Simulation of LAN Interconnection via ATM

by

Kurt Damm

A Thesis Submitted to the

Graduate Faculty in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Department: Electrical Engineering and Computer Engineering
Major: Computer Engineering

Iowa State University
Ames, Iowa
1994

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1. INTRODUCTION

Today most networks are dedicated to a special service, e.g., telephony or data transmission [1, 2]. ISDN standards were developed to support different services on the same network. ISDNs are now in an early stage of implementation. Despite the enthusiasm during the development of ISDN, it faces considerable difficulties on becoming established in the marketplace. Reasons for this may include the lack of new attractive services and the limited bandwidth [1, 2]. The International Telecommunications Union-Telecommunication Standardization Sector (ITU-TSS)—the former CCITT—is now in the process of standardization of a broadband ISDN (BISDN). This BISDN concept is intended to overcome the above mentioned shortcomings of the ISDN. The progress made in VLSI and optical transmission technology made new network concepts feasible. One of these new concepts is the asynchronous transfer mode (ATM). ITU-TSS has chosen ATM to become the transmission technique for the BISDN. ATM relies on the low bit error rate of the optical transmission medium. Therefore, all flow and error control has been shifted to the network boundaries. Furthermore, to make the processing at intermediate nodes easier, ATM is based on fixed sized packets called cells. An ATM cell consists of a 48 bytes data field and a five bytes header. In ATM, the cells of one channel are not restricted to a certain slot as in the synchronous transfer mode (STM). An ATM cell can always take the next free

slot. If there is no data to transmit unassigned cells which contain no information are inserted at the switch output and are discarded at the input of the next switch.

One of the conceptual problems to deal with in the BISDN is the combination of constant-bit-rate (CBR) traffic and varying-bit-rate (VBR) traffic. The challenge is to achieve high transmission link utilizations while maintaining a high quality of service. In this thesis, a model was developed using a simulation tool called OPNET. The model was intended to simulate the combination of VBR and CBR traffic in an ATM based network. Several simulations were made to investigate some of the problems which arise due to the combination of the distinct traffic types. The results were used to discuss network management concepts described in the literature to deal with this problems. Furthermore, common parameters used to describe bursty traffic were investigated and the ATM adaptation layer type 3/4 and the ATM adaptation type 5 were compared.

The solution to the above mentioned problems might be crucial for the BISDN to be able to compete against specialized network concepts [3]. Furthermore, a better understanding of the CBR-VBR traffic interactions is needed to be able to deal with the traffic exchange with hosts and future LANs in the Giga bits per second range. Another important impact of this improved understanding might be a better utilization of advanced video coding techniques based on the transmission of only non-redundant information. These techniques may lower the bandwidth requirements of video transmissions considerably but cause more traffic variance over time.

The remainder of this chapter gives an overview over future BISDN services and its classification according to ITU-TSS.

1.1 The BISDN Services

The BISDN services are intended to be used by private customers as well as by businesses. It is crucial for the success of the BISDN that it is able to offer new services in a cost effective manner and that it can easily adapt to new, future services whose characteristics are not yet defined or even totally unknown. Those services may include data transmission, audio and video transmission (still and moving pictures) or arbitrary mixtures of the above mentioned components. A service consisting of different service components is called a multimedia service. One example of a multimedia service is a multiparty desktop conference system. The technical issues of such a system (MERMAID) are discussed in [4]. Multimedia services are very likely to play an important role in the telecommunication market in the near future.

ITU-TSS defines in its Recommendation I.211 two different service types. The two types are interactive services and distribution services. The interactive services are further subdivided in conversational, messaging and retrieval services. In the distribution services a distinction is made on whether or not there is a user-individual presentation control.

The Tables 1.1–1.5 show examples of future BISDN services as defined by ITU-TSS [5, 1].

The conversational services are two way real-time communication services. Examples are shown in Table 1.1. The interconnection of LANs falls within this service class. Video-telephony and video conferencing are other examples which are expected to become very popular.

Retrieval Services allow a customer to get information stored in public libraries on demand. This information might be multimedia and ranges from travel informa-

tion to remote medical image communication. For further examples refer to Table 1.2.

Table 1.3 shows examples of messaging services. Messaging services are services without real-time constraints that are used for the exchange of (multimedia) messages. Possible services in this category are enhancements to existing electronic mail services.

The key word in distribution services with a user-individual presentation control is cabletext. Cabletext is an enhancement to the existing videotext and may incorporate multimedia information to such systems (see Table 1.4).

Distribution services without a user-individual presentation control are also called broadcast services. Important members of this service class are HDTV and electronic newspapers. Table 1.5 shows a more exhaustive list of examples.

Table 1.1: Conversational Broadband Services

Type of information	Examples of broadband services	Applications
Moving pictures (video) and sound	Broadband video-telephony	Communication for the transfer of voice (sound), moving pictures, and video scanned still images and documents between two locations • Tele-education • Tele-shopping • Tele-advertising
	Broadband Multipoint-Videoconference	Multipoint communication for the transfer of voice (sound), moving pictures, and video scanned still images and documents between more than two locations • Tele-education • Business conference • Tele-advertising
	Video-surveillance	• Building security • Traffic monitoring
	Video/audio information transmission service	• TV signal transfer • Video/audio dialogue • Contribution of information
Sound	Multiple sound-program signals	• Multi-lingual commentary channels • Multiple program transfers

Table 1.1 (Continued)

Data	High speed unrestricted digital information transmission service	• High speed data transfer – LAN interconnection – MAN interconnection – Computer-computer interconnection • Transfer of video information • Transfer of other information types • Still image transfer • Multi-site interactive CAD/CAM
	High volume file transfer service	• Data file transfer
	High speed teleaction	• Real-time control • Telemetry • Alarms
Document	High speed telefax	User-to-user transfer of text, images, drawings, etc.
	High resolution image communication service	• Professional images • Medical images • Remote games
	Document communication service	User-to-user transfer of multimedia documents

Table 1.2: Retrieval Broadband Services

Type of information	Examples of broadband services	Applications
Text, data, graphics, sound, still images, moving pictures	Broadband videotext	• Videotex including moving pictures • Remote education and training • Telesoftware • Tele-shopping • Tele-advertising • News retrieval
	Video retrieval service	• Entertainment purposes • Remote education and training
	High resolution image retrieval service	• Entertainment purposes • Remote education and training • Remote Professional image communication • Remote Medical image communication
	Document retrieval service	Multi-media retrieval from information centers, archives, etc.
	Data retrieval service	Telesoftware

Table 1.3: Messaging Broadband Services

Type of information	Examples of broadband services	Applications
Moving pictures (video) and sound	Video mail service	Electronic mailbox service for the transfer of moving pictures and accompanying sound
Document	Document mail service	Electronic mailbox for multimedia documents

Table 1.4: Distribution Broadband Services with User-Individual Presentation Control

Type of information	Examples of broadband services	Applications
Text, graphics, sound, still images	Full channel broadcast videography	• Remote education and training • Tele-advertising • News retrieval • Telesoftware

Table 1.5: Distribution Broadband Services without User-Individual Presentation Control

Type of information	Examples of broadband services	Applications
Data	High speed unrestricted digital information distribution service	• Distribution of unrestricted data
Text, graphics, still images	Document distribution service	• Electronic newspaper • Electronic publishing
Moving pictures and sound	Video information distribution service	• Distribution of video/audio signals
Video	Existing quality TV distribution service (NTSC, PAL, SECAM)	TV program distribution
	Extended quality TV distribution service • Enhanced definition TV distribution service • High quality TV	TV program distribution
	High definition TV distribution service	TV program distribution
	Pay-TV (pay-per-view, pay-per-channel)	TV program distribution

CHAPTER 2. THE BISDN PROTOCOL REFERENCE MODEL

The BISDN Protocol Reference Model (PRM) as defined by the ITU-TSS is shown in Figure 2.1. It consists of three planes: the user plane, the control plane, and the management plane. The user plane is responsible for the transfer of user data while call control and connection control functions fall into the responsibility of the control plane. The management plane incorporates two different functionalities. These are the layer management functions and the plane management functions. The layer management functions take care of the management of the resources and parameters at the respective layer. Functions relating to the whole system are part of the plane management functions. In addition, the plane management functions provide coordination between all planes.

The functions of the higher layers are not yet defined. On the other hand, considerable progress has been made by ITU-TSS in the standardization of the ATM Adaptation Layer (AAL), the ATM layer, and the Physical Layer. Figure 2.2 shows the functions of these layers and their sublayers.

A short description of the illustrated layers and their respective functions is given in the following sections.

