X.21 Analysis Revisited: the Micro Tracer

Gerard J. Holzmann

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Protocol validation by exhaustive reachability analysis is a much tried procedure that has a well known set of flaws and strengths. The major strength is that the entire process can be automated for many types of protocol models. The major flaws are the time and space requirements. Efforts to extend the analytical power of the automated tools are usually of only casual interest since they only help to boost the complexity of the algorithms and render them almost useless in practice.

Although solving the performance problem is hard and requires sophisticated code, programming the basic symbolic execution routine itself is almost trivial. As an example, this paper discusses a one page Awk program that performs the task. The micro tracer is powerful enough to analyze the standard test−cases that are normally published, including but not restricted to, the alternating bit protocol and CCITT recommendation X.21.

1. Introduction

The behavior of a protocol can be formalized by a set of interacting finite state machines. The formal model can be analyzed by exhaustive inspection of all possible execution sequences for the set of machines. The number of sequences is typically large, and so are the time and space requirements of the symbolic execution algorithms.

Automated protocol validators have been developed for many different flavors of concurrent languages, using equally many different formal models. The capability of such systems, however, is often restricted to problems of the size of X.21 [2] and the alternating bit protocol [1], the two most frequently used test cases. For problems like these, with a composite state space size of a few thousand reachable states or less, not much sophistication is required to build a working validator.

The "micro tracer" to be described can analyze protocol systems with an arbitrary number of processes, it can model value passing, and it can detect execution loops. Surprisingly, these three features already make it stronger than the first validators described in the literature (e.g. [5]).

2. Model

We will use a basic finite state machine (FSM) model and a simple, relatively low−level, input language. Higher level constructs can easily be built with the primitives provided here.

Each process in the protocol is represented by a FSM in the model. We assume that the machines interact via signals. Each signal has a value and the value of any signal can be inspected and changed by any FSM in the system. Each FSM has a state. For each state a finite number of rules define how the state of the FSM can change. Those transition rules are of two types. The transition can either test a signal value or set it, but not both. (Note that this is different from a standard FSM model.)

A rule of the type

```
out A idle busy high S
```

defines an unconditional transition for FSM `A` in state `idle`. `A` can change its state to `busy` and change

the value of signal S to high.

A rule of the type

```
inp B idle busy high S
```

defines a conditional transition for FSM B in state idle. B can change its state to busy, provided that signal S has the value high. The value of the signal does not change when B makes the transition. A third, and last, type of rule defines the initial state for each machine:

```
init A idle
init B idle
```

The above format defines the complete input language for our model system. Note that variables, messages and mailboxes can, with some effort, all be modeled with signals and FSM's. A large class of protocols can be modeled in this manner.

As one example, the alternating bit protocol [1] can be defined as follows. We use two processes, a sender S and a receiver R. Each process has read access to one signal (an input port) and write access to another signal. For convenience, the signal read by S is also called S and the signal read by R is called R.

```
inp S state2 state2 ack0 S
inp S state2 state3 ack1 S
inp S state4 state1 ack0 S
inp S state4 state4 ack1 S
out S state1 state2 msg1 R
out S state3 state4 msg0 R
init S state1
```

The sender has four states. In the initial state, state1, the sender "transmits" signal msg1. It changes its state to state2 and awaits the response ack1. When that signal is detected the next signal, msg0, is transmitted and the sender will wait, in state4, for acknowledgement ack0.

The receiver process can be coded as follows.

```
inp R state2 state1 msg0 R
inp R state2 state3 msg1 R
inp R state4 state5 msg1 R
inp R state4 state6 msg0 R
out R state1 state2 ack0 S
out R state3 state4 ack1 S
out R state5 state4 ack1 S
out R state6 state2 ack0 S
init R state2
```

It can be hard to find inconsistencies in lists of rules like the above, especially if there are hundreds or thousands of rules to be considered. An automated validator can help.

