51* specify the header of the `User` module. The module is declared to be a `systemprocess`. It has only one interaction point, `U`, which may be connected to a `USAP` channel playing the role of the `user`. As no queueing discipline is given for the interaction point, it defaults to `individual queue`, as specified in line *3*. Similar module headers are given for the other systems in the specification: `Cms` (the communications medium) at lines *56–58* and `Abra` at lines *63–66*.

```
53      body UserBody for User;
54      external;
```

The body `UserBody` for the `User` module is given elsewhere (not in this document).

Similarly, lines *60–61* specify that the body for `Cms` is given elsewhere.

4.4 Station Body

```
68      body AbraBody for Abra;
69
70          module Station process;
71              ip USER : USAP(provider);
72                 PEER : PeerCode(peer);
73          end;
```

Simply nesting the definition of the `Station` module within the body for `Abra` makes `Station` a child of `Abra`. It is important to note that unlike Pascal, any variables that may be defined for a module body are not inherited by its children; *i.e.*, the scope of a variable is restricted to the module in which it is declared.

```
75          body StationBody for Station;
```

There may be more than one body for a module header; which body should be used is decided when the module is instantiated (see section 4.7). In this specification, however, there is at most one body given for any module.

To help follow the specification of the Station, a finite state diagram of the Abracadabra Protocol is shown in figure 3.

```
77              state
78                  CLOSED, CRSENT, CRRECV, ESTAB, DRSENT;
```

The Station module has five states. This state is sometimes referred to as the *control state* to distinguish it from the total state of the module, which includes the values of the variables, contents of the queues, *etc.* In the Station, as often happens, the control state corresponds to the progress of the connection as seen by the Station.

```
80              stateset
81                  CRignore = [CRRECV];
82                  CCignore = [CLOSED, CRRECV, DRSENT];
83                  DTignore = [CLOSED, CRSENT, CRRECV, DRSENT];
84                  AKignore = [CLOSED, CRSENT, CRRECV, DRSENT];
85                  DCignore = [CLOSED, CRSENT, CRRECV];
86
87                  ConReqIgnore  = [CRSENT, CRRECV, ESTAB, DRSENT];
88                  ConRespIgnore = [CLOSED, CRSENT, ESTAB, DRSENT];
89                  DatReqIgnore  = [CLOSED, CRSENT, CRRECV, DRSENT];
90                  DisReqIgnore  = [CLOSED, DRSENT];
```

Estelle permits the definition of a set of states, called a `stateset`. This is used as a convenience to collapse several actual transitions into one text that is more succinct and easier to read. For example, a transition that is specified as applying to `AKignore` (see line *414*) may fire if the control state of the module is any of the states listed in line *84*.

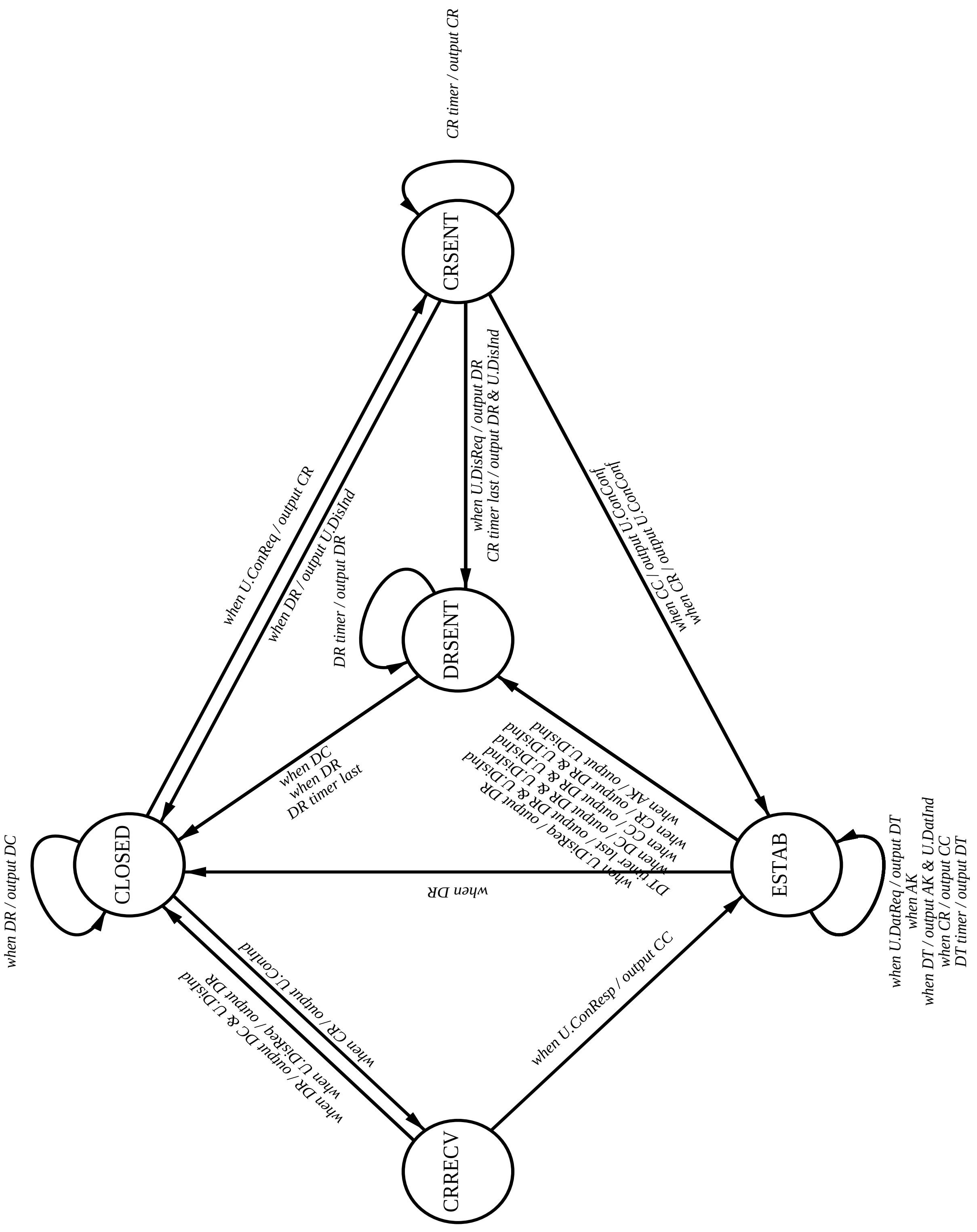
CRSENT
DRSENT
CLOSED
ESTAB
CRRECV
CR timer / output CR
when U.ConReq / output CR
when DR / output U.DisInd
DR timer / output DR
when U.DisReq / output DR
CR timer last / output DR & U.DisInd
when CC / output U.ConConf
when CR / output U.ConConf
when DC
when DR
DR timer last
when AK / output U.DisInd
when CR / output DR & U.DisInd
when CC / output DR & U.DisInd
when DC / output DR & U.DisInd
when U.DisReq / output DR
DT timer last / output DR & U.DisInd
when DR
when DR / output DC
when CR / output U.ConInd
when U.DisReq / output DR
when DR / output DC & U.DisInd
when U.ConResp / output CC
when U.DatReq / output DT
when AK
when DT / output AK & U.DatInd
when CR / output CC
DT timer / output DT

Figure 3: Abracadabra Station Automaton

As used here, each state set is intended to list the various control states in which a specific interaction is to be ignored. This is accomplished by a series of transitions ({ 30 } – { 38 }) that simply consume the interaction to be ignored and make no other changes.

```
        var
            Sending : boolean;
            SendSeq, RecvSeq : SeqType;
            OldSendSeq : SeqType;
            CRRetranRemaining : integer;
            DTRetranRemaining : integer;
            DRRetranRemaining : integer;
            OldData : UserDataType;
            DTorAK : boolean;
```

Variables are defined in the same way they are in Pascal. The meanings of most of the variables above should be obvious when they are used. However, two of these variables are of special interest because they represent a conscious trade off between more control states and values of variables.

