Optica Applicata, Vol. XXXIII, No. 4, 2003

Just-enough-time signaling protocol: formal description using extended finite state machine (EFSM)

A. HALIM ZAIM

Istanbul University, Department of Computer Engineering,
Avcilar, 34850, Istanbul, Turkey, e-mail: ahzaim@istanbul.edu.tr

In this study, we use an extended finite state machine (EFSM) model to describe just-enough-time (JET) signaling scheme running over a core DWDM (dense wavelength division multiplexing) network which utilizes optical burst switches (OBS). We apply an eight-tuple EFSM model. The state machines and transitions for a connection setup process between a source client node and a destination client node through an ingress and one or multiple intermediate switches are defined. We define some message channels for communication between the EFSMs.

Keywords: just-enough-time (JET), extended finite state machine (EFSM), formal protocol description, optical burst switches OBS.

1. Introduction

We are witnessing a great change in the protocol design phase during recent years in telecommunication industry. Formal design approaches (standardized or non-standardized) gain more and more importance. Instead of traditional design cycle, which includes a three step design cycle consisting of a high level design, low level design and coding and testing, a formal design approach that uses methods that help the designer verify the correctness of the design decisions as they are made, started to be used more frequently. For more information on formal design approaches, refer to [1]–[5].

Although there are different formal description techniques (FDT) for protocol specifications, extended finite state machines based approaches are the most popular FDTs, especially for communication protocols. Extended finite state machine (EFSM) approaches are fully expressive and usable, in particular, as a means of describing communication protocols. EFSM based techniques are used on the communication fields, because they are easier than most of the other approaches, and suitable for implementations. Therefore, in this study we used an EFSM based description model.

Just-enough-time (JET) signaling protocol is introduced in [6], [7]. The signaling architecture is based on wavelength routing and burst switching. Signaling is JET, indicating that signaling messages travel enough time ahead of the data they describe. Signaling is out of band, with signaling packets undergoing electro-optical conversion at every hop.

JET signaling approaches to optical burst switching (OBS) have been previously studied in the literature [8], [9].These approaches are characterized by the fact that the signaling messages are sent ahead of the data to inform the intermediate switches. The common thread is the elimination of the round-trip waiting time before the information is transmitted (the so-called tell-and-go approach): the switching elements inside the switches are configured for the incoming burst just before the burst arrives to the switching elements. The variants on the signaling schemes mainly differ in how to calculate the processing time at the switches for reservation purposes.

The organization of the paper is as follows. In Section 2, JET signaling protocol is explained briefly. The EFSM model is given in great detail in Sec. 3. In Section 4, we give the formal specification of JET protocol showing all state diagrams and explaining the state machines. Section 5 concludes our paper.

2. Just-enough-time signaling protocol

In the JET network we use an estimated setup and estimated teardown signaling scheme. Explanation of different signaling schemes can be found in [10].

The JET protocol is shown in Fig. 1, where the start and end of the burst are predicted based on the extra information contained in the *Setup* message.

Regardless of the type of the connection, it is initiated with a *Setup* message sent by the originator of the burst to its ingress switch. The ingress switch consults with delay estimation mechanism based on the destination address and returns the updated delay information to the originator by using a *Setup_Ack* message, at the same time acknowledging the receipt of the *Setup* message by the network. The *Setup_Ack* message also informs the originating node which channel/wavelength to use when sending the data burst.

The originator waits an offset time T based on its knowledge of the round-trip time to the destination client if it does not receive a *Setup_Ack* from the ingress switch. Otherwise, it waits for the ΔT duration of time that is returned to the source from the ingress switch based on the estimation of the round-trip delay. After waiting enough time, the originator sends the burst on the indicated wavelength. The *Setup* message at the same time is traveling across the network, informing the switches on the path of the burst arrival. If no blocking occurs on the path, the *Setup* message eventually reaches the destination node, which then receives the incoming burst shortly thereafter.