3. A Micro Analyzer

The Awk program shown in Figure 1 can analyze FSM systems, written in the format given above, for the presence of deadlocks. It is deliberately presented in its basic form. The program can be run on UNIX® systems by typing

```
$ awk –f micro−tracer < protocol
```

where 'micro−tracer' is the file containing the Awk program, and 'protocol' is the file with the transition rules defining the system to be analyzed. (For an introduction to Awk see reference [3].)

The first three lines of the tracer deal with the input. Data are stored in two arrays. The initial state of machine A is stored in array element proc[A]. The transitions that machine A can make from state s are stored in move[A,s]. All data are stored as strings, and most arrays are also indexed with strings. All valid moves for A in state s, for instance, are concatenated into the same array element move[A,s], and later unwound as needed in function run().

The line starting with END is executed when the end of the input file has been reached and the complete protocol specification has been read. It initializes the signals and calls the symbolic execution routine run().

The program contains three function definitions: run(), mkstate(), and unwrap(). The global system state, state, is represented as a concatenation of strings encoding process and signal states. The

```
$1 == "init" {      proc[$2] = $3}
$1 == "inp"  {      move[$2,$3]=move[$2,$3] $1 "/" $4 "/" $5 "/" $6 "/;" }
$1 == "out"  {      move[$2,$3]=move[$2,$3] $1 "/" $4 "/" $5 "/" $6 "/;" }
END          {      verbose=0; for (i in proc) signal[i] = "-"
                    run(mkstate(state))
                    for (i in space) nstates++;
                    print nstates " states, " deadlocks " deadlocks"
             }

function run(state,  i,str,moved)# 1 parameter, 3 local vars
{
       if (space[state]++) return # been here before

       level++; moved=0
       for (i in proc)
       {      str = move[i,proc[i]]
              while (str)
              {      v = substr(str, 1, index(str, ";"))
                     sub(v, "", str)
                     split(v, arr, "/")
                     if (arr[1] == "inp" && arr[3] == signal[arr[4]])
                     {      Level[level] = i " " proc[i] " -> " v
                            proc[i] = arr[2]
                            run(mkstate(k))
                            unwrap(state); moved=1
                     } else if (arr[1] == "out")
                     {      Level[level] = i " " proc[i] " -> " v
                            proc[i] = arr[2]; signal[arr[4]] = arr[3]
                            run(mkstate(k))
                            unwrap(state); moved=1
       }      }      }
       if (!moved)
       {      deadlocks++
              print "deadlock " deadlocks ":"
              for (i in proc) print " " i, proc[i], signal[i]
              if (verbose)
                     for (i = 1; i < level; i++) print i, Level[i]
       }
       level--
}
function mkstate(state, m)
{      state = ""
       for (m in proc) state = state " " proc[m] " " signal[m]
       return state
}
function unwrap(state, m)
{      split(state, arr, " "); nxt=0
       for (m in proc) { proc[m] = arr[++nxt]; signal[m] = arr[++nxt] }
}
```

Figure 1 – Micro Tracer

function `mkstate( )` creates the composite, and the function `unwrap( )` restores the arrays `proc` and `signal` to the contents that correspond to the description in `state`. (The recursive step in `run( )` alters their contents.) Function `run( )` uses three local variables, but only one real parameter `state` that is passed by the calling routine.

The analyzer runs by inspecting the possible moves for each process in turn, checking for valid `inp` or `out` moves, and performing a complete depth-first search. Any state that has no successors is flagged as a deadlock. A backtrace of transitions leading into a deadlock is maintained in array `Level` and can be printed when a deadlock is found.

The first line in `run( )` is a complete state space handler. The composite `state` is used to index a large array . If the array element was indexed before it returns a count larger than zero: the state was analyzed before, and the search can be truncated.

After the analysis completes, the contents of array is available for other types of probing. In this case, the micro tracer just counts the number of states and prints it as a statistic, together with the number of deadlocks found.

4. Application of the Micro Tracer

The exhaustive analysis of the alternating bit protocol example takes 11.2 CPU seconds on a VAX−750. The complete analysis of the X.21 protocol (see appendix) takes five minutes.