Consider `DTorAK`. Once the Station has established a connection, its response to a `CR` depends on whether or not it has previously received a `DT` or an `AK`. If not, it responds by sending a `CC`; if so, it enters the error phase. This is discussed in more detail below when considering transitions { 17 } and { 18 }. The specifier could attempt to deal with this by creating two control states, say `ESTAB1` and `ESTAB2`, that behave like `ESTAB` except for their handling of a `CR`. It was felt that it would be clearer to indicate the special case by a flag than to create an essentially redundant state. Similarly, `ESTAB` could have been split into two states to reflect whether or not it was free to accept a new message to send. Instead, the flag `Sending` was used. These are matters of intent and style, decisions based what information it is important to convey and how best to express it.

```
            procedure InitVar;
                begin
                    Sending := false;
                            ⋮
                end;
```

Procedures and functions in Estelle are used just like procedures and functions in Pascal.

Functions are assumed to be *demonstrably pure* and thus to have no side effects. This is accomplished by allowing them to alter only local variables, to have neither `var` parameters nor parameters containing pointers, and to invoke only pure procedures and functions. Procedures are not presumed to be pure unless they are declared with the keyword `pure`. Unlike pure functions, pure procedures are allowed `var` parameters. Except for procedures and functions that are `primitive`, which have global scope, the scope of a procedure or function is restricted to the module body in which it is defined and does not include the descendants of that module. Procedures and functions may not reference the non-Pascal objects such as modules, interaction points, states, interactions *etc.* introduced into Estelle. The intent of this restriction is to make those operations that affect the underlying finite automaton more visible by ensuring that they remain in the bodies of transitions.

```
116         initialize
117             to CLOSED
118                 begin { 1 }
119                     { Variables are initialized when leaving
120                       CLOSED state, since the protocol module
121                       may cycle through CLOSED repeatedly. }
122                 end;
```

The initialization transition, indicated by the keyword `initialize` in line *116* may initialize the control state and any variables of the module. In the one shown here, only the state is initialized, but a comment informs the reader that variables are initialized when the automaton leaves the `CLOSED` state, since the reader presumably expected to see them initialized here.

The format of the initialization section is the same as that of the transition section, which will be discussed immediately below, but only those portions of a transition that make sense are permitted (namely, a provided-clause and a to-clause).

4.4.1 Connection Phase

```
124         trans
125
126         { *** Connection Phase *** }
127
128             { user requests connection }
129             from CLOSED to CRSENT
130                 when USER.ConReq
131                     begin { 2 }
132                         { initialize module variables whenever
133                           leaving CLOSED }
134                         InitVar;
135                         output PEER.CR;
136                         CRRetranRemaining := N-1;
137                     end;
```

Here we come to the first transition of the Station automaton, indicated by the keyword `trans`. A transition comprises two parts: the enabling condition and the actions. The enabling condition specifies the conditions that must be met before the transition may be fired. Note that even when these conditions have been met, however, the transition may not fire for a number of reasons, including parent/child priority and non-determinism. The actions indicate what will happen as a result of firing the transition.

Line *129* indicates that the transition may take place only when the control state is `CLOSED` and that after the transition fires, the control state will be `CRSENT`. The transition may fire only when the head of the queue associated with the `USER` interaction point is a

`ConReq`. Except for scoping rules as discussed below (section 4.4.2) and the nesting rules (section 4.4.2), the order of the clauses — from, to, when, *etc.* — is immaterial to the meaning of the specification.

The actions of this transition are given in lines *131–137*. First, since the control state will leave `CLOSED`, the procedure `InitVar` is invoked to initialize the variables of the module as was promised in the comment in lines *119–121*. Next a `CR` destined for the peer Station module is output through the `PEER` interaction point. The retransmission timer for `CR`s is initialized to `N-1`, because at most N transmissions of a `CR` are allowed, and one has just been made.

```
139                 { other user accepted connection }
140                 from CRSENT to ESTAB
141                     when PEER.CC
142                         begin { 3 }
143                             output USER.ConConf;
144                             CRRetranRemaining := -1;
145                         end;
```

The system that sends the `CR` is usually referred to as the *initiator*, and the system that receives it is usually referred to as the *responder*. Transition { 2 } begins the attempt to establish a connection from the initiator's side. Transition { 3 } completes the connection establishment on the initiator's side, if all went well: it informs the user by outputting a `ConConf` through the `USER`, and it cancels the possible retransmission of `CR`s by setting `CRRetranRemaining` to `-1`. The effect of this is reflected in transitions { 7 } and { 8 }.

```
147                 { colliding CRs }
148                 from CRSENT to ESTAB
149                     when PEER.CR
150                         begin { 4 }
151                             output USER.ConConf;
152                             CRRetranRemaining := -1;
153                         end;
```

Transition { 4 } is to take care of the case when the two users decide to try to open a connection at the same time.[5] Instead of receiving a `CC`, the initiator receives a `CR`, which is treated exactly as though it were a `CC`. This corresponds to the service diagram shown in figure 4.

```
155                 { other user rejected connection }
156                 from CRSENT to CLOSED
157                     when PEER.DR
158                         begin { 5 }
159                             output USER.DisInd;
160                             CRRetranRemaining := -1;
161                         end;
```

Of course it is always possible for the responding user to refuse the connection. If so,

[5]Apologies to Albert Einstein.

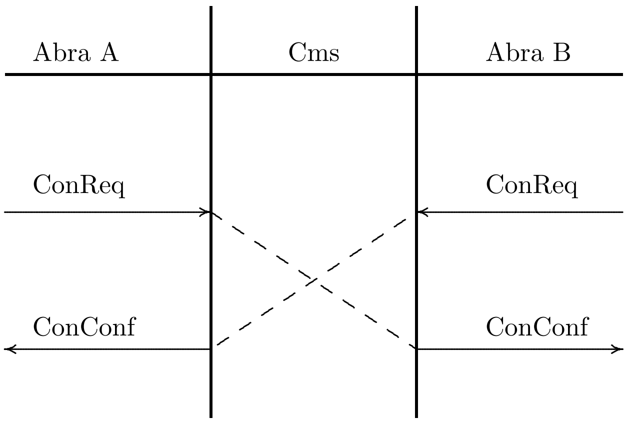

Figure 4: Simultaneous Connection Request Service Diagram

transition { 5 } handles the case: the initiating user is sent a `DisInd` and the CR retransmission timer is effectively turned off.

```
163              { sender requests disconnection }
164              from CRSENT to DRSENT
165                  when USER.DisReq
166                      begin { 6 }
167                          output PEER.DR;
168                          CRRetranRemaining := -1;
169                          DRRetranRemaining := N-1;
170                      end;
```

The initiating user may become impatient and decide to cancel the connection before being informed that it was open. When the Station is in the `CRSENT` state, this `DisReq` causes it to send a `DR` to the other Station, to indicate that it should close the connection that was previously requested, shut off possible retransmissions of the now irrelevant `CR` and set up for possible retransmissions of the `DR`.

```
172              { retransmission timer for CR fires }
173              from CRSENT to same
174                  provided CRRetranRemaining > 0
175                      delay (P)
176                          begin { 7 }
177                              CRRetranRemaining :=
178                                  CRRetranRemaining - 1;
179                              output PEER.CR;
180                          end;
```