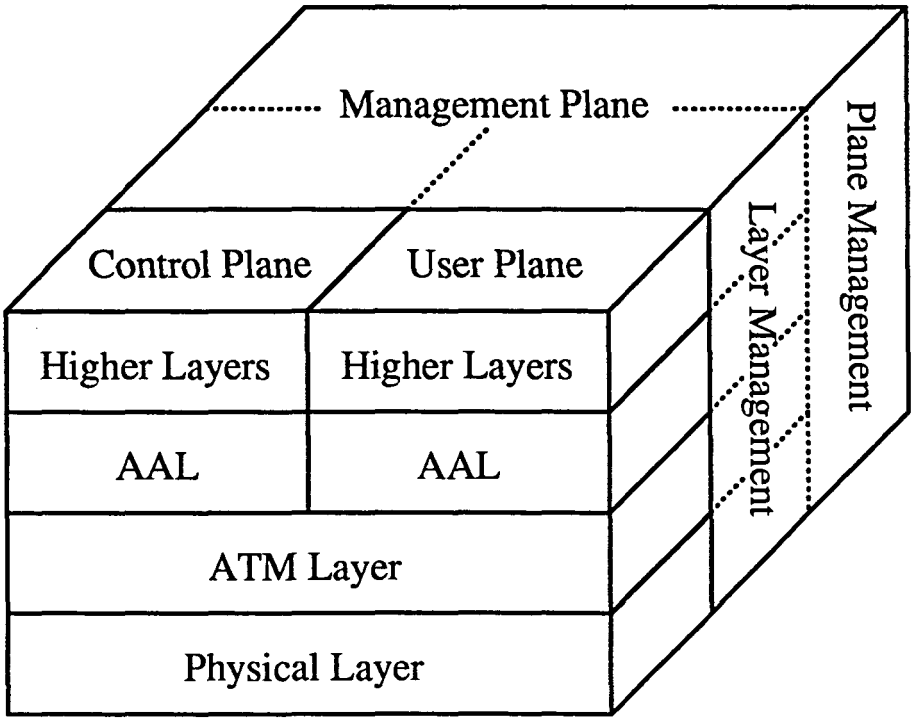
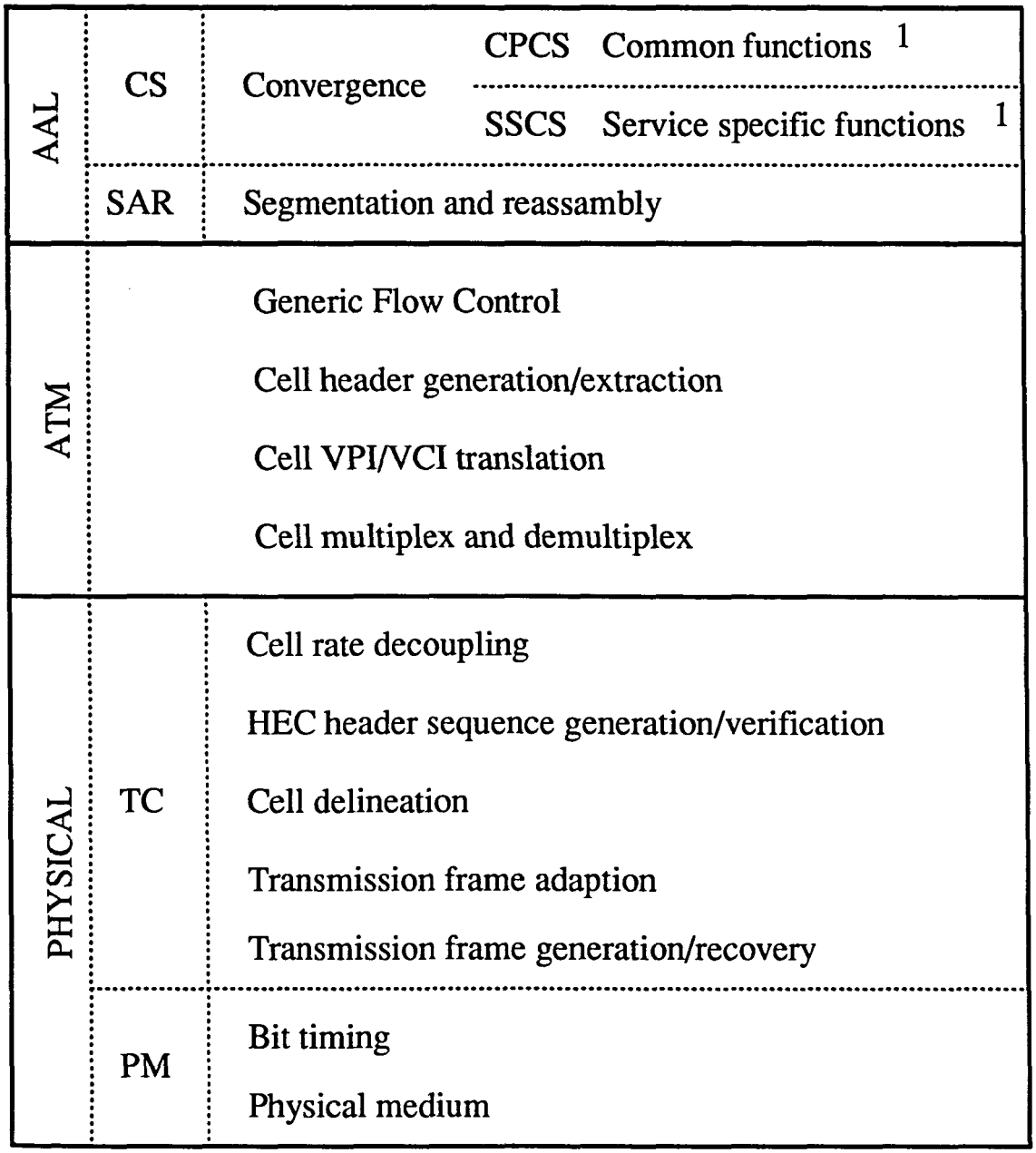


Figure 2.1: The BISDN Protocol Reference Model

2.1 Physical Layer

The Physical Layer consists of two sublayers. The two sublayers are the Physical Medium (PM) and the Transmission Convergence (TC) sublayer. The functions of the PM sublayer are divided into two sections, one dealing directly with the physical medium and the other handling the bit timing. The functions dealing with the physical medium are those functions concerned with the bit transmission and bit alignment, the line coding, and the electrical/optical conversion (if needed).

While bits are already recognized by the PM sublayer, the TC sublayer performs the tasks concerned with transmission frames. Furthermore, the TC sublayer gener-



1: CPCS and SSCS are only defined for AAL type 3/4 and AAL type 5.

Figure 2.2: The Layer Functions

ates the HEC header sequence for outgoing cells and verifies the HEC sequence for incoming cells. In addition, the HEC header sequence provides the means for the cell delineation. Finally, the function known as cell rate decoupling inserts unassigned cells during transmission and discards received unassigned cells.

2.2 ATM Layer

The ATM layer is independent of the underlying physical medium and performs four functions as shown in Figure 2.2. The basic functional unit of the ATM layer is the ATM cell. Its functions are described in the following sections.

2.2.1 ATM Cell Header Functionality

There are two different ATM cell headers defined by ITU-TSS. The cell header formats are shown in Figure 2.3 and Figure 2.4. One header format is used at the User-Network-Interface (UNI), the other at the Network-Network-Interface (NNI).

The difference between those two header formats is the four bit Generic Flow Control (GFC) field. It is only used at the UNI. At the NNI the four bits are used to enhance the Virtual Path Identifier (VPI) field from eight to twelve bits. The GFC field is part of congestion control strategies. Congestion control comprises three parts. First, an ATM network user must specify the traffic he intends to incorporate into the network. This traffic description is then used to determine whether or not this data is allowed to enter the network, i.e., whether or not the call is admitted. Finally, some kind of policing is needed to control the parameters agreed upon at the call establishment. The role of the GFC is to control terminals attached to user networks. The mechanisms to reach this goal are not yet defined [1, 2]. However,