It will not be hard to make changes in the program to validate a language with a richer semantics (i.e. rendez vous or asynchronous buffered message exchanges), or to validate the protocol for properties other than deadlocks. Any serious user, however, will discover that this micro analyzer works fine for small test cases (and may even outperform many ‘real’ protocol validators that have been described in the literature), but fails for the larger problems, e.g. with in the order of 10^5 or 10^6 reachable states.

A solution to that problem, the real problem of designing a practical validator, requires more thought and, alas, more code [4]. The difference in size between the validator described here and the one described in reference [4] is roughly 1:50. The difference in speed is more than 1:5000.

Acknowledgement

The micro tracer is directly inspired by a little Awk program that executes finite state machine descriptions, written by Jon Bentley.

5. References

[1] Bartlett, K.A., Scantlebury, R.A., and Wilkinson, P.T. "A note on reliable full−duplex transmission over half−duplex lines," Communications of the ACM, 1969, Vol. 12, No. 5, 260−265.

[2] Recommendation X.21 (Revised), CCITT (International Telegraph and Telephone Consultative Committee), Geneva, Switzerland, AP VI−No. 55−E, March 1976.

[3] Aho, A.V., Kernighan, B.W., Weinberger, P.J., "The AWK Programming Language," Addison−Wesley, 1988, 210 pgs.

[4] Holzmann, G.J., "An improved protocol reachability analysis technique," AT&T Bell Laboratories, TM 11271−870527−07, May 27, 1987, 18 pgs. (to appear in: Software Practice and Experience, 1988).

[5] West, C.H., and Zafiropulo, P., "Automated validation of a communications protocol: the CCITT X.21 Recommendation," IBM Journal of R&D, Vol. 22, No. 1, Jan. 1978, pp. 60−71.

APPENDIX

The X.21 Protocol

The transition rules are based on the two−process model for the call establishment phase of CCITT Recommendation X.21 derived in [5]. Interface signal pairs T, C and R, I are combined. Each possible combination of values on these line pairs is represented by a distinct lower−case ASCII character below. Note that since the lines are modeled as true signals, the receiving process can indeed miss signals if the sending process changes them rapidly and does not wait for the peer process to respond.

Figure 2 shows a state diagram for the dte and dce processes. Setting a signal to value `b` is indicated by `!b`, checking for a required signal value `c` is indicated by `?c`.

The error listings in section **4** are reproduced precisely as printed by the tracer. After each step number, the name of the executing machine is printed, followed by its state and an arrow. Behind the arrow is the transition rule: `inp` or `out`, the new state, the required or provided signal value, and the signal name.

1. Transition rules for the 'dte' process.

```
inp dte state01 state08 u dte
inp dte state01 state18 m dte
inp dte state02 state03 v dte
inp dte state02 state15 u dte
inp dte state02 state19 m dte
inp dte state04 state19 m dte
inp dte state05 state19 m dte
inp dte state05 state6A r dte
inp dte state07 state19 m dte
inp dte state07 state6B r dte
inp dte state08 state19 m dte
inp dte state09 state10B q dte
inp dte state09 state19 m dte
inp dte state10 state19 m dte
inp dte state10 state6C r dte
inp dte state10B state19 m dte
inp dte state10B state6C r dte
inp dte state11 state12 n dte
inp dte state11 state19 m dte
inp dte state12 state19 m dte
inp dte state14 state19 m dte
inp dte state15 state03 v dte
inp dte state15 state19 m dte
inp dte state16 state17 m dte
inp dte state17 state21 l dte
inp dte state18 state01 l dte
inp dte state18 state19 m dte
inp dte state20 state21 l dte
inp dte state6A state07 q dte
inp dte state6A state19 m dte
inp dte state6B state07 q dte
inp dte state6B state10 q dte
inp dte state6B state19 m dte
inp dte state6C state11 l dte
inp dte state6C state19 m dte
out dte state01 state02 d dce
out dte state01 state14 i dce
out dte state01 state21 b dce
out dte state02 state16 b dce
out dte state03 state04 e dce
out dte state04 state05 c dce
out dte state04 state16 b dce
out dte state05 state16 b dce
out dte state07 state16 b dce
out dte state08 state09 c dce
out dte state08 state15 d dce
out dte state08 state16 b dce
out dte state09 state16 b dce
out dte state10 state16 b dce
out dte state10B state16 b dce
out dte state11 state16 b dce
out dte state12 state16 b dce
out dte state14 state01 a dce
out dte state14 state16 b dce
out dte state15 state16 b dce
out dte state18 state16 b dce
out dte state19 state20 b dce
out dte state21 state01 a dce
out dte state6A state16 b dce
out dte state6B state16 b dce
out dte state6C state16 b dce
```