Upon the receipt of the *Setup* message, the destination node may choose to send a *Connect* message acknowledging the successful connection (indeed, the receipt of the *Setup* by the destination only guarantees that the connection has been established; it

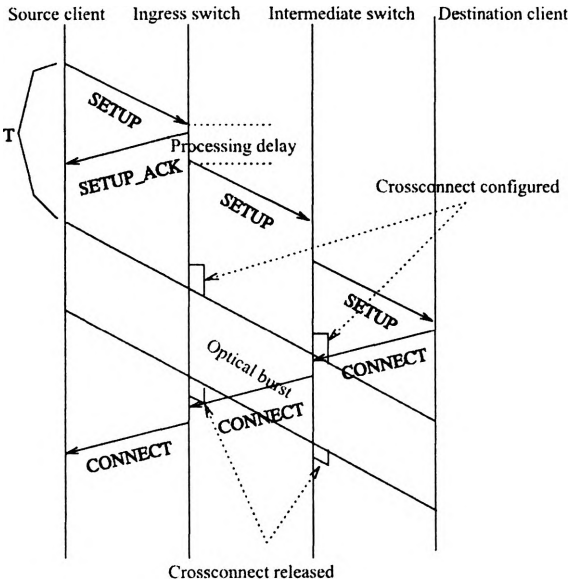

Fig. 1. JET signaling protocol.

does not guarantee its successful completion, since a connection may be preempted somewhere along the path by a higher-priority connection). The actual use of preemption is a subject of further study. The *Connect* message is also used to modify the offset times.

3. Extended finite state model

Ordinary finite state machine (FSM) representation is not powerful enough to model in a succint way the JET signaling protocol, because the protocol specifications include variables, timers and operations based on these values. Therefore, we define an EFSM model with the addition of some variables. In this model, each EFSM can be formally represented as an eight-tuple (Σ, S, s, V, E, T, A, δ), where: Σ – set of messages that can be sent or received, S – set of states, s – initial state, V – set of variables, E – set of predicates that operate on variables, T – set of timers, A – set of actions that operate on variables, δ – set of state transition functions, where each state transition function is formally represented as follows:

$$S \times \Sigma \times E(V) \times T \rightarrow \Sigma \times A(V) \times S.$$

There are two types of transitions: spontaneous and when transitions. A spontaneous transition does not have an input event on its condition part. A when transition, on the other hand, includes an input event satisfying the condition.

A transition is shown as $S_1 \xrightarrow{T} S_2$. This means there is a transition T at state S_1 and it goes to state S_2 (T is an outgoing transition, S_1 – the head state and S_2 – the tail state).

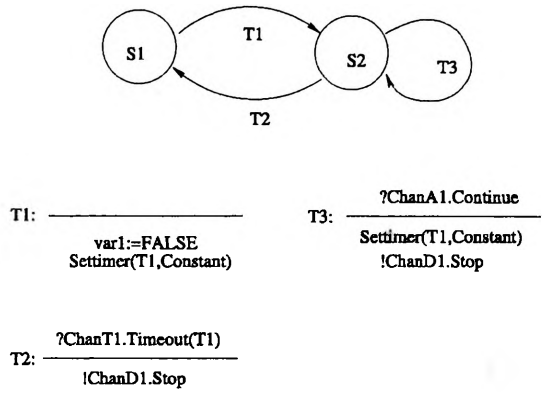

Fig. 2. Example of EFSM model

A transition consists of two parts: a condition part and an action part. The condition part has an input event and a predicate (Boolean expression). An action may be an output event or a statement operating on variables. A transition is executed when an input event is available, and a predicate is true. Once a transition is triggered, the action part is executed. An example of EFSM is shown in Fig. 2.

In Figure 2, *?Chan .m* shows an input message from the given channel carrying the message *m*, and *!Chan .m* shows an output message to the indicated channel carrying the message *m*. The *Settimer(T,C)* is an action defined to operate on timers. It sets the timer *T* to a value specified by *C*. Timers create *Timeout* messages using timer channels. As seen in Fig. 2, three transitions are defined in the EFSM. The definitions for each transition are given below the figure. The first transition, *T1* is a spontaneous transition, and is executed without an input event. The *T2* and *T3*, on the other hand, are when transitions because they are triggered once the input messages are received.