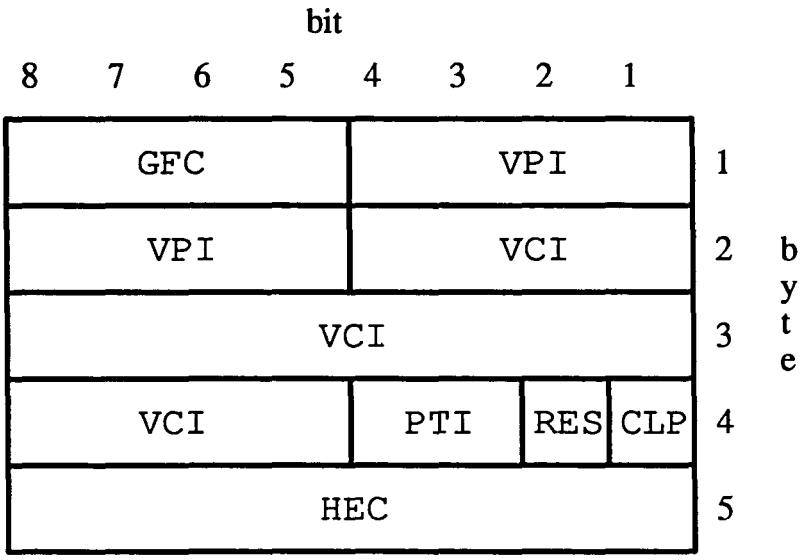


Figure 2.3: The ATM Cell Header Format at the User-Network Interface

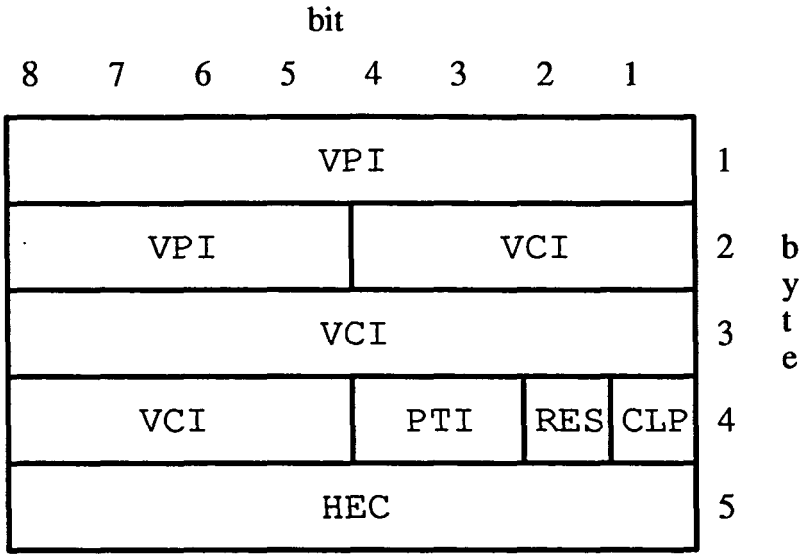


Figure 2.4: The ATM Cell Header Format at the Network-Network Interface

congestion control has been the subject of intense research (e.g., [6, 7, 8, 9]) and some controversy about its technical and economical feasibility [10, 11].

The ATM routing field consists of the VPI and the Virtual Channel Identifier (VCI). Its use is described in the next section. The two bit Payload Type Identifier (PTI) field is used to distinguish user information and Operation and Maintenance (OAM) cells. Furthermore, it is used in the AAL type 5 to indicate the last segment of a segmented CPCS_PDU (see Section 2.3.2). The Reserved (RES) field has not yet been defined. It is intended to provide the means to further enhance the cell header functionality. The Cell Lost Priority (CLP) may be used to tag cells exceeding the guaranteed bandwidth (CLP=1). Those cells should be first discarded in the case of network congestion. Finally, the Header Error Control (HEC) field, which is processed by the physical layer, provides the means to detect bit errors in the cell header. This is particularly important since faulty routing information can lead to the wrong delivery of cells. A misrouted cell may disturb two connections: the one it was originally destined for and the one it was accidently delivered to. I.e., the connections experience cell loss and cell gain, respectively.

2.2.2 Virtual Paths and Virtual Connections

The BISDN is based on virtual connections. A virtual connection (VC) is identified by a Virtual Connection Identifier (VCI). Several VCIs are put together to form a virtual path (VP). A VP is identified by a Virtual Path Identifier (VPI). VCI and VPI constitute the routing field of the ATM cell header. The relationship between VCs, VPs and the physical medium is shown in Figure 2.5.

As can been seen, a VCI has only to be unique within a VP. The VP in turn

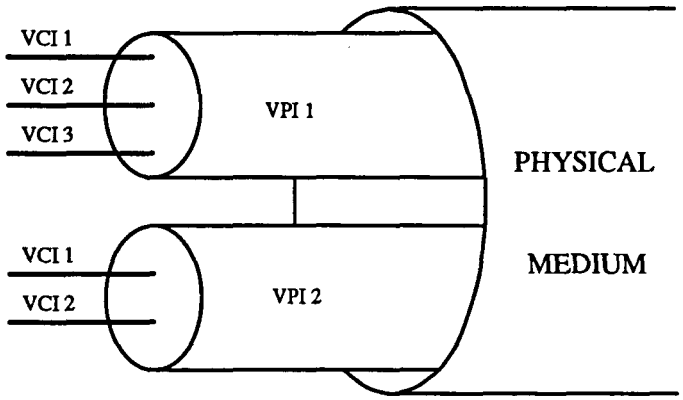


Figure 2.5: The Relationship of Virtual Connections, Virtual Paths and the Physical Medium

has only significance within one physical link. In other words the combination of VC and VP give the complete routing information within one link. VPIs and or VCIs are changed on a per link basis. In nodes where only VPs are switched (ATM crossconnects) the VCs within a VP remain unchanged.

The concept of VPs simplifies the resource management of ATM networks [12]. The reserved capacity of a VP need to be updated to keep track of changes in the flow of traffic, but this has to be done not as frequent as if the capacity would only be allocated on a VC basis. VCs are usually reserved on a per call basis and released when they are no longer needed. If there is already a VP with sufficient resources for the anticipated connection, no call processing has to be done at intermediate nodes [12].

Another advantage of the use of VCs is that it provides the means to decouple the components of multimedia services, e.g., video telephony. One could start a call

without video and later start and stop the video transmission [2].

2.3 ATM Adaptation Layer

The ATM Adaptation Layer (AAL) consists of two sublayers. These two sublayers are the Convergence Sublayer (CS) and the Segmentation and Reassembly (SAR) sublayer.

The CS is service dependent and might be empty for some services. It encapsulates user data and provides transmission and error detection facilities. These functions are provided on the basis of user specific logical units such as bytes, bit streams or variable length packets.

The SAR sublayer converts CS packets to ATM cells (segmentation) at the User-Network-Interface (UNI) and restores the CS packets upon reception of all segments from the ATM network (reassembly). The SAR sublayer provides transmission and error detection on a per cell basis.

At present, there are four different AAL types defined by the ITU-TSS. Originally, the distinction was made according to the timing relation, the bit rate, and the connection mode (see Figure 2.6).

One AAL type was defined for each service class. ATM Adaptation Layer type 3 (for Class C data) and AAL type 4 (for Class D data) emerged to one standard, namely AAL type 3/4. Simultaneously, a new AAL was proposed by the LAN industry. This proposal is now being standardized by ITU-TSS as AAL type 5 [13]. For the AAL type 3/4 and the AAL type 5 the CS is further subdivided into the Service Specific Convergence Sublayer (SSCS) and the Common Part Convergence Sublayer (CPCS). The CPCS provides the necessary means to transport variable sized data

	CLASS A	CLASS B	CLASS C	CLASS D
Timing relation between source and destination	Required		Not required	
Bit rate	Constant	Variable		
Connection mode	Connection oriented			Connectionless

Figure 2.6: The Original AAL Type Distinction

across a virtual connection. The SSCS may incorporate functions like flow control and retransmission. It is expected that the SSCS is an empty layer for most services during the introduction phase of BISDN [13]. Therefore, the SSCS is assumed to be an empty layer throughout the thesis and is not mentioned in subsequent sections.

Since the AAL type 3/4 and AAL type 5 were of special interest for this thesis, they are described in more detail in the following sections.

2.3.1 AAL type 3/4

Two frame formats are of particular interest in the AAL. These are the packet formats of the CPCS_PDU and the SAR_PDU.