2. Transition rules for the 'dce' process.

```
inp dce state01 state02 d dce
inp dce state01 state14 i dce
inp dce state01 state21 b dce
inp dce state02 state16 b dce
inp dce state03 state04 e dce
inp dce state04 state05 c dce
inp dce state04 state16 b dce
inp dce state05 state16 b dce
inp dce state07 state16 b dce
inp dce state08 state09 c dce
inp dce state08 state15 d dce
inp dce state08 state16 b dce
inp dce state09 state16 b dce
inp dce state10 state16 b dce
inp dce state10B state16 b dce
inp dce state11 state16 b dce
```

```
inp dce state12 state16 b dce
inp dce state14 state01 a dce
inp dce state14 state16 b dce
inp dce state15 state16 b dce
inp dce state18 state16 b dce
inp dce state19 state20 b dce
inp dce state21 state01 a dce
inp dce state6A state16 b dce
inp dce state6B state16 b dce
inp dce state6C state16 b dce
out dce state01 state08 u dte
out dce state01 state18 m dte
out dce state02 state03 v dte
out dce state02 state15 u dte
out dce state02 state19 m dte
out dce state04 state19 m dte
out dce state05 state19 m dte
out dce state05 state6A r dte
out dce state07 state19 m dte
out dce state07 state6B r dte
out dce state08 state19 m dte
out dce state09 state10B q dte
out dce state09 state19 m dte
out dce state10 state19 m dte
out dce state10 state6C r dte
out dce state10B state19 m dte
out dce state10B state6C r dte
out dce state11 state12 n dte
out dce state11 state19 m dte
out dce state12 state19 m dte
out dce state14 state19 m dte
out dce state15 state03 v dte
out dce state15 state19 m dte
out dce state16 state17 m dte
out dce state17 state21 l dte
out dce state18 state01 l dte
out dce state18 state19 m dte
out dce state20 state21 l dte
out dce state6A state07 q dte
out dce state6A state19 m dte
out dce state6B state07 q dte
out dce state6B state10 q dte
out dce state6B state19 m dte
out dce state6C state11 l dte
out dce state6C state19 m dte
```