Transition { 7 } has no when-clause; its availability to fire does not depend on the receipt of any input, and so it is called a *spontaneous transition*. Such transitions are available to fire whenever the enabling conditions of the transition are true. In many cases, however, it is desirable to associate timing parameters with spontaneous transitions. These are indicated by the presence of a delay-clause, such as that in line *175*. The delay-clause may have two arguments t_0 and t_1. The corresponding condition may be regarded as true provided the other parts of the enabling conditions have been true continuously for at least t_0 time units; it must be regarded as true if the other parts of the enabling conditions remain true for t_1 time units. The difference is this: between times t_0 and t_1 the choice is not determined by the specification: the transition may fire or not; the decision is left to the implementor. After time t_1, the transition must fire unless some other transition is also available, in which case the usual Estelle rules of non-determinism apply. In other words, if it is the only one available transition, it must fire. One possible use for such a scheme is when a protocol uses "piggy-backed" acknowledgments, where acknowledgments accompany data, if any are available. An acknowledgment may be required to wait t_0 time to allow data to accumulate, but if none have appeared by time t_1, then the acknowledgment will be sent without data.

Two special conventions exist for delay-clauses. If t_1 may be arbitrarily long, meaning "forever", it may be indicated as an asterisk (`*`). A transition that is marked `delay(0,*)` may fire at any time that the other conditions are met, but it need not fire. One use of this might be to indicate that a connection that becomes unused may remain open or be closed, with the decision left to the implementor. The other special convention is used where t_0 and t_1 are the same. The form `delay(t)` is used to indicate that `t`$= t_0 = t_1$.

It is not possible to use a delay-clause in a transition with a when-clause.

The conditions in lines *173–175* together with the actions in lines *176–180* and the initialization in line *136* allow the Station to retransmit at most $N-1$ `CR`s, with a delay of P time units (presumably seconds, according to line *4*) between them.

```
182                 { terminate retransmission of CR }
183                 from CRSENT to DRSENT
184                     provided CRRetranRemaining = 0
185                         delay (P) { allow time for last CR }
186                             begin { 8 }
187                                 { enter error phase }
188                                 output USER.DisInd;
189                                 output PEER.DR;
190                                 CRRetranRemaining := -1;
191                                 DRRetranRemaining := N-1;
192                             end;
```

After N attempts to send a `CR` with no appropriate reply, the initiating Station gives up: it sends its user a `DisInd` and tries to tear down the connection, in case the responding Station actually thought one existed. It is important to realize that although the initiating Station received no `CC`, the responding Station may have sent several and may believe that a connection has been established.

The obverse situation of the responder is handled by transitions { 9 } – { 12 }. Note that `CC`s are not retransmitted. The initiator will resend `CR`s if necessary, and each one received will result in a `CC`'s being sent in reply (line *208* and line *284*).

4.4.2 Data Transfer Phase

```
226         { *** Data Transfer Phase *** }
227
228             { send data in DT PDU }
229             from ESTAB to same
230                 when USER.DatReq
231                     provided not Sending
232                         begin { 13 }
233                             OldData := UserData;
234                             output PEER.DT(SendSeq, OldData);
235                             OldSendSeq := SendSeq;
236                             SendSeq := (SendSeq + 1) mod 2;
237                             Sending := true;
238                             { turn on retransmission timer }
239                             DTRetranRemaining := N-1;
240                         end;
```

Transition { 13 } is responsible for sending data. Line *229* indicates that after it fires, the control state will remain `ESTAB`. It cannot fire if `Sending` is `true`. However, the user may nevertheless place data to be sent into the queue associated with the `USER` interaction point during that time. When `Sending` eventually becomes `true`, the data will be sent.

The identifier `UserData` is defined in line *26* as the parameter of the `DatReq` interaction. The presence of `DatReq` in line *230* opens a region that extends through the end of the transition in which the identifier `UserData` is defined. Its value is the value that was associated with it when the interaction was sent by the user. It must be copied (line *233*) because the value will be needed after the transition has executed if it is necessary to retransmit the PDU. Similarly, `OldSendSeq` will be used if it is necessary to retransmit the PDU. `SendSeq` is then updated to the other bit, to be used for comparison with the acknowledgment.

```
242             { receive ack with correct sequence number in AK PDU }
243             from ESTAB to same
244                 when PEER.AK
245                     provided Seq = SendSeq
246                         begin { 14 }
247                             Sending := false;
248                             { turn off retransmission timer }
249                             DTRetranRemaining := -1;
250                             DTorAK := true;
251                         end;
```

In line *245*, the parameter of the `AK` interaction, `Seq`, is compared with `SendSeq`. Because of the scoping rule discussed above, this is one of those cases where the order of the enabling clauses is important: if the provided-clause preceded the when-clause, `Seq` would be undefined. The convention for acknowledgment used here is a common one; the peer Station acknowledges receipt of a PDU by sending the sequence number of the next PDU it expects to receive.

The data have successfully been sent, so the `Sending` flag is turned off. This allows the next PDU to be sent as soon as there is one. The DT retransmit counter is set to `-1` so that no more retransmissions of the PDU may be made. As discussed in section 4.4, `DTorAK` is set to `true` to control the reaction of the Station to receiving another `CR`. The actual effect of this may be observed in transitions { 17 } and { 18 }.

```
253              { receive acknowledgement with incorrect sequence number }
254              from ESTAB to DRSENT
255                  when PEER.AK
256                      provided Seq <> SendSeq
257                          begin { 15 }
258                              { enter error phase }
259                              output USER.DisInd;
260                              output PEER.DR;
261                              DTorAK := true;
262                              DTRetranRemaining := -1;
263                              DRRetranRemaining := N-1;
264                          end;
```

Transition { 15 } takes care of the case where the `AK` has the wrong sequence number by entering the error phase. Note that there are two outputs made in this transition, one to the user to indicate that the connection will be broken and the other to the peer to break the connection. Since an `AK` has been received, `DTorAK` is set to `true`. Note, however, that the control state after this transition fires will be `DRSENT`, and the only transitions that test `DTorAK` are from `ESTAB`. Figure 3 shows that the only transition from `DRSENT` to a different control state is a transition to `CLOSED`, and whenever the control state leaves `CLOSED`, `DTorAK` is reset to `false` in `InitVar`. Thus line *261* could be removed from the specification without changing its behavior.[6]

```
266              { receive data in DT PDU }
267              from ESTAB to same
268                  when PEER.DT
269                      begin { 16 }
270                          if Seq = RecvSeq then
271                              begin
272                                  output USER.DatInd(UserData);
273                                  RecvSeq := (RecvSeq + 1) mod 2;
274                              end;
275                          { send AK with next expected sequence number }
276                          output PEER.AK(RecvSeq);
277                          DTorAK := true;
278                      end;
```

Transition { 16 } handles receipt of data. If the sequence number in the PDU is the

[6]But should it be?

expected one, then the data are presented to the user because this is the first time they have been received and the sequence number is advanced to the next one. Even if the sequence number is not the expected one, the PDU is acknowledged in case previous acknowledgments were lost. As in transition { 14 }, `DTorAK` is set to `true`.

```
280        from ESTAB to same
281            when PEER.CR
282                provided not DTorAK
283                    begin { 17 }
284                        output PEER.CC;
285                    end;
286
287        from ESTAB to DRSENT
288            when PEER.CR
289                provided DTorAK
290                    begin { 18 }
291                        { enter error phase }
                                ⋮
296                    end;
```

Transitions { 17 } and { 18 } select the actions to perform if a `CR` is received. Presumably if a `DT` or an `AK` has already been received, then the peer Station must be in the `ESTAB` state, so it should not be sending `CR`s any more. Thus a `CR` represents a protocol error. On the other hand, if neither a `DT` nor an `AK` has yet been received, then the peer Station may retransmit several `CR`s until it receives a `CC` in response.

```
298        from ESTAB to DRSENT
299            when PEER.CC
300                begin { 19 }
301                    { enter error phase }
                            ⋮
306                end;
307            when PEER.DC
308                begin { 20 }
309                    { enter error phase }
                            ⋮
314                end;
```

Here use is made of the Estelle's transition nesting facility, which allows the specifier to telescope transitions.