Protocols among different processes can often be modeled as a collection of communicating finite state machines where interactions between the processes are modeled by the exchange of messages [11]. EFSMs communicate with each other by message passing through a number of first-in-first-out (FIFO) unidirectional queues (channels), which associate with some buffers at the endpoints of the corresponding EFSMs, respectively.

4. JET protocol specifications

JET protocol can be defined as a set of extended finite state machines communicating with each other via message transfer. The protocol consists of unicast connections. In this section, we define the state diagrams of a source client, destination client, ingress switch and intermediate switch.

4.1. Source client

The FSM is defined for the source client sending unicast messages. The state diagram and the state transitions are given in Figs. 3 and 4, respectively.

The state transitions use four different channels: *ChanUpper*, *ChanNSUp*, *ChanNSDown*, and *ChanT1*. *ChanUpper* is the channel between the client node signaling protocol layer and the upper layer. *ChanNSUp* is the upstream channel between the client node and the ingress switch. That is, the flow is from the ingress switch to the client node. *ChanNSDown* is the downstream channel between the client node and the ingress switch, and the direction of the flow is from the client to the switch. *ChanT1* is the timer channel used to receive timeout messages from the indicated timers.

The state diagram waits in the *WAIT_FOR_OPEN* state until an open message comes from the upper layer. The open message triggers the transaction *T1*, and the machine goes to the state *WAIT_FOR_SETUP_ACK*. The action part of this transaction requires generation of a *Setup* message and setting of the timer setup acknowledgment timer (*SA_Timer*) to a predetermined value.

When the machine is in *WAIT_FOR_SETUP_ACK*, there are four possible transactions. Three of them, namely *T2*, *T3* and *T4* take us back to *WAIT_FOR_OPEN* state. *T2* is the event of receiving a *Failure* notice from the ingress switch. In this case we close the connection by telling the Upper Layer that a connection failure has happened. *T3* is a timer event. *SA_Timer* times out indicating that we did not receive

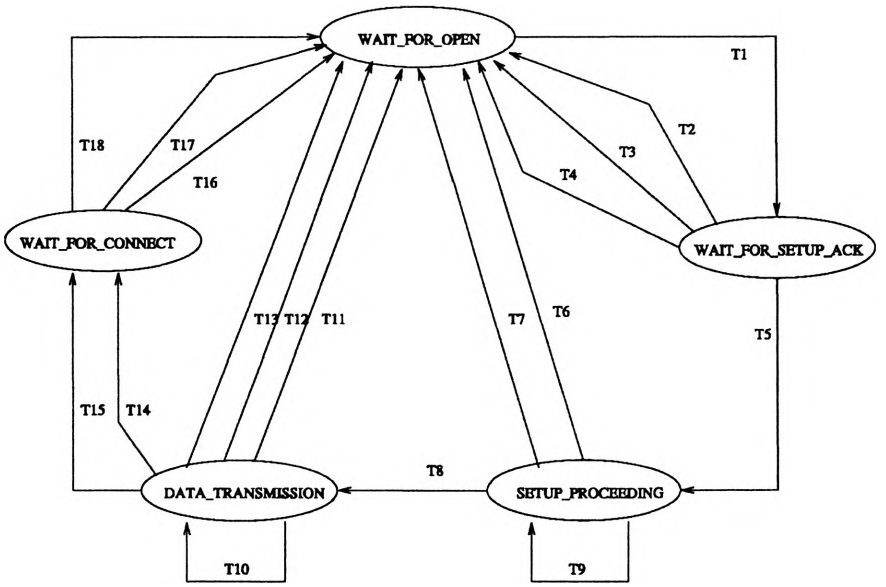

Fig. 3. State diagram for source client.