The frame format of a CPCS_PDU for AAL type 3/4 is shown in Figure 2.7. The Common Part Identifier (CPI) is used to indicate if the Buffer Allocation size (BAsize) field is used. If CPI is set to zero, the BAsize field contains an estimate of the size of the current CPCS_PDU. It is intended to be used at the receiving side to preallocate sufficient buffer space for the reception of the CPCS_PDU. The Beginning tag (Btag) and the Ending tag (Etag) fields are used for the purpose of error detection. Their value is set to a equal value prior to transmission and must

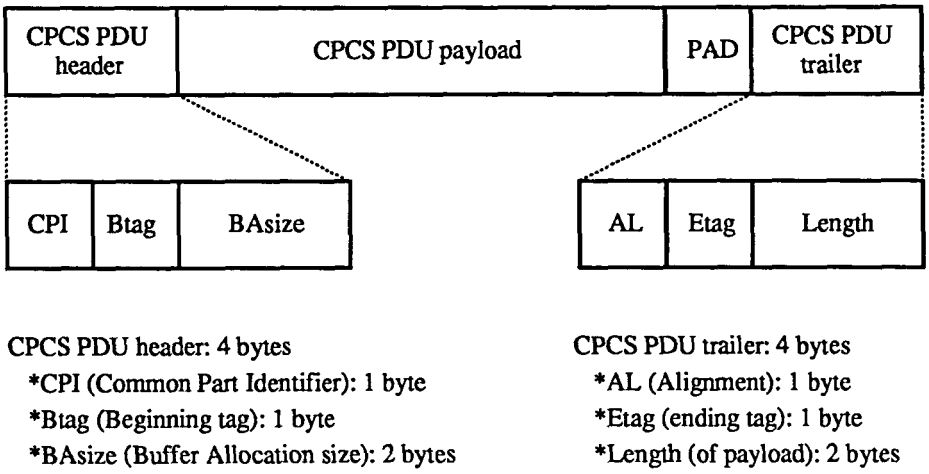
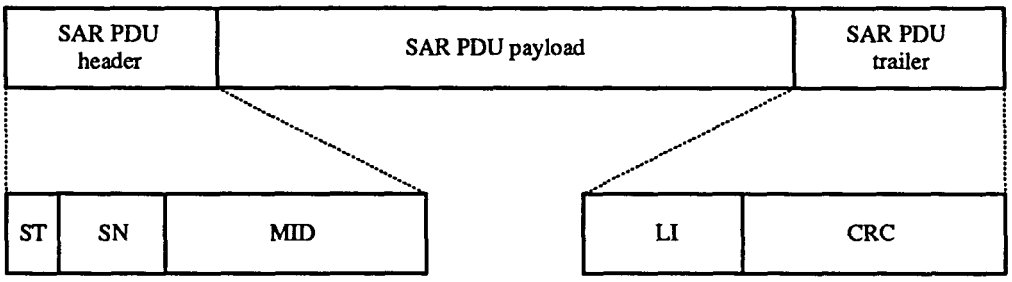


Figure 2.7: The AAL Type 3/4 CPCS_PDU Frame Format

not be the same for two successive frames. The Padding (PAD) field assures that the CPCS_PDU payload is aligned on a four bytes boundary. The Alignment (AL) field pads the trailer to four bytes. The Length field reports the length of the CPCS_PDU payload.

The format of a AAL type 3/4 SAR_PDU is shown in Figure 2.8. The Segment Type (ST) field indicates the message type of the SAR_PDU. There are four defined segment types. These are the Single-Segment Message (SSM), the Begin of Message (BOM), the Continuing of Message (COM), and the End of Message (EOM). The Sequence Number (SN) field is used to number the segments of one CPCS_PDU and provides the means to detect lost or wrongly delivered segments. The Multiplexing Identifier (MID) distinguishes different ATM packet streams which are multiplexed into one virtual connection. The Length Indicator (LI) field indicates the number of useful bytes in the SAR_PDU payload. Its value is always 44 for BOM and COM



SAR PDU header: 2 bytes
*ST (Segment Type): 2 bits
*SN (Sequence Number): 4 bits
*MID (Multiplexing Identification): 10 bits

SAR PDU trailer: 2 bytes
*LI (Length Indicator): 6 bits
*CRC (Cyclic Redundancy Check): 10 bits
SAR PDU payload: 44 bytes

Figure 2.8: The AAL Type 3/4 SAR_PDU Packet Format

segments and may vary from 0–44 for SSM and EOM messages. Finally, the Cyclic Redundancy Check (CRC) provides the means to detect bit errors.

The schematic procedure of the segmentation as defined for AAL type 3/4 is shown in Figure 2.9. A higher layer PDU is adopted as a AAL_SDU. A four byte header and a four byte trailer is added to the AAL_SDU to generate a CPCS_PDU. This CPCS_PDU is then divided into 44 byte segments. These segments are adopted as SAR_PDU payloads. The last payload might be not entirely filled with useful information, i.e., with information corresponding to the segmented CPCS_PDU. Each SAR_PDU payload is encapsulated with a two byte header and a two byte trailer to form a SAR_PDU. The SAR_PDU is then passed to the ATM layer. In the ATM layer the SAR_PDU is adopted as the ATM cell payload. An ATM header is added to form a ATM cell. This cell is passed to the Physical layer and subsequently sent to the respective destination.

At the receiving side the reverse functions are performed. The Physical layer

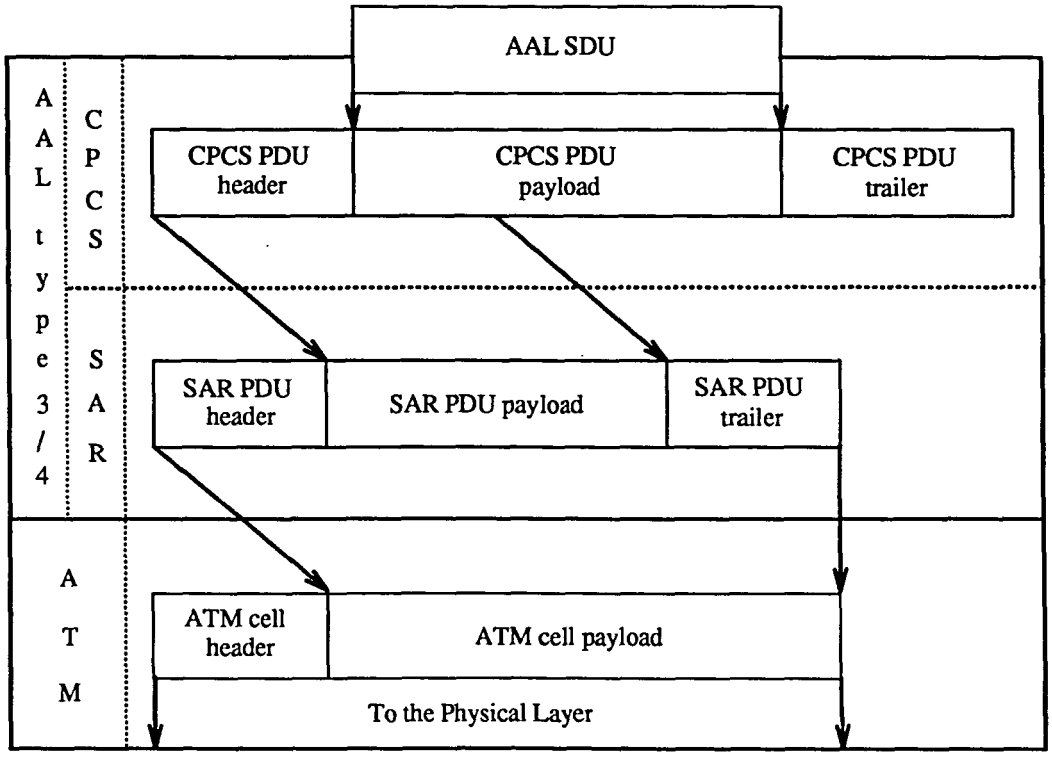


Figure 2.9: Schematic AAL Type 3/4 Segmentation

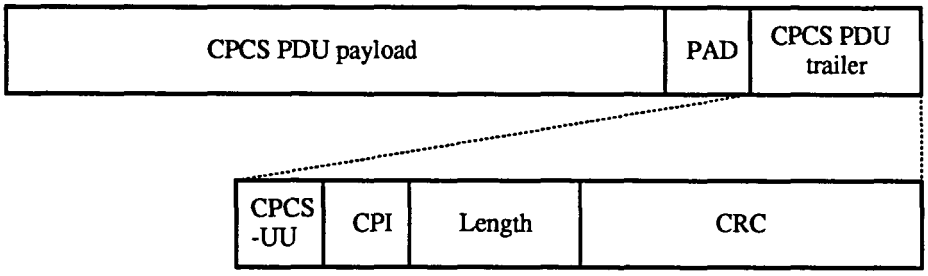
passes a received cell to the ATM layer. At the ATM layer the header is stripped off and the data field, i.e., the SAR_PDU is passed to the SAR sublayer. The SAR sublayer first checks the CRC to find out if the received SAR_PDU is error free. If an error is detected the SAR_PDU is discarded. AAL type 3/4, however, provides the means to pass partly reassembled CPCS_PDUs to the CPCS user entity. A CPCS_PDU which is to be reassembled is identified by the MID. The reception of a SAR_PDU with the ST field indicating a BOM causes the SAR sublayer to start the reassembly of a new CPCS_PDU. A CPCS_PDU with the same MID which has not yet been completely reassembled indicates an error. This is because ATM

guarantees the cell sequence integrity. The cell sequence integrity also insures that subsequently received COM segments with the same MID have to have consecutive sequence numbers. Similarly, an EOM segment has to have the SN following the SN of the last received COM segment. Otherwise, an error has occurred, i.e., a segment has been lost or gained. If an EOM segment arrives, the CPCS_PDU is reassembled and passed to the CPCS sublayer. In the CPCS sublayer the Btag and Etag fields are compared. If they do not have equal values, an error has occurred. Next, the CPCS sublayer checks if the reported length of the CPCS_PDU matches the length of the reassembled CPCS_PDU. If no error has been detected, the AAL_SDU is decapsulated and passed to the CPCS user.