Repeating the keyword `when` in line *307* causes the enabling clauses preceding the `when` in line *299* to apply to transition { 20 } in addition to the effect they have on transition { 19 }. Thus transition { 20 }, like transition { 19 } may fire only if the control state is `ESTAB`, and if it fires, the control state will become `DRSENT`. This facility is not unique to the when-clause. Any of the enabling clauses may be handled in this way. If any clause in the series of enabling clauses of a transition (say T_0) is repeated immediately following the body of the transition (thus indicating the enabling clauses of the next transition, T_1), then the clauses of T_0 before that repeated clause apply to the successor transition T_1. Making use of this facility is one way to write well-structured specifications. In the case only of a sequence of nested provided-clauses, a special form, `provided otherwise`, may be used at the end of the sequence to represent the negation of the conditions of the other provided-clauses. For example, in

```
from S1 provided C1        to T1 begin - - - end;
        provided C2        to T2 begin - - - end;
        provided C3        to T3 begin - - - end;
        provided otherwise to T4 begin - - - end
```

`provided otherwise` means `(not (C1 or C2 or C3))`.

Using the keyword `trans` between two transitions prevents nesting; the presence of `trans` prevents the enabling clauses of the transition before it from applying to those of the transition after it.

Like transition { 18 }, transitions { 19 } and { 20 } represent reactions to protocol errors.

The handling of retransmissions of `DT` PDUs in transitions { 21 } and { 22 } is very similar to the handling of retransmissions of `CR` PDUs in transitions { 7 } and { 8 }.

4.4.3 Disconnection Phase

```
{ *** Disconnection Phase *** }

    { receive disconnect request from user }
    from ESTAB to DRSENT
        when USER.DisReq
            begin { 23 }
                output PEER.DR;
                DTRetranRemaining := -1;
                DRRetranRemaining := N-1;
            end;
```

The user indicates that the connection should be broken by issuing a `DisReq`. This is handled in transition { 23 }. It has the effect of an *abrupt* disconnection: because the `DT` retransmission counter is reset to `-1`, if the last message sent on behalf of the user was not received, it will be lost, and the user will not be informed of this. This is unique to the last message; for other messages, transition { 22 } presents the user with a `DisInd` if it was unable to deliver the message.

```
    { receive DC }
    from DRSENT to CLOSED
        when PEER.DC
            begin { 24 }
                DRRetranRemaining := -1;
            end;
```

If all goes as planned, the DR sent on line *344* will be cause a DC to be sent in reply. As disconnection is not a confirmed service, the user is not informed that the disconnection took place, so the Station simply resets the DR retransmission counter and returns to the CLOSED state. An argument similar to that regarding line *261* could be made here as well; line *353* is unnecessary.

```
356          { receive DR }
357          from DRSENT to CLOSED
358              when PEER.DR
359                  begin { 25 }
360                      DRRetranRemaining := -1;
361                  end;
```

If a DR is received instead of a DC, presumably both users decided to end the connection at the same time. The Station treats the DR just as it did the DC in transition { 24 }.

```
363          { receive DR }
364          from ESTAB to CLOSED
365              when PEER.DR
366                  begin { 26 }
367                      output USER.DisInd;
368                      output PEER.DC;
369                      DTRetranRemaining := -1;
370                  end;
```

On the other side, when a DR arrives, the user is informed via a DisInd, the peer Station is sent a DC, and the connection is abruptly broken.

```
372          { reply to retransmitted DR }
373          from CLOSED to same
374              when PEER.DR
375                  begin { 27 }
376                      output PEER.DC;
377                  end;
```

If the DC was not received by the peer, it will retransmit the DR after this Station has entered the CLOSED state. Transition { 27 } takes care of replying to such a DR.

Transitions { 28 } and { 29 } take care of retransmitting the DR if necessary. The are similar to transitions { 7 } and { 8 } or transitions { 21 } and { 22 }, except that if the DR retransmission counter reaches zero there is no point in entering the disconnection phase again, so the Station merely goes to the CLOSED state.

Transitions { 30 } to { 38 } are to take care of possible interactions that should either not arise or should be ignored. The arrival of an interaction for which no transition is applicable is sometimes called an *unspecified reception.* The presence of such an unexpected interaction at the head of a queue causes the queue to remain blocked until something (*e.g.*, the arrival of an interaction in another queue or the firing of a spontaneous transition) changes the state of the automaton. Unspecified reception often is the result of a service violation. For example, if the user could be relied on not to make a DisReq when

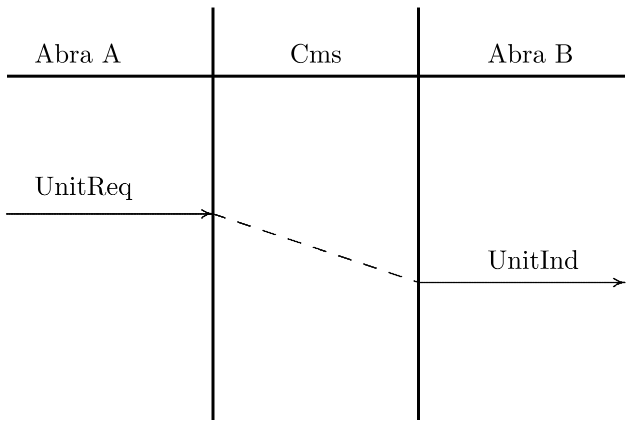

Figure 5: Communications Medium Service Diagram

there is no connection, then transition { 38 } would be unnecessary. Protocol specifiers take differing attitudes toward this situation: some would leave out those transitions that should not arise if the other modules behave as expected; others, recognizing the foibles of human beings and their systems, explicitly deal with these unspecified receptions. This completes the specification of the Station.

4.5 TransCode

It is now time to confess that we have purposely hidden the actual structure to be used in this specification. The ISO OSI Reference Model [ISO, IS7498, clause 5.3] defines communication between peer-entities in such a way that one would not expect two peer Station modules to exchange PDUs directly. Entities at layer N make use of the services provided by layer $(N-1)$. The actual communication in this case is thus between the `Abra` module and the `Cms` communications medium module. The PDUs of the Station are to be wrapped in a UnitReq which is delivered as a UnitInd. The service of the Communications Medium is shown in figure 5.

We introduce a module named `TransCode` to deal with this. Like the `Station`, it is a child of `Abra`. Its functions are to accept a PDU from the Station and wrap it into a UnitReq that it presents to the Cms, and *vice-versa*, to accept a UnitInd from the Cms and unwrap it and present it to the Station as an appropriate PDU. This behavior allows each Station module to maintain the fiction that it communicates directly with its peer.

```
449        module TransCode process;
450            ip Up   : PeerCode(coder);
451               Down : MSAP(user);
452        end;
```

The TransCode module has two interaction points, `Up` and `Down`. `Up` will be connected to the Station while `Down` will ultimately be connected to the communications medium. There are a dozen transitions, six to deal with encoding the six PDU types (see line *14*) into `UnitReq` PDUs and six to deal with decoding them.