Fig. 4. State transitions for source client.

the setup acknowledgment within the expected time from the ingress switch. This event again indicates a connection failure. *T4* event is triggered by the upper layer. Upper layer wants to close the connection. In this case, the connection is closed and the ingress switch is notified by a *Release* message. On the other hand, if we receive a setup acknowledgment from the ingress switch within the specified time, *T5* is triggered by changing the state to *SETUP_PROCEEDING*. During that transition, the setup timer (*Setup_Timer*) and the connection timer (*Conn_Timer*) are set. *Setup_Timer* is used to determine the starting time of data burst and is calculated by the ingress switch and returned back to the source within *Setup_Ack* message. Connection timer, on the other hand, is used as a backup mechanism in closing the connection in case we do not get the *Connect* message although we are supposed to.

Once we are in *SETUP_PROCEEDING* state, a *Close* or a *Failure* message as mentioned above can take us back to *WAIT_FOR_OPEN* state (*T6*, *T7*). *T8* is a timer

event and is triggered with the *Setup_Timer* indicating that the start time for burst came. In this case we update the *Burst_Delay* by subtracting the processing time from the burst delay variable received by the *Setup* message and set the *Burst_Timer*. Then we notify the upper layer that data burst can be sent. The self-loop (*T9*) is used in case we receive the *Connect* message and all it does is to change the status of the variable *Conn_Rcvd* which is used as a flag variable showing the reception of the *Connect* message.

In *DATA_TRANSMISSION* state, *T10* is again a self-loop similar to *T9*, *T11*, *T12* and *T13* are *Close*, *Burst_Timer* timeout and *Failure* events, respectively, taking the state machine back to the starting state. *Close* and *Failure* events are already explained. *T12* indicates that the estimated burst time ended and we close the connection. *T14* and *T15* are the *Burst_Timer* timeout and *Close* events but in these transitions the state machine waits for a *Connect* message from the ingress switch and it has not been received until that time. We then go to the *WAIT_FOR_CONNECT* state.

In *WAIT_FOR_CONNECT* state, we either receive the *Connect* message and close the connection successfully, or get a *Failure* or connection timeout and close the connection with a failure (*T16*, *T17* and *T18*, respectively).

4.2. Destination client

The second state machine belongs to the destination side. The role of the destination client is to complete the *Setup* process and start receiving data until closing the connection with a timeout calculation based on the estimation of the burst time.

The state diagram and transitions are shown in Figs. 5 and 6.

The state diagram of the destination client is much simpler than the state diagram of the source client. The state machine waits at the *WAIT_FOR_SETUP* state until receiving a *Setup* message from the ingress switch (*T1*). Once it receives the *Setup*

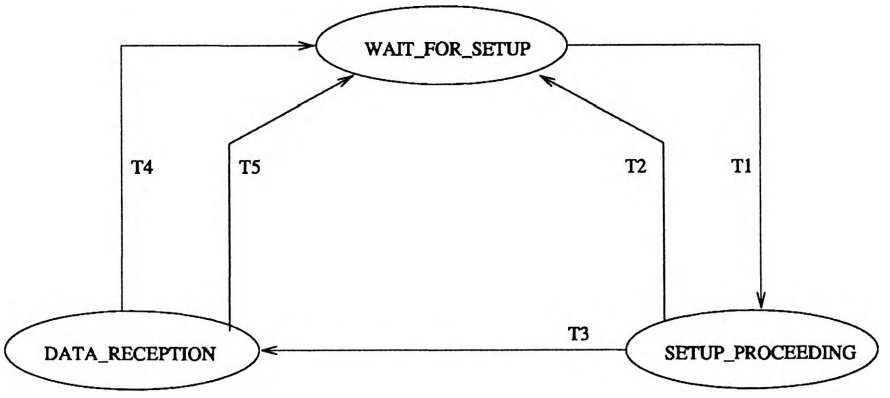

Fig. 5. State diagram for destination client.