2.3.2 AAL type 5

In the AAL type 5 approach all SAR_PDU payload encapsulation has been removed. Furthermore, the Btag, Etag and BAsize fields of the CPCS_PDU are omitted. Through the low error probability of fiber optic technology it is sufficient to shift responsibility for the multiplexing to higher layers and to simplify the error detection facilities [13].

The frame format of the CPCS_PDUs for AAL type 5 is shown in Figure 2.10. The error detection of the CPCS_PDUs is taken care of by means of a four byte CRC. The LI is set to the length of the CPCS_PDU payload. It provides the means to detect a segment loss or gain in the rare cases where the CRC is not able to detect such errors. The use of the CPI field has not been defined. It is set to zero. The CPCS-User-to-User indication (CPCS-UU) provides the means to exchange one byte of data between two peer CPCS user entities. The PAD field is responsible to align



CPCS PDU trailer: 8 bytes
*CPCS-UU (user-to-user indication): 1 byte
*CPI (Common Part Identifier): 1 byte
*Length (of CPCS PDU payload): 2 bytes
*CRC (Cyclic Redundancy Check): 4 bytes

PAD (Padding): 0-47 bytes

Figure 2.10: The AAL Type 5 CPCS_PDU Packet Format

the CPCS_PDU on a 48 byte boundary.

The simplified segmentation of an AAL_SDU is shown in Figure 2.11. An AAL-SDU passed to the CPCS is adopted as the CPCS_PDU payload. An eight bytes header and the padding field are added to form the CPCS_PDU. The fields in the header are set as described above. In the SAR sublayer the CPCS_PDU is divided into 48 bytes segments. These segments are passed to the ATM layer where an ATM header is added. The ATM cells are then passed to the physical layer and sent to their respective destination.

There are only two different SAR_PDUs in AAL type 5. They are distinguished by the value of the ATM-layer-user-to-ATM-layer-user (AUU) parameter in the Payload Type Indication (PTI) field of the ATM cell header. The AUU is set to one for the last (or the only) segment of the CPCS_PDU. In all other segments, AUU

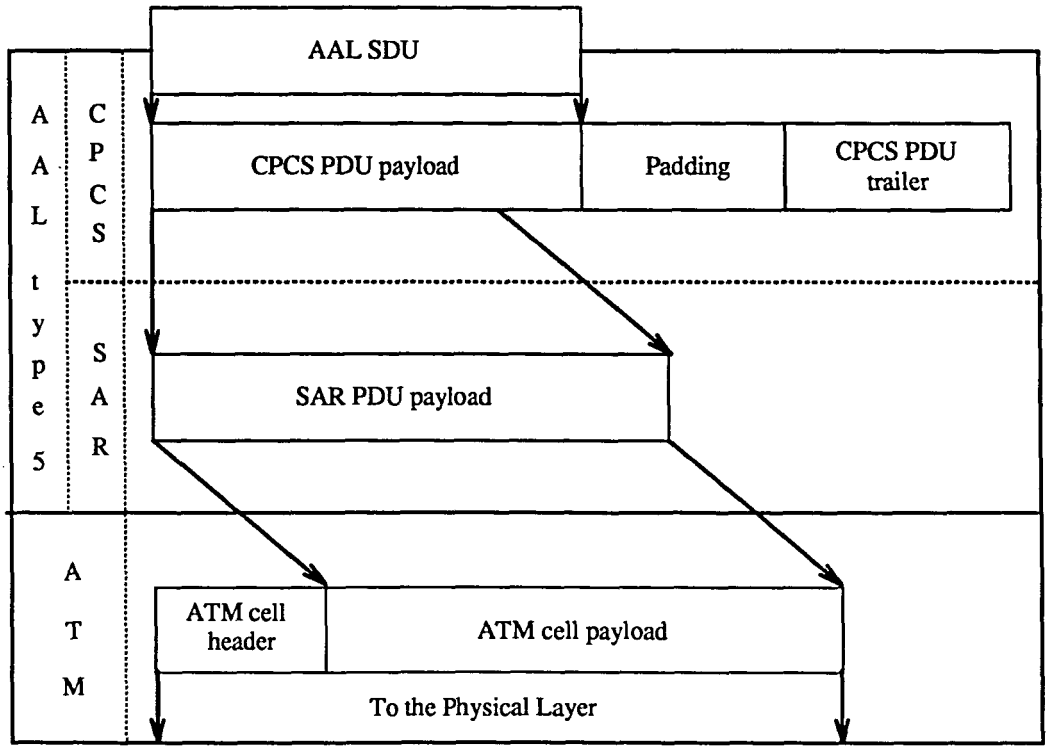


Figure 2.11: Schematic AAL Type 5 Segmentation

is set to zero. At the receiving side, the reception of an ATM cell with AUU set to one indicates the last segment of a CPCS_PDU has been received. This initiates the reassembly of the received segments to restore the original CPCS_PDU. In case the CRC did not detect an error and the length reported in the Length field matches the length of the reassembled CPCS_PDU payload, the AAL_SDU is decapsulated and is delivered to the AAL user process. A segment arriving after a segment with AUU set to one is considered the beginning of a new CPCS_PDU.

CHAPTER 3. THE DEVELOPED OPNET MODEL

Before the developed model is described, an overview over the OPNET simulation tool is given. An exhaustive description of OPNET can be found in [14, 15]. The following is a basic description that is intended to make the model description easier to understand.

3.1 The OPNET Simulation Tool

3.1.1 General Description of OPNET

OPNET provides the means to model the behavior and to evaluate the performance of communication networks and distributed systems. This is done by using discrete event simulation.

The modeling consists of three phases

- Specification
- Data collection and simulation
- Analysis

The relationship between these phases is as follows. First, a specification (=system model) is developed. This specification is then compiled to an executable program.

During the execution of the program, the behavior of the system is simulated and statistical data is collected. Then, the collected data is analyzed. After analyzing the data, the above mentioned cycle might be repeated several times to either change the model or run the simulation program again using different parameters. If the developed model is reasonably accurate, the collected data should represent the behavior and/or performance of the actual system.

The specification of an OPNET model is achieved using four tools or editors. They are called

- Network
- Node
- Process
- Parameter

The network, node and process editors are called the three modeling domains of OPNET. They are in a hierarchical relationship to each other. The parameter editor may be used in either of the three modeling domains and is used to provide modeling parameters and data structures.

3.1.2 The Network Domain

The main purpose of the network domain is to define the topology of the model. The building blocks needed to define the topology of the model described later in this chapter, were subnetworks, fixed communication nodes, and duplex point-to-point communication links. Subnetworks provide the means to introduce several

hierarchical levels within the network domain. There is no limit on the number of subnetworks used, but eventually there has to be a subnet consisting only of communication nodes and links. Fixed communication nodes are connected to each other with communication links. The behavior of the fixed communication node is determined by the chosen node model. The node model is edited in the node domain.

3.1.3 The Node Domain

The node model may represent a variety of computing and communication devices, e.g., bridges or workstations. The node model is build using modules. Generally speaking, there are two different kinds of modules. There are modules whose behavior is pretty much predefined and can only be influenced changing built in attribute values. Modules of this class are ideal generators, point-to-point transmitter, and point-to-point receiver. On the other hand, there are processor and queue modules whose behavior can be customized by means of process models. The process models are defined in the process domain.

The node model can be build using any number of modules. There are two ways in which the modules can be connected to each other. One way is using packet streams, the other way is using statistical wires. Packet streams are used to transmit data packets while statistical wires are used to transmit single numerical values or control information.

3.1.4 The Process Domain

As mentioned above, a process model specifies the behavior of a queue or process module in the node domain. The process model is defined using a state transition

diagram (STD). In the STD, actions are taken after being invoked by an interrupt. An interrupt may be caused by different events, e.g., the arrival of a packet, a statistical interrupt or the begin or the end of the simulation and so on. A special interrupt is the self interrupt. A self interrupt is scheduled by the process model itself. This will cause the process model being invoked again at a later time.