The TransCode module is so simple that it has no control state, so the transitions have neither a from- nor a to-clause. Alternatively, one could supply a single state, say S, initialize the control state to S, and add `from S to S` to each transition. This seems artificial and unnecessary, so Estelle does not insist on it. Other than that, there is nothing new about Estelle in the specification of the TransCode module body, so we do not deal with it further.

The actual structure of the system is that found in figure 6.

4.6 Binding

This new structure does add a level of complexity. The inner structure of the `Abra` module is not supposed to be known to the rest of the specification. Thus the `User` module will be connected to the `Abra` module, but the interactions that the `User` generates should be handled by the `Station` module. This is cared for by an operation in Estelle called *attach*. After a parent module attaches one of its external interaction points to the external interaction point of a child, the interactions that arrive at the parent's interaction point are transparently[7] passed to the child's interaction point. If the attachment is broken by a *detach* operation, any unprocessed interactions that were passed to the child as a consequence of the attach operation are returned to the parent.[8] In the other direction, an interaction that is output through an interaction point that has been attached will transparently be forwarded to the appropriate recipient.

4.7 Main Bodies

```
        { main body for AbraBody }
        modvar
            S : Station;
            XC : TransCode;
        initialize
            begin
                { instantiate the modules }
                init S with StationBody;
                init XC with TransCodeBody;

                { make connections }
                attach USER to S.USER;
                connect S.PEER to XC.Up;
                attach MEDIUM to XC.Down;
            end;
    end; { AbraBody }
```

[7]No pun intended.

[8]In some cases, there may be interactions in a queue that resulted from connection operations, and these will not be returned to the parent, nor will interactions in a common queue that were not received through the detached interaction point be delivered to the parent.

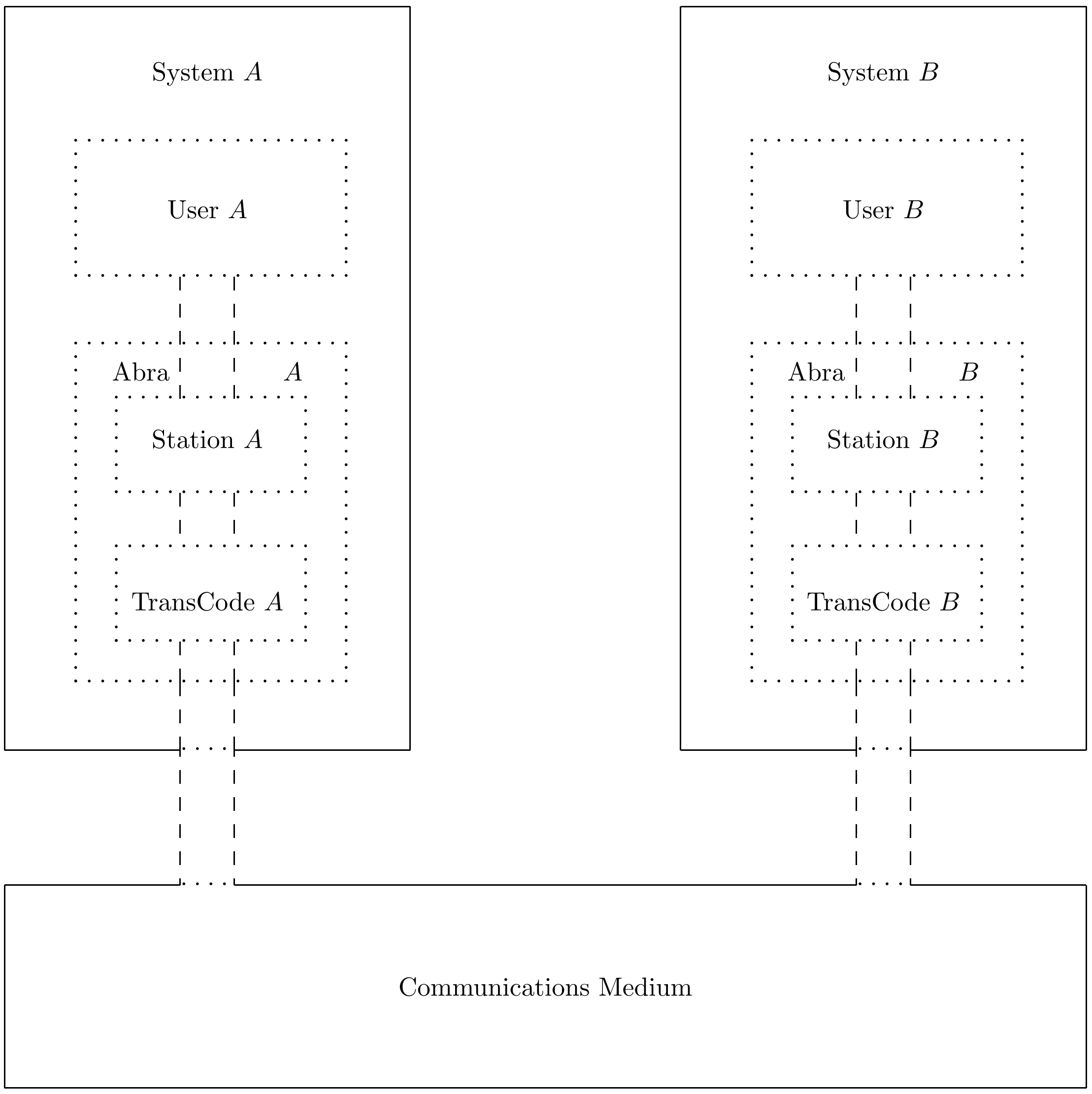

Figure 6: Actual Abracadabra System

```
572
573      { main body for Specification Abracadabra }
574          modvar
575              A, B   : Abra;
576              UA, UB : User;
577              CM     : Cms;
578
579          initialize
580              provided (N > 0) and (P > 0)
581                  begin { 1 }
582                      init UA  with UserBody;
583                      init UB  with UserBody;
584                      init A   with AbraBody;
585                      init B   with AbraBody;
586                      init CM  with CmsBody;
587                      connect UA.U to A.USER;
588                      connect UB.U to B.USER;
589                      connect A.MEDIUM to CM.CMA;
590                      connect B.MEDIUM to CM.CMB;
591                  end;
592
593    end. { Specification Abracadabra }
```

The main body of each module (and of the specification itself) occurs at the end of the body (or specification). Lines *557–559* and lines *574–577* declare module variables to be used to instantiate the modules necessary to make the system. The behavior of the specification begins with the initialization of the main module, which as noted above (section 4.1) may only fire if both N and P are positive. Lines *582–586* instantiate the module variables with their bodies. A module may be instantiated with any body that is defined for it. As a module is instantiated, it is initialized. This may cause other modules to be instantiated, as in the case of line *584*, which causes the initialization beginning at line *560* to take place. In turn, this causes the initialization at line *116* to take place. Thus at the end of the initialization transition of the main body (lines *581–591*), the entire system will exist. In this specification there are not transitions outside the initialization sections that alter the structure in any way, so it is static. Other specifications create and delete module instances (*e.g.*, when opening or closing a connection) and thus have a dynamic structure.