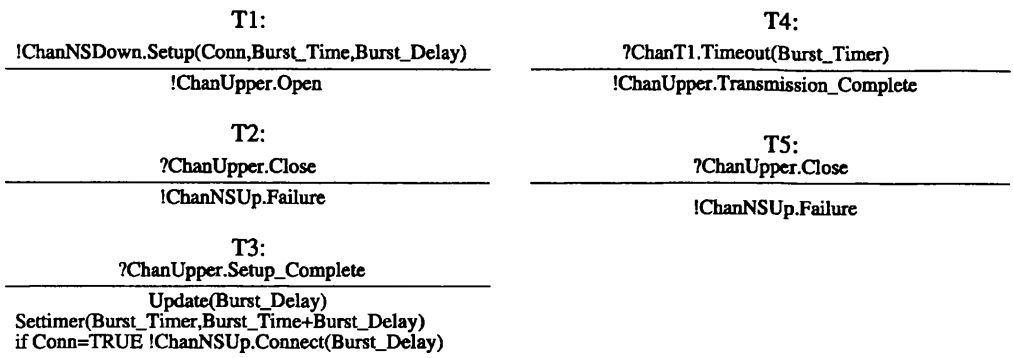

Fig. 6. State transitions for destination client.

message, it sends an open request to the upper layer. If the upper layer replies with a *Close* message, we go back to the *WAIT_FOR_SETUP* state generating a *Failure* message backwards to the ingress switch (*T2*). On the other hand, as shown in *T3*, if the upper layer replies with a *Setup_Complete* message notifying that the destination client completed the setup process successfully, it updates the *Burst_Delay*, sets the *Burst_Timer* and sends a *Connect* message if it is requested.

From *DATA_RECEPTION* state, we can go to the *WAIT_FOR_SETUP* state either by receiving a *Burst_Timer* timeout or a *Close* message. If the upper layer closes the connection earlier than the estimated time, this implies an error, so we generate a *Failure* message. Otherwise we close the connection and notify the upper layer about the end of the successful connection termination.

4.3. Ingress switch

This subsection gives the state diagram of an ingress switch receiving a *Setup* request from the source client, and configuring itself, and sending back the configuration information together with setup acknowledgment.

Setup message is sent by the source client. Once the *Setup* message is received, the ingress switch goes to *CHECKING_RESOURCES* state setting setup timer (*Setup_Timer*) and runs some checks (*T1*), *e.g.*, cyclic redundancy check (CRC), buffer overflow, cross connect error, *etc.* A *RunChecks* function is defined in this state machine. This function returns an error code specified with the variable *ErrorCode*. If there is an error, this variable indicates the type of an error found and return to *WAIT_FOR_SETUP* state. If it is null, the state machine stays at *CHECKING_RESOURCES* state (*T4*). If there is an error, *T2* is triggered and state machine goes back to *WAIT_FOR_SETUP* state generating a *Failure* message and notifying the upper layer to close the connection. If we receive a *Setup_Timer* timeout, indicating that the burst will start coming, we go to *DATA_TRANSFER* state. At *DATA_TRANSFER* state, we can get a *Burst_Timer* timeout (*T5* or *T7*) or a *Failure* (*T6*) from the intermediate switch forcing the state machine to go back to starting state.