The STD has some features which go beyond that of traditional finite state machines:

- The STD maintains state variables to keep state information.
- Upon entry or exit of a state, arbitrary complex actions can be taken using C-programming and predefined functions, called Kernel Procedures (KPs) [16, 17]. These actions are called entry and exit executives, respectively. Typical actions of these executives are the change of state information, sending a packet, updating statistics and so on.
- The transition between states might be based on conditions. In addition, functions may be performed every time the transition is traversed.
- Model attributes provide the means to generalize the model.

A state of the STD can be either forced or unforced (the forced states are drawn bold in the figures). An unforced state is a stable state, i.e., it takes another interrupt to leave this state. Forced states are entered and left in one invocation. They have mainly the purpose to make the STDs easier to read (provide flow chart like modeling [14]). A special state of the STD is the initial state. The purpose of the initial state is to set up everything for the actual simulation. This may include the initialization

of state variables, registering of global statistics, loading of distribution functions, storing of model attributes in state variables, and so on. Typically, the initial state is invoked by a begin of simulation interrupt and is not entered again once the simulation has started.

Transitions specify a change from one state to another (sometimes the beginning and the ending state is the same). Usually, the change of states is based on some condition. A condition is a boolean expression that determines if a state transition is to be committed. Special forms of transitions are the default transition and the unconditional transition. The default transition is taken if, after the reception of an interrupt, no other condition is true. This is sometimes used to prevent run time errors caused by false interpreted interrupts. An unconditional transition is a transition that advances the process model to another state without any condition bound to it. A common use of unconditional transitions is to jump back from a forced to an unforced state after all actions related to the forced state have been taken.

3.1.5 Statistics

As was described above, the goal of developing a model of a system is to find something out about the behavior and/or performance of that system. The collection of statistical data during the execution of a simulation program provides an important help to reach this goal. There are two different kinds of statistics in OPNET. These statistics are the output vector statistics and the output scalar statistics, or vector and scalar statistics for short. A vector statistic typically records a simulation value (e.g., the delay experienced by a packet) over the simulation time. The data of a vector statistic are collected during a single execution of the simulation program.

In contrast, scalar statistics gain their data from several program executions. The data of the scalar statistics of one program execution are organized as a group of independent system values (e.g., mean and peak queue lengths). A scalar statistic shows the relationship between two system parameters instead of the value of one system parameter over time. Typically, each point in the scalar statistic correspond to the data collected during one simulation run.

Statistics can be either global or local. A local statistic collects data from within only one process module, while global statistics can be influenced by different modules.

3.2 Model Scope and Limitations

The first goal of this thesis was to develop a model that makes it possible to simulate the behavior of an ATM based network. The focus of the work was to see how an ATM based network performs if varying-bit-rate (VBR) and constant-bit-rate (CBR) traffic is combined. More specific, the case was considered where two FDDI networks are connected over an ATM network.

The network configuration shown in Figure 3.1 was chosen. It is built symmetrically consisting of the ATM nodes *ATM_1* and *ATM_4*, the ATM switches *ATM_2* and *ATM_3*, and the subnetworks *FDDI_1* and *FDDI_2*. The model for the FDDI subnetworks is shown in Figure 3.2. A FDDI usually connects several stations (up to 1000 [18]). Since the focus of this work was not on FDDI and to save simulation time, the FDDI network in the model consists of only three stations. The three stations divide the functionality of the FDDI among each other: one generates all the bursty traffic of the network (*vbr_station*), one all the constant-bit-rate traffic (*cbr_station*),

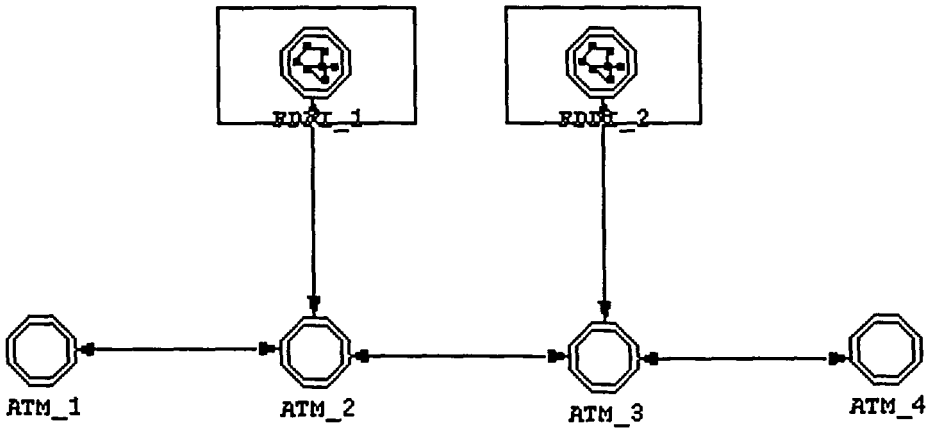


Figure 3.1: The Network Model

and one station bridges the traffic destined to the other FDDI network to the ATM network (*fddi_atm_link*).

There are three different methods for the exchange of data between FDDI networks via ATM networks [19]. The data exchange can be done using either a router or a bridge. Therefore, the term bridge should stand for both in the following description.

In the first approach, each bridge maintains a semi-permanent VC to each bridge of networks it might want to exchange data with. This method can also be used to emulate an extended LAN if the LANs are owned by the same corporation.

In the second approach, each bridge keeps a semi-permanent VC to a connectionless server rather than to other bridges. The connectionless server than forwards the data to the corresponding destination network. This method is more economical as the number of LANs that want to exchange data increases.

In the third approach, a VC is established every time data is to be exchanged and

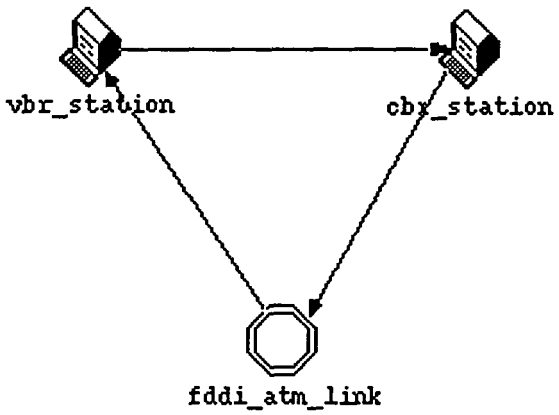


Figure 3.2: The FDDI Subnetwork

released when no longer needed. This is very appealing since the network resources are only reserved when needed, but this might put a considerable burden on the signalling system.

The first approach was chosen for the model. This confirms with the expectation that semi-permanent VPs between bridges are to be used for the LAN-LAN interconnection in early stage ATM networks [20].

In the scope of this work, it is assumed that the traffic between the ATM nodes *ATM_1* and *ATM_4* and vise versa is constant, and that no traffic originated in the ATM nodes is destined for the FDDI networks. Similarly, no data generated at the FDDI networks is destined for an ATM node. To make the above outlined traffic assumptions feasible, four VPs were considered as shown in Figure 3.3. These VPs are set up at the initialization of the OPNET model and are determined by the value of appropriate model simulation attributes.

As was mentioned above, the interest of the thesis was rather on the combination of VBR and CBR traffic than on a detailed protocol modeling. This statement is

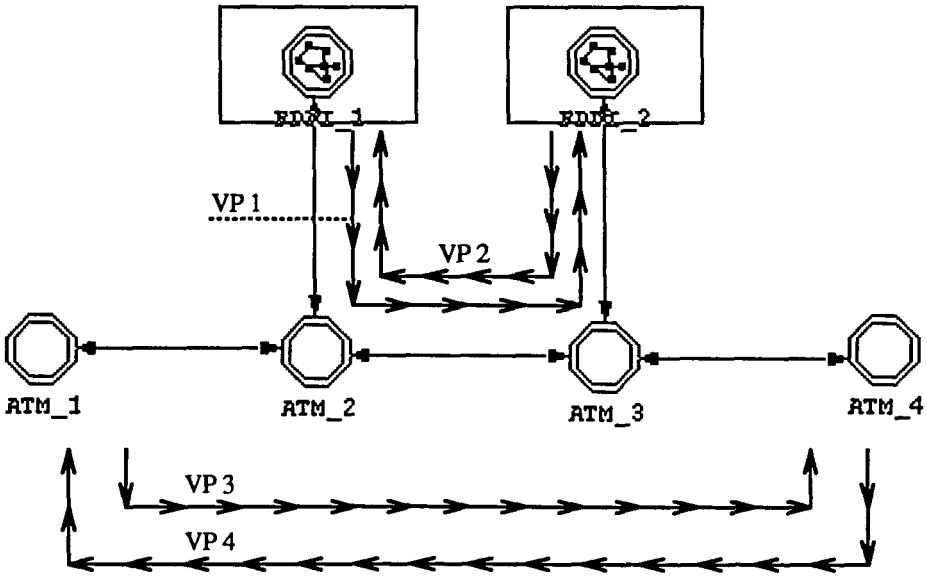


Figure 3.3: The Virtual Paths of the Model

also the reason that the ATM functions are not explicitly modeled according to their layered representation.