When an `Abra` module is initialized, creates its two children modules, a Station module `S` and a TransCode module `XC`, and in line *568* it connects the `PEER` interaction point of the Station to the `Up` interaction point of the TransCode. And it attaches its own `USER` interaction point to the `USER` interaction point of the Station `S`. Thus when the `User` initiates an interaction, it will appear in the queue of the Station module, not the queue of the Abra module, even though in line *587* and line *588* the User modules are connected to the Abra modules, and when the Station module outputs an interaction, it will end up in the queue of the User module. Similarly, the `Down` interaction point of the TransCode module is attached to the `MEDIUM` interaction point of the Abra module, so that `UnitReq` and `UnitInd` interactions will be occur between TransCode and Medium modules without intervention of the Abra modules.

4.8 Other Features

Naturally not all features of Estelle will appear in every specification. Some of those that do not appear in this specification are discussed briefly below.

- disconnect — to undo the effect of connect. There are no consequences to the contents of the queue of the interaction point that is disconnected.
- detach — to undo the effect if attach. Any interactions in the queue of the detached interaction point that were placed there as a consequence of the parent's prior attach are returned to the parent by the detach.
- exported variables — to allow a module to share specified variables with its parent. Siblings may not share variables with each other.
- release and terminate — two ways of deleting a module instance. A module may delete its children but not itself nor its siblings. Release and termination are recursive; *i.e.*, a module that is being released first releases its children. The interaction points of a module being released are disconnected or detached, as appropriate. As a consequence, queues of the parent will receive unprocessed interactions from detached interaction points, as discussed above (section 4.6). The contents of queues of a module being terminated are simply discarded.
- all, exist, and forone — techniques for dealing with ordinal types or sets of modules. The `all` construct permits statements to be iterated over each member of a finite ordinal type or over each module in a domain. The `exist` construct tests if there is a value or module satisfying a given criterion. The `forone` construct executes a (compound) statement for a value or module that satisfies a given criterion.
- arrays of interaction points — to handle multiple users or providers, each capable of the same kinds of interactions.
- priority — to control which transitions may fire when more than one is enabled.

The interested reader may wish to read other tutorials ([Dembinski and Budkowski, 1989], [Linn, 1987], [Turner, 1993, chapter 2], [ISO/IEC JTC 1/SC 21, N5710]) for further insights into Estelle and to the Estelle International Standard itself for authoritative answers to questions.

5 Testing

Whenever complex software is considered — and communications software is certainly complex — questions of correctness must arise. One way of increasing reliability of software is to specify it formally. For international standards specifying communications protocols, this is especially important, as the implementations are frequently provided by vendors from around the world who are not in direct contact with one another but whose software products must work together. Given the sums of money involved and the consequences of failure, it is certain that eventually questions of correctness will have to be considered in courts of law.

There is a large literature concerning conformance of implementations to their specifications. Specification techniques based on extended finite automata admit generation of test suites via state space exploration techniques, and Estelle is no exception (see *e.g.*, [Favreau and Linn, 1987]). For general overviews of available testing methods, see [Sidhu, 1990], [Miller, 1990] and [Hogrefe, 1991].

In this paper, however, we shall focus on a narrower question: how can one begin to test a specification? This should take place long before the more formal tests and the conformance tests are done. Under ideal circumstances this testing goes along with developing the protocol. We call this *early testing* to differentiate it from the more formal tests generally performed later in the development cycle.

Even though the Abracadabra protocol has been presented here as a completed protocol, we can nevertheless begin to do some testing with it, the kind of testing that is usually done during and immediately after designing the protocol. We shall discover that it has some defects that are not well known. Much of this analysis comes from [Blumer and Parker, 1990].

The first test of an Estelle specification is simply syntactic. Estelle compilers obviously must spot undeclared identifiers and the usual range of errors detected by compilers. They can warn of variables that are declared but never referenced, *etc.*

One of the next things to check for is unspecified receptions (discussed on section 4.4.3). This is one of the earliest tests partly because it is one of the easiest: one need merely produce a table of the transitions applicable to each state and input pair. Many tools produce this information as an option (see *e.g.*, the `xref` program or the `_fsm` structure produced by the Phoenix software [Blumer, 1986], or the information file `fsm.inf` produced by the NBS (now NIST) compiler [NBS, 1987].)

Such a table is found in table 1. This shows the transition number for each possible input and control state of the finite automaton that underlies the Station module. As can be seen, there is indeed an unspecified reception in the Abracadabra protocol. If a `CR` PDU is received in the state `DRSENT`, there is no transition to handle it.

It is reasonable to ask if this situation may arise. It does indeed, and in many different ways. Here is the way shown by Blumer and Parker: Assume User B has attempted to open a connection and that all N transmissions of a `CR` have failed, so transition $\{ 8 \}$ fires, and the control state of Station B is `DRSENT`. Now User A attempts to open a connection, so Station A fires transition $\{ 2 \}$ and enters the control state `CRSENT`. Station B receives

Transition Number	CLOSED	CRSENT	CRRECV	ESTAB	DRSENT
ConReq	2	35	35	35	35
ConResp	36	36	10	36	36
DatReq	37	37	37	13	37
DisReq	38	6	11	23	38
CR	9	4	30	17, 18	$\square$
CC	31	3	31	19	31
DT	32	32	32	16	32
AK	33	33	33	14, 15	33
DR	27	5	12	26	25
DC	34	34	34	20	24

Table 1: Transition table for Station

the `CR` from Station A but has no transition to deal with it. Station B will retransmit its `DR`s and Station A will retransmit its `CR`s. Ultimately, Station B will fire transition $\{\,29\,\}$ and enter the `CLOSED` state. It processes the `CR` in its queue, outputting a `ConInd` to User B and entering the `CRRECV` state. One version of this sequence of events is shown in figure 7.

From this point, there are several possible scenarios, depending on events and their order. Although investigating these possibilities is actually quite interesting, all we need note here is that this does represent an error in the protocol specification, one that can be expected to occur. Blumer and Parker suggest enlarging the set `CRignore` (line *81*) to include the state `DRSENT`. Transition $\{\,30\,\}$ will then discard any `CR` that arrives in the state `CRignore`.

One of the next things to check for is that each of the transitions can be executed. This entails checking that each state is reachable from an initial state and that there are no incompatible requirements to executing any transition. For example, it might happen that every possible way of reaching state S somehow entails setting a variable v to a negative value, but that to execute the transition in question, v must be positive. In this case, the transition cannot ever fire.

Technically, like many other things we should like to verify, this is unsolvable (*i.e.*, equivalent to solving the halting problem for Turing Machines). In other words, this cannot be checked by any automatic procedure; rather each specification requires an individual proof that each transition may be executed. Such a proof may become extremely complicated, however it may usually be done by demonstrating a series of inputs that will allow the transition to fire.

In providing such a demonstration, we usually rely on two things: the problem is easily solvable for the underlying finite automaton using well-known techniques, and automatic tools can provide irreplaceable help, especially for complex protocols. Thus, in the case of the Abracadabra protocol we can indeed verify that all transitions are executable.