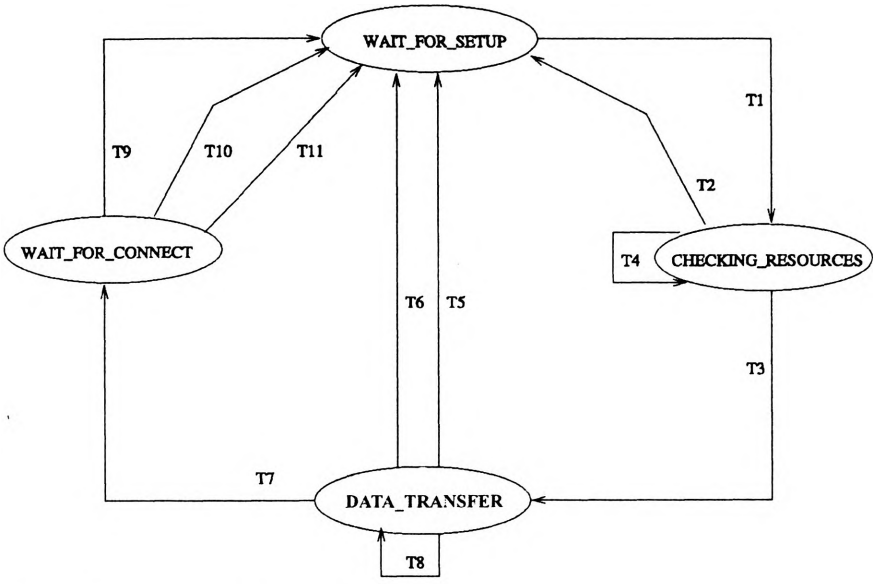

Fig. 7. State diagram for ingress switch.

T1:
?ChanNSDown.Setup(Conn,Burst_Time,Burst_Delay)

RunChecks(ErrorCode)
!ChanUpper.Open
Settimer(Setup_Timer, Δ T)

T2:
ErrorCode

!ChanNSUp.Failure
!ChanUpper.Close

T3:
?ChanT1.Timeout(Setup_Timer)

Update(Burst_Delay)
if Burst_Time!=Unspecified
{ Settimer(Burst_Timer,Burst_Time+Burst_Delay) }
Settimer(Conn_Timer,Conn_Time)

T4:
No ErrorCode

!ChanNSUp.Setup_Ack(Δ T)
!ChanSSDown.Setup(Conn,Burst_Time,Burst_Delay)

T5:
?ChanT1.Timeout(Burst_Timer)
---
if Conn=TRUE AND Conn_Rcvd=TRUE
{ !ChanNSUp.Connect(Burst_Delay) }

T6:
?ChanSSUp.Failure

!ChanNSUp.Failure

T7:
?ChanT1.Timeout(Burst_Timer) AND
Conn=TRUE AND Conn_Rcvd=FALSE

!ChanUpper.Close

T8:
?ChanSSUp.Connect(Burst_Delay)

Conn_Rcvd=TRUE

T9:
?ChanSSUp.Connect(Burst_Delay)

!ChanNSUp.Connect(Burst_Delay)

T10:
?ChanT1.Timeout(Conn_Timer)

T11:
?ChanSSUp.Failure

!ChanNSUp.Failure

Fig. 8. State transitions for ingress switch.

In case, we receive a *Burst_Timer* timeout, we check the status of the variables *Conn* and *Conn_Rcvd* to decide whether we need to wait for a *Connect* message or not as explained in earlier state machines. In *T8* we stay at the same state changing only the

status of the variable *Conn_Rcvd* to *TRUE*. At *WAIT_FOR_CONNECT* state, we wait for a *Connect* message. If we receive the *Connect* message we just pass it to the next switch and go to starting point. If at that time, we get a *Failure* message or a connection timer timeout, we again close the connection (*T10* and *T11*). The state diagram and the state transitions are given in Figs. 7 and 8.

The state transitions use five different channels: *ChanSSUp*, *ChanSSDown*, *ChanNSUp*, *ChanNSDown*, and *ChanT1*. *ChanNSUp*, *ChanNSDown* and *ChanT1* have already been defined. *ChanSSUp* is the channel between the ingress switch and the intermediate switch with the flow from the intermediate switch to ingress switch. *ChanSSDown* is the same channel with opposite flow direction.

4.4. Intermediate switch

The state diagram of an intermediate switch is similar to the state diagram of an ingress switch shown in Fig. 7. There are some notification changes and a minor change in the transition *T4* that does not generate a *Setup_Ack* message back. The transitions for an intermediate switch are given in Fig. 9 for completeness.

For intermediate switches some channels are defined as *ChanXS* because it is not known whether there is a switch or a node connected to the switch the diagrams belong to. Therefore these channels are defined anonymously.