3. Initialization

```
init dte state01
init dce state01
```

4. Error Listings (verbose mode)

```
deadlock 1:
     dce state21 b
     dte state16 l
1 dce state01 -> out/state08/u/dte/;
2 dce state08 -> out/state19/m/dte/;
3 dte state01 -> inp/state18/m/dte/;
4 dte state18 -> inp/state19/m/dte/;
5 dte state19 -> out/state20/b/dce/;
6 dce state19 -> inp/state20/b/dce/;
7 dce state20 -> out/state21/l/dte/;
8 dte state20 -> inp/state21/l/dte/;
9 dte state21 -> out/state01/a/dce/;
10 dce state21 -> inp/state01/a/dce/;
11 dce state01 -> out/state08/u/dte/;
12 dce state08 -> out/state19/m/dte/;
13 dte state01 -> inp/state18/m/dte/;
14 dte state18 -> out/state16/b/dce/;
15 dce state19 -> inp/state20/b/dce/;
16 dce state20 -> out/state21/l/dte/;
deadlock 2:
     dce state03 b
     dte state16 v
1 dce state01 -> out/state08/u/dte/;
2 dce state08 -> out/state19/m/dte/;
3 dte state01 -> inp/state18/m/dte/;
4 dte state18 -> inp/state19/m/dte/;
5 dte state19 -> out/state20/b/dce/;
6 dce state19 -> inp/state20/b/dce/;
7 dce state20 -> out/state21/l/dte/;
8 dte state20 -> inp/state21/l/dte/;
9 dte state21 -> out/state01/a/dce/;
10 dce state21 -> inp/state01/a/dce/;
11 dce state01 -> out/state08/u/dte/;
12 dce state08 -> out/state19/m/dte/;
13 dte state01 -> out/state21/b/dce/;
14 dce state19 -> inp/state20/b/dce/;
15 dte state21 -> out/state01/a/dce/;
16 dte state01 -> inp/state18/m/dte/;
17 dce state20 -> out/state21/l/dte/;
18 dce state21 -> inp/state01/a/dce/;
19 dce state01 -> out/state18/m/dte/;
20 dte state18 -> inp/state19/m/dte/;
21 dce state18 -> out/state01/l/dte/;
22 dte state19 -> out/state20/b/dce/;
23 dte state20 -> inp/state21/l/dte/;
24 dce state01 -> out/state08/u/dte/;
25 dce state08 -> inp/state16/b/dce/;
26 dte state21 -> out/state01/a/dce/;
27 dte state01 -> inp/state08/u/dte/;
28 dce state16 -> out/state17/m/dte/;
29 dce state17 -> out/state21/l/dte/;
30 dce state21 -> inp/state01/a/dce/;
31 dce state01 -> out/state08/u/dte/;
32 dte state08 -> out/state15/d/dce/;
33 dce state08 -> inp/state15/d/dce/;
34 dce state15 -> out/state03/v/dte/;
35 dte state15 -> inp/state03/v/dte/;
36 dte state03 -> out/state04/e/dce/;
37 dte state04 -> out/state05/c/dce/;
38 dte state05 -> out/state16/b/dce/;
deadlock 3:
     dce state03 b
     dte state20 v
1 dce state01 -> out/state08/u/dte/;
2 dce state08 -> out/state19/m/dte/;
3 dte state01 -> inp/state18/m/dte/;
```

```
4 dte state18 -> inp/state19/m/dte/;
5 dte state19 -> out/state20/b/dce/;
6 dce state19 -> inp/state20/b/dce/;
7 dce state20 -> out/state21/l/dte/;
8 dte state20 -> inp/state21/l/dte/;
9 dte state21 -> out/state01/a/dce/;
10 dce state21 -> inp/state01/a/dce/;
11 dce state01 -> out/state08/u/dte/;
12 dce state08 -> out/state19/m/dte/;
13 dte state01 -> out/state21/b/dce/;
14 dce state19 -> inp/state20/b/dce/;
15 dte state21 -> out/state01/a/dce/;
16 dte state01 -> inp/state18/m/dte/;
17 dce state20 -> out/state21/l/dte/;
18 dce state21 -> inp/state01/a/dce/;
19 dce state01 -> out/state18/m/dte/;
20 dte state18 -> inp/state19/m/dte/;
21 dce state18 -> out/state01/l/dte/;
22 dte state19 -> out/state20/b/dce/;
23 dte state20 -> inp/state21/l/dte/;
24 dce state01 -> out/state08/u/dte/;
25 dce state08 -> inp/state16/b/dce/;
26 dte state21 -> out/state01/a/dce/;
27 dte state01 -> inp/state08/u/dte/;
28 dce state16 -> out/state17/m/dte/;
29 dce state17 -> out/state21/l/dte/;
30 dce state21 -> inp/state01/a/dce/;
31 dce state01 -> out/state18/m/dte/;
32 dte state08 -> out/state15/d/dce/;
33 dte state15 -> inp/state19/m/dte/;
34 dce state18 -> out/state01/l/dte/;
35 dce state01 -> inp/state02/d/dce/;
36 dce state02 -> out/state03/v/dte/;
37 dte state19 -> out/state20/b/dce/;
deadlock 4:
     dce state21 b
     dte state16 -
1 dte state01 -> out/state02/d/dce/;
2 dte state02 -> out/state16/b/dce/;
3 dce state01 -> inp/state21/b/dce/;
307 states, 4 deadlocks
```