The two major functions performed by an ATM switch are the VPI/VCI translation and the routing of cells from the input to the output lines [1]. Furthermore, the switch inserts cells containing no information at the output whenever there is no data to transmit. These unassigned cells are discarded at the input of the next switch. This principle, known as cell rate decoupling, has not been incorporated in the model. Cell rate decoupling is necessary to provide the means for ATM switching at this high speed but is not needed to model it. The VPI/VCI translation, i.e., the changing of cell headers, has been omitted in the model since it has no influence on the collected data. The routing of cells is performed in a pure output queueing approach. The performance of output queueing is optimal in terms of the delay in

relation to the throughput. On the other hand, pure output queueing switches tend to have severe implementation problems [21]. Since this work is not concerned about the implementation of switches, the output queueing approach was adopted. The modeled switch performs better than a real world switch and is not concerned about drawbacks of a particular switch design.

ATM switching is a very complex task on its own. For a detailed discussion on ATM switching refer to the literature. A good description of different switch designs can be found in [2]. [21] comprises a good discussion of different queueing strategies and [22] includes a comparison of different switching architectures.

Packet formats used in the model may differ from their specifications. The differences are mentioned in the model description, where appropriate.

The segmentation and reassembly (SAR) of FDDI MAC frames follows the description in Section 2.3. The same assumptions were made, i.e., it was assumed that the SSCS is empty. Therefore, SSCS is not mentioned in the description of the SAR process.

Before the model is described in more detail, an overview over the available model attributes is given.

3.3 The Model Simulation Attributes

3.3.1 The Model Attributes of the ATM Nodes

src.interarrival args: This attribute originates in the ideal generator module of the ATM node. It is used to determine the number of packets to be generated at the ATM node. Its value is 1/number of packets to be generated.

proc.VPI_SET: This attribute originates in the process model *atm_nd_proc.* It determines the value to which the VPI field of the ATM cells originated at this node is set. This value determines where the cells are routed to as shown in Figure 3.3.

proc.VECTOR_STAT_ENABLE: This attribute originates in the process model *atm_nd_proc.* It signals whether or not the vector statistic about the end-to-end delay experienced by the ATM cells is to be recorded.

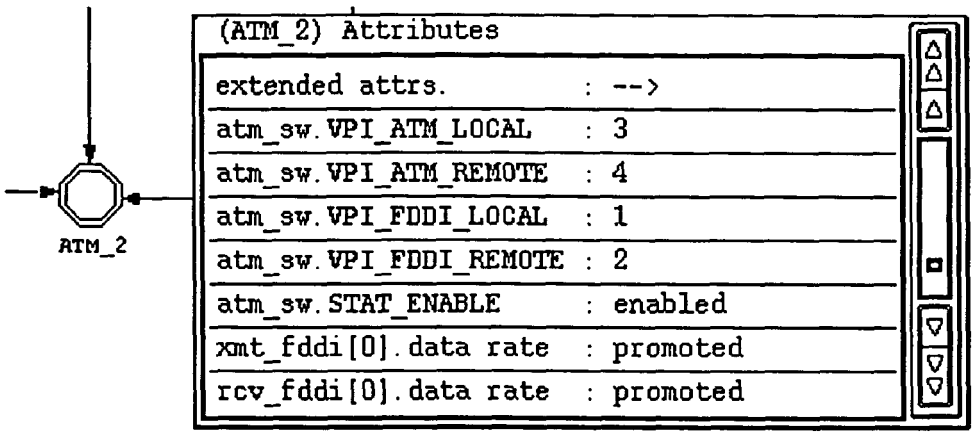


Figure 3.4: The Extended Model Attributes of the ATM Switch

3.3.2 The Model Attributes of the ATM Switches

Figure 3.4 shows the extended simulation model attributes of the ATM Switch *ATM_2.* They are described in general below.

atm_sw.VPI_ATM_LOCAL, atm_sw.VPI_ATM_REMOTE,

atm_sw.VPI_FDDI_LOCAL, atm_sw.VPI_FDDI_REMOTE: These four values originate in the process model *atm_sw_proc* and constitute the routing table for *atm_sw*. They are used to transmit the ATM cells according to the virtual paths shown in Figure 3.3. The suffixes LOCAL and REMOTE stands for one hop and two hops away, respectively. *VPI_ATM_LOCAL* for *ATM_2* (see Figure 3.4), for example, corresponds to the model attribute *VPI_SET* of the ATM node one hop away, i.e., *ATM_1*, which has the value three.

atm_sw.STAT_ENABLE: This attribute originates in the process model *atm_sw-proc* and determines whether or not the scalar statistics of the ATM switch are to be recorded.

xmt_fddi[0].data rate: This attribute originates in the point-to-point transmitter module *xmt_fddi* in the node model *atm_sw*. It determines the transmission capacity of the point-to-point transmitter in bits per second, and thus the time it takes to transmit a packet from the ATM switch to the FDDI subnetwork.

rcv_fddi[0].data rate: This attribute originates in the point-to-point receiver module *rcv_fddi* in the node model *atm_sw*. It determines the transmission capacity of the point-to-point receiver in bits per second and thus the time it takes to receive a packet from the FDDI subnetwork at the ATM switch.

3.3.3 The Model Attributes of the FDDI VBR Stations

This model is based on the OPNET FDDI example model. A copy of the model description from [23] can be found in Appendix C. In the following the model attributes not contained in the original model are explained. They all originate in the

process model *fddi_gen_vbr* of the processor module *llc_src*.

llc_src.dest_ring_id: This attribute indicates to which value the field *dest_ring_id* of the Interface Control Information (ICI) *fddi_mac_req_II* is to be set. The value determines to which FDDI network the FDDI frame should be delivered.

llc_src.vbr_gen_seed_I, llc_src.vbr_gen_seed_II: These two attributes are the seed numbers to initialize the random number generator used to determine the random numbers in *fddi_gen_vbr*.

llc_src.traffic_dist: This attribute determines the function according to which the frames at this station are to be generated.

llc_src.idle_dist: This attribute contains the function which determines the distribution of the idle periods in the process model *fddi_gen_vbr*. The outcome of the distribution *idle_dist* is used to schedule a self interrupt. This interrupt than determines the end of a busy period.

llc_src.idle_dist_arg: This attribute is the parameter of the above mentioned function *idle_dist*. Its value is the mean outcome of the distribution in seconds. The function and the parameter together determine the actual outcome of the distribution.

llc_src.busy_dist: This attribute contains the function which determines the distribution of the busy periods in the process model *fddi_gen_vbr*. The outcome of the distribution *busy_dist* is used to schedule a self interrupt. This interrupt than determines the end of a idle period.

llc_src.idle_dist_arg: This attribute is the parameter of the above mentioned function *busy_dist.* Its value is the mean outcome of the distribution in seconds. The function and the parameter together determine the actual outcome of the distribution.

3.3.4 The Model Attributes of the FDDI CBR Stations

This model is based on the OPNET FDDI example model. A copy of the model description from [23] can be found in Appendix C. The model attributes *llc_src.dest_ring_id* and *llc_src.traffic_dist* are not contained in the original model. These two attributes correspond to the attributes *llc_src.dest_ring_id* and *llc_src.traffic_dist* of the VBR stations described above.

3.3.5 The Model Attributes of the FDDI-ATM Bridges

bridge_proc.VPI_SET: This attribute originates in the process model *bridge_aal5_proc.* It determines the value to which the VPI field of the ATM cells—produced due to the segmentation of FDDI frames—at this node is set. This value determines where the cells are routed to as shown in Figure 3.3.

bridge_proc.STAT_ENABLE: This attribute originates in the process model *bridge_aal5_proc* and determines whether or not the scalar statistics of the FDDI-LAN bridge are to be recorded.

xmt_atm[0].data rate: This attribute originates in the point-to-point transmitter module *xmt_atm* in the node model *bridge_nd.* It determines the transmission capacity of the point-to-point transmitter in bits per second and thus the time

it takes to transmit a packet from the FDDI subnetwork to the ATM switch.

rcv_atm[0].data rate: This attribute originates in the point-to-point receiver module *rcv_atm* in the node model *bridge_nd.* It determines the transmission capacity of the point-to-point receiver in bits per second and thus the time it takes to receive a packet from the from the ATM switch at the FDDI subnetwork.

mac_sync bandwidth, mac.T_Req, mac.station_address, mac.ring_id: These four attributes correspond to the attributes of the OPNET FDDI Example Model. As mentioned before, a copy of the description from [23] can be found in Appendix C.