This question may actually be asked in two different ways: is it possible to execute the transition at all? and is it possible to execute the transition if the modules interacting

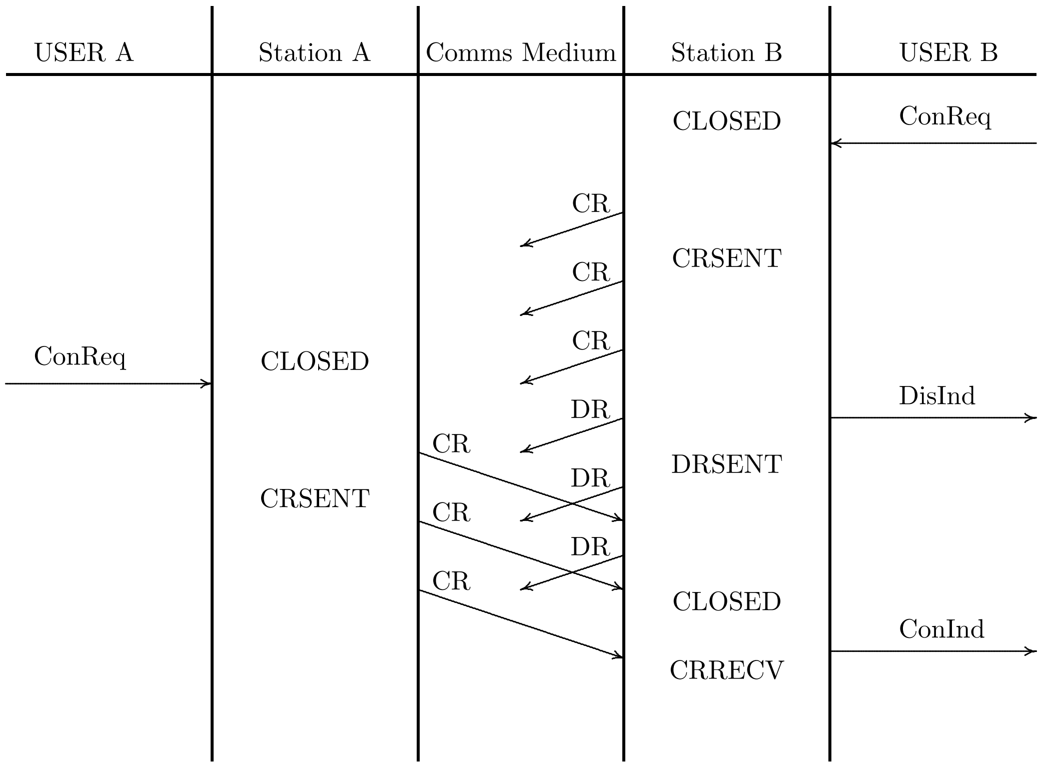

Figure 7: Unspecified Reception

with this module obey all the constraints placed on them? This latter question may be more difficult to answer, because the inputs used to show that the transition may fire must now satisfy additional constraints. On the other hand, there are transitions, *e.g.* those dealing with errors, that are not expected to fire under normal conditions, and thus this question is not one that should be asked about those transitions.

The Abracadabra protocol depends on parameters. These should be checked for reasonable ranges. The amount of time P that must elapse before retransmission is often critical to the functioning of a protocol like Abracadabra, so it is one of the things that should be checked. If P is set to too high a value, throughput may suffer, but if it is set to too low a value, then unnecessary retransmissions will take place. Indeed, if the product $N \times P$ is too small, no communication can take place at all. Clearly P has to be set to a value greater than the round-trip across the communications medium. However, the medium was assumed to be able to delay messages, so there is no value that actually represents the maximum round-trip. Blumer and Parker show that this can lead to problems.

A delayed `CR` can cause a disconnection as shown in figure 8. Here both Users start to open a connection at the same time. Both Station A and Station B send `CR`s. The `CR` from Station B is sufficiently delayed that Station A retransmits its `CR`. Each of these arrives at Station B. The first causes B to enter the ESTAB state, and thus the second causes a `CC` to be sent (transition $\{17\}$). Meanwhile, the arrival of the delayed `CR` causes Station A to enter the ESTAB state, so when the `CC` arrives, transition $\{19\}$ fires, breaking the connection.

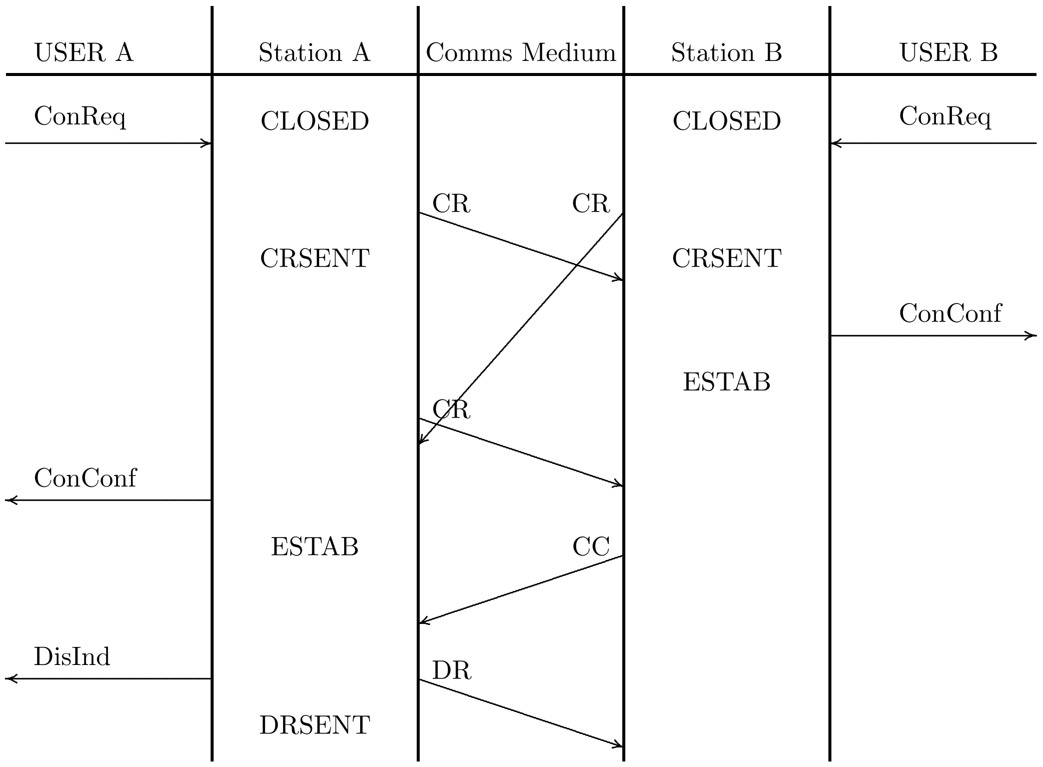

Figure 8: Delayed CR

A similar problem can be demonstrated when an acknowledgment is delayed. Again the connection is broken.

Another Abracadabra problem, pointed out in [de Saqui-Sannes and Courtiat, 1990], occurs when one Station acts only as a receiver and all DRs sent to it are lost; it will never realize that the connection has closed.

Other tests that help ensure that a protocol under development is performing as desired include simulation. There are several Estelle simulation systems listed in the next section. One that is particularly unusual is Grope [New and Amer, 1991], which shows the execution of the protocol in animation.

Using paths through the underlying automaton, one may work backwards from the protocol specification and derive a service specification that can be compared with the original service specification for correctness. Such a study is reported in [Sidhu and Blumer, 1986].

6 Tools

As mentioned above, testing cannot take place without tools. Indeed the available tools affect how we investigate protocols.[9] Besides the tools mentioned above, there are several tools available for Estelle that have been reported on in the literature, including [Bull S.A., 1989], [Bull S.A. and Marben S.A., 1989], [Phalippou and Groz, 1989], [Vuong and Chan,

[9]It is said that to someone with a hammer everything appears to be a nail.

1989], [Sijelmassi and Strausser, 1991], and [de Saqui-Sannes and Courtiat, 1990].

In addition to commenting on the usefulness of Estelle and the tools that support it, [Diaz *et al.*, 1990] and [Chamberlain and Amer, 1990] also mention several systems and protocols in various areas that have been specified in Estelle.

7 Conclusion

This paper has presented a tutorial in Estelle, including a detailed example of its use based on the Abracadabra protocol. In addition, some of the initial tests to be performed on a protocol were discussed. When applied to the Abracadabra protocol these exposed some problems with it.