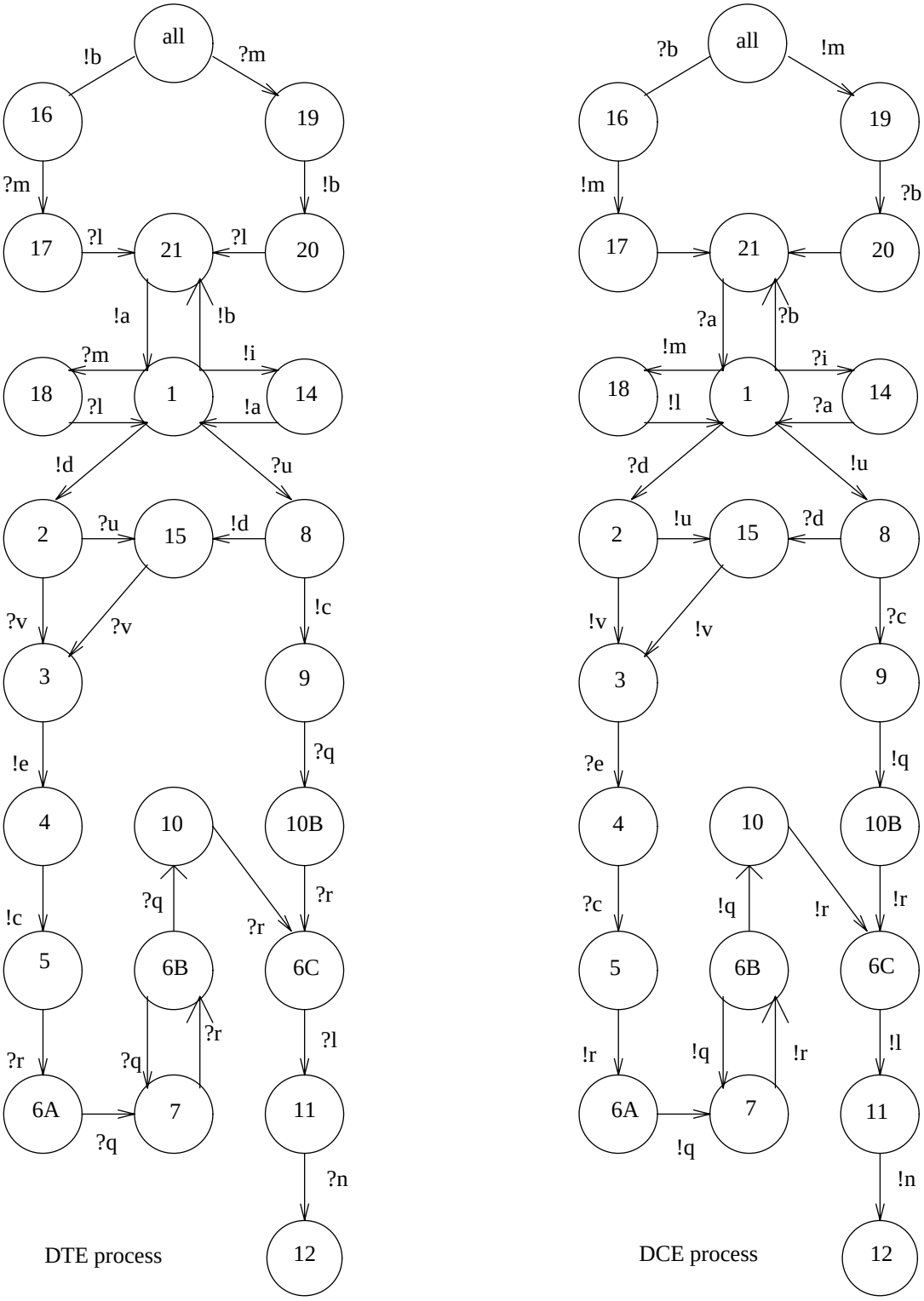

Figure 2 – X.21 Protocol Model