3.4 The ATM Nodes

An ATM node in the model has to accomplish several tasks. It has to receive cells destined for the node, use the information they contain to update statistics, and destroy the packet to free the memory associated to it. On the other hand, it has to assign a VPI value to the cells originated at this node and send the cells to the ATM switch. The OPNET node model for the ATM nodes is shown in Figure 3.5.

The model consists of four modules. The point-to-point receiver *rcv* gets the cells destined for the node. The point-to-point transmitter *xmt* sends the cells which were produced by the ideal generator *src* to the ATM switch. The actual processing of the cells takes place in the ATM node process model *atm_nd_proc* (see Figure 3.6) which resides in the process module *proc.*

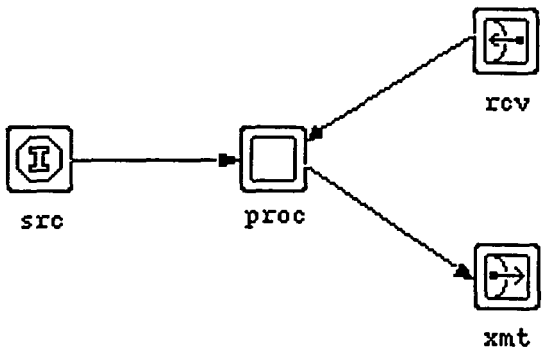


Figure 3.5: The ATM Node Model

3.4.1 The Process Model of the ATM Nodes

The ATM node process model *atm_nd_proc* is shown in Figure 3.6. It is composed of five states. Each state is described below:

init: The *init* state registers a global statistics handler for a statistic to record the end-to-end delay experienced by the received cells. Another task performed is to retrieve the value of the module attribute *VPI_SET*. *VPI_SET* is used to set the VPI field in the cell header for packets produced at this node.

After the initialization is done, an unconditional transition is made to the *idle* state.

idle: In the *idle* state the process model waits for an event to happen. An event can be either the arrival of a cell from the ideal generator *src*, the arrival of a cell from the point-to-point receiver *rcv*, or the end of the simulation. Either of these events causes a transition to another state, *xmt, rcv* or *stats*, respectively.

xmt: The *xmt* state is encountered when a packet from the *src* module arrives. The actions taken are, get the packet, set the VPI field in the header, and send the

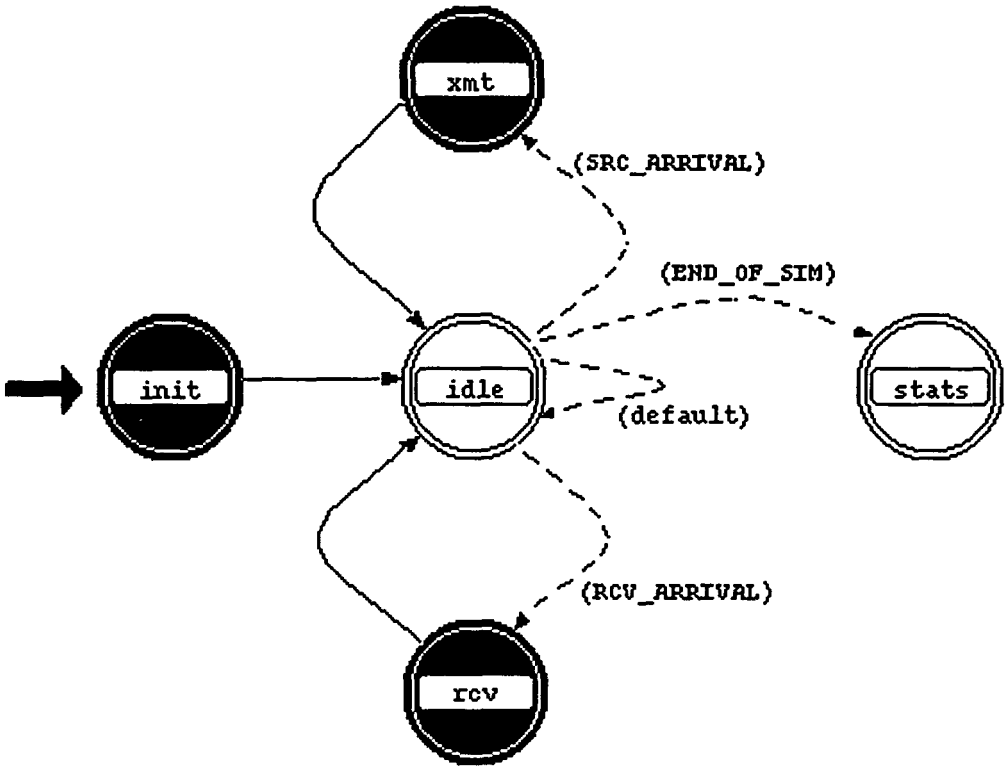


Figure 3.6: The ATM Node Process Model

packet to *rcv* (and thus to the ATM switch). After finishing, an unconditional transition is made back to the *idle* state.

rcv: The *rcv* state handles the arrival of a cell at the point-to-point receiver. After retrieving the packet, the end-to-end delay is calculated and the peak end-to-end delay is updated, if appropriate. If the flag *VEC_STAT_ENABLE* is set, the end-to-end delay is recorded in a global statistic. Note that for longer simulation runs the end-to-end delay statistic will result in huge vector output files (16 bytes per received packet). Therefore, one might not always want to record this statistic. Finally, the received cell is destroyed to free the memory

associated with it, and an unconditional transition is made back to the *idle* state.

stats: The transition to the *stats* state is committed when the the simulation is finished and the end of simulation interrupt had been enabled.

The statistic of interest is the peak end-to-end delay. It records the longest end-to-end delay experienced by a ATM cell during the simulation run. Note that only one of the two ATM nodes has to write this statistic since its value is determined by a global variable accessed by both nodes.

3.5 The ATM Switches

The work of the ATM switch is to route incoming cells to the appropriate output. No data originates at the switch. The OPNET node model for the ATM switches (*ATM_2* and *ATM_3*) is shown in Figure 3.7.

The ATM switch node model contains one point-to-point transmitter and one point-to-point receiver for each (duplex) communication link attached to it. The communication lines connect the switch to an ATM node, a FDDI subnetwork, and the other ATM switch (see Figure 3.1). The actual cell switching between these attached nodes is done by the ATM switch process model *atm_sw_proc* (see Figure 3.8) ruling the behavior of the processor module *atm_sw*.

The statistical wires (doted lines) provide the means to gather statistical information about the switch-to-switch connection. In addition, the information about the queue length of packets awaiting transmission to the other switch might be used in a model enhancement to decide whether or not a new incoming cell is to be discarded.

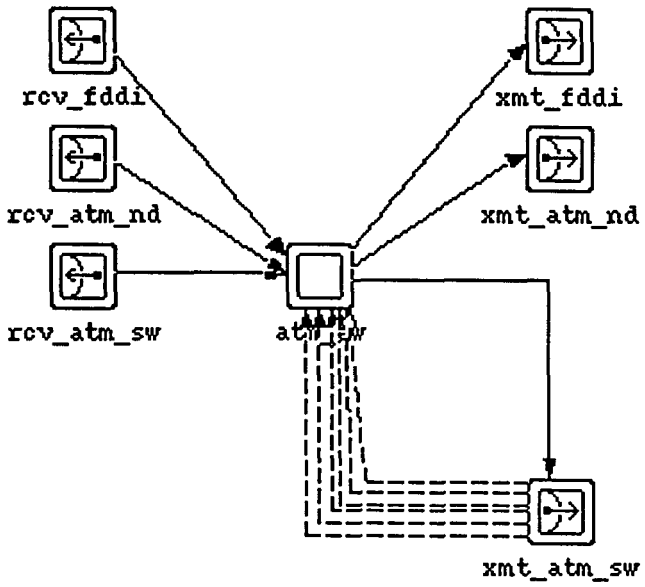


Figure 3.7: The ATM Switch Node Model

3.5.1 The Process Model of the ATM Switches

The process model *atm_sw_proc* is shown in Figure 3.8. Its three states are described below:

initial: In the *initial* state, the value of the variable *max_queue_length*, representing the maximum queue length at the link between the ATM switches, is set to zero (no packets, no queue). The second function of this state is to store routing information contained in the module attributes *VPI_ATM_LOCAL, VPI_ATM_REMOTE, VPI_FDDI_LOCAL* and *VPI_FDDI_REMOTE* in the state variables *vpi1-4*. The routing information corresponds to the virtual paths shown in Figure 3.3. It is used in the *route_pk* state to deliver the cells to their correct output line. After initializing, the process model *atm_sw_proc* makes an unconditional transition to the *idle* state.