At present, there is growing use of Estelle. Among the most recent examples of its use (in progress as this is written) is a proposed informative annex to the Transaction Processing standard [ISO, DIS10026].

References

Bartlett, K.A., Scantlebury, R. A., and Wilkinson, P. T. [1969]. A note on reliable full-duplex transmission over half-duplex links. *CACM,* **12**, 260–261.

Blumer, Tom P. [1986]. *Estelle Development System, Users Manual.* Phoenix Technologies Ltd. Cambridge, Massachusetts.

Blumer, Tom P. and Parker, Jeff [1990]. Testing of Estelle protocols via automatic implementation. Technical Report BCCS-90-05, Boston College.

Blumer, Tom P. and Tenney, Richard L. [1982]. An automated formal specification technique for protocols. *Computer Networks,* **6**, 201–217.

Bochmann, Gregor V. and Sunshine, Carl [1980]. Formal methods in communication protocol design. *IEEE Transactions on Communications,* **COM-28**, 624–631.

Budkowski, Stanislaw and Dembinski, Piotr [1987]. An introduction to Estelle: A specification language for distributed systems. *Computer Networks and ISDN Systems,* **14** (1), 3–23.

Bull S.A. [1989]. *Estelle Simulator/Debugger User Reference Manual.* Les Clayes s/s Bois.

Bull S.A. and Marben S.A. [1989]. *Estelle-to-C Compiler User Reference Manual.* Les Clayes s/s Bois.

Chamberlain, Samuel C. and Amer, Paul D. [1990]. A new user's experiences and impressions with Estelle. In [Vuong, 1990] pp. 471–474.

de Saqui-Sannes, P. and Courtiat, J.-P. [1990]. From the simulation to the verification of Estelle* specifications. In [Vuong, 1990] pp. 393–407.

Dembinski, Piotr and Budkowski, Stanislaw [1989]. Specification language Estelle. In *The Formal Description Technique Estelle—Results of the ESPRIT/SEDOS Project* (Diaz, Michel, Ansart, Jean-Pierre, Courtiat, Jean-Pierre, Azema, Pierre, and Chari, Vijaya, eds.) pp. 35–75. Elsevier Science Publishers B.V. (North Holland) Amsterdam. This is an extension of [Budkowski and Dembinski, 1987].

Diaz, Michel, Dufau, Jean, and Groz, Roland [1990]. Experiences using Estelle within Sedos Estelle Demonstrator. In [Vuong, 1990].

Favreau, Jean-Philippe and Linn, Jr., Richard J. [1987]. Automatic generation of test scenario skeletons from protocol specifications written in Estelle. In *Protocol Specificiation, Testing, and Verification, VI* (Sarikaya, Behçet and Bochmann, Gregor V., eds.). Amsterdam: Elsevier Science Publishers B.V. (North Holland).

Hogrefe, Dieter [1991]. Conformance testing based on formal methods. In [Quemada *et al.*, 1991] pp. 207–222.

ISO, IS7185. *Programming Language — PASCAL*. Geneva: International Organization for Standardization, 1982.

ISO, IS7498. *Information processing systems — Open systems interconnection — Basic Reference Model*. Geneva: International Organization for Standardization, 1984.

ISO, IS8073. *Information Processing Systems — Open Systems Interconnection — Connection Oriented Transport Protocol Definition*. International Organization for Standardization, 1986. Also issued as CCITT X.224.

ISO, IS8807. *Information processing systems — Open systems interconnection — LOTOS, an FDT based on the Temporal Ordering of Observation Behaviour*. Geneva: International Organization for Standardization, 1989.

ISO, IS9074. *Information processing systems — Open systems interconnection — Estelle — A formal description technique based on an extended state transition model*. Geneva: International Organization for Standardization, 1989.

ISO, DIS10026. *Information processing systems — Open systems interconnection — Distributed Transaction Processing*. Geneva: International Organization for Standardization, 1991.

ISO, TR10167. *Information processing systems — Open systems interconnection — Guidelines for the Application of Estelle, LOTOS, and SDL*. Geneva: International Organization for Standardization, 1991.

ISO/IEC JTC 1/SC 21, N5710. *Revised Text of ISO 9074: 1989/PDAM 1, Information processing systems — Open systems interconnection — Estelle — A formal description technique based on an extended state transition model — Proposed Draft Amendment 1: Estelle Tutorial*. Geneva: International Organization for Standardization, 1991.

Linn, Jr., Richard J. [1987]. Tutorial on the features and facilities of Estelle. ICST report, National Bureau of Standards.

Logrippo, Luigi, Probert, Robert L., and Ural, Hasan, eds. [1990]. *Protocol Specificiation, Testing, and Verification, X.* Amsterdam. Elsevier Science Publishers B.V. (North Holland).

Merlin, Philip M. [1979]. Specification and validation of protocols. *IEEE Transactions on Communications,* **COM-27**, 1671–1680.

Miller, Raymond E. [1990]. Protocol verification: The first ten years, the next ten years; some personal observations. In [Logrippo *et al.*, 1990] pp. 199–225.

Moore, Edward F., ed. [1964]. *Sequential Machines: Selected Papers.* Reading, Massachusetts: Addison-Wesley.

NBS [1987]. *User Guide for the NBS Prototype Compiler for Estelle.* Washington.

New, Darren H. and Amer, Paul D. [1991]. Protocol visualization of Estelle specifications. In [Quemada *et al.*, 1991] pp. 551–554.

Phalippou, Marc and Groz, Roland [1989]. Using Estelle for verification: an experience with the T.70 teletex transport protocol. In [Turner, 1989] pp. 185–199.

Postel, Jon, ed. [1980]. *DoD Standard Transmission Control Protocol.* RFC: 761. Marina del Rey, California: Information Sciences Institute.

Quemada, Juan, Mañas, Jose, and Vázquez, Enrique, eds. [1991]. *Formal Description Techniques, III.* Amsterdam. Elsevier Science Publishers B.V. (North Holland).

Rose, Marshall T. [1991]. *The Simple Book.* Englewood Cliffs, New Jersey: Prentice Hall.

Schwartz, Mischa [1987]. *Telecommunication Networks: Protocols, Modeling and Analysis.* Reading, Massachusetts: Addison-Wesley.

Sidhu, Deepinder P. [1990]. Protocol testing: The first ten years, the next ten years. In [Logrippo *et al.*, 1990] pp. 47–68.

Sidhu, Deepinder P. and Blumer, Tom P. [1986]. Verfication of NBS class 4 transport protocol. *IEEE Transactions on Software Engineering,* **COM-34**, 781–789.

Sijelmassi, Rachid and Strausser, Brett [1991]. NIST integrated tool set for Estelle. In [Quemada *et al.*, 1991] pp. 543–546.

Tarnay, Katie [1991]. *Protocol Specification and Testing.* Budapest: Akadémiai Kiadó. Also published by Plenum Press, New York.

Turner, Kenneth J., ed. [1989]. *Formal Description Techniques.* Amsterdam. Elsevier Science Publishers B.V. (North Holland).

Turner, Kenneth J. [1993]. *Using Formal Description Techniques — An Introduction to Estelle, LOTOS and SDL.* Chichester: John Wiley & Sons.

Vuong, Son T., ed. [1990]. *Formal Description Techniques, II.* Amsterdam. Elsevier Science Publishers B.V. (North Holland).

Vuong, Son T. and Chan, Wendy Y. L. [1989]. Validation of the Ferry Clip local testing system using an Estelle-C compiler. In [Turner, 1989] pp. 337–351.