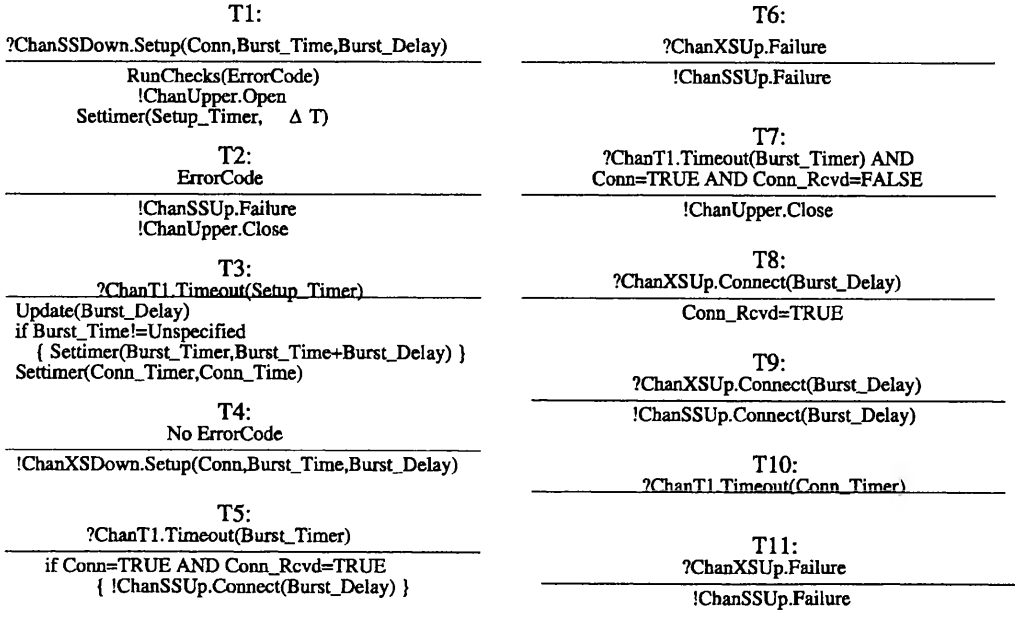

Fig. 9. State transitions for intermediate switch.

5. Conclusions

In this paper, we present a formal description of the JET signaling protocol for unicast traffic. The state diagrams and transitions for unicast traffic flows are given in this paper. We focused on the connection setup phase using the signaling channel. The reliability issue is left to the transport layer. The future work includes an addition of preemptive features to the switching elements. We also want to define the EFSMs for multicast traffic. These definitions will be used for protocol testing as an extension to this study.

References

[1] BOWMAN H., BLAIR G.S., BLAIR L., CHETWYND A.G., Computer Commun. **18** (1995), 964.
[2] HOLZMANN G.J., IEEE Software **9** (1992), 17.
[3] KING P.W., IEEE Trans. Comp. **40** (1991), 387.
[4] TURNER K.J., *The Use of Formal Methods in Communications Standards*, [In] *IEE Colloquium on 'Formal Methods Protocols'*, IEE, London 1991, pp. 1/1-3.
[5] HANSSON H., JONSSON B., ORAVA F., PEHRSON B., *Formal Design of Communication Protocols*, ISS'90.
[6] NAIL K., SARIKAYA B., IEEE Software **9** (1992), 27.
[7] TURNER J.S., J. High Speed Net. **8** (1999), 3.
[8] YOO M., QIAO C., *Just-enough-time (JET): A High Speed Protocol for Bursty Traffic in Optical Networks*, IEEE/LEOS Tech. Global Info. Infra., August 1997, pp. 26–27.
[9] QIAO C., YOO M., J. High Speed Net. **8** (1999), 69.
[10] YOO M., QIAO C., DIXIT S., JSAC **18** (2000), 2062.
[11] BALDINE I., ROUSKAS G.N., PERROS H.G., STEVENSON D., IEEE Commun. Magazine **40** (2002), 82.

Received July 17, 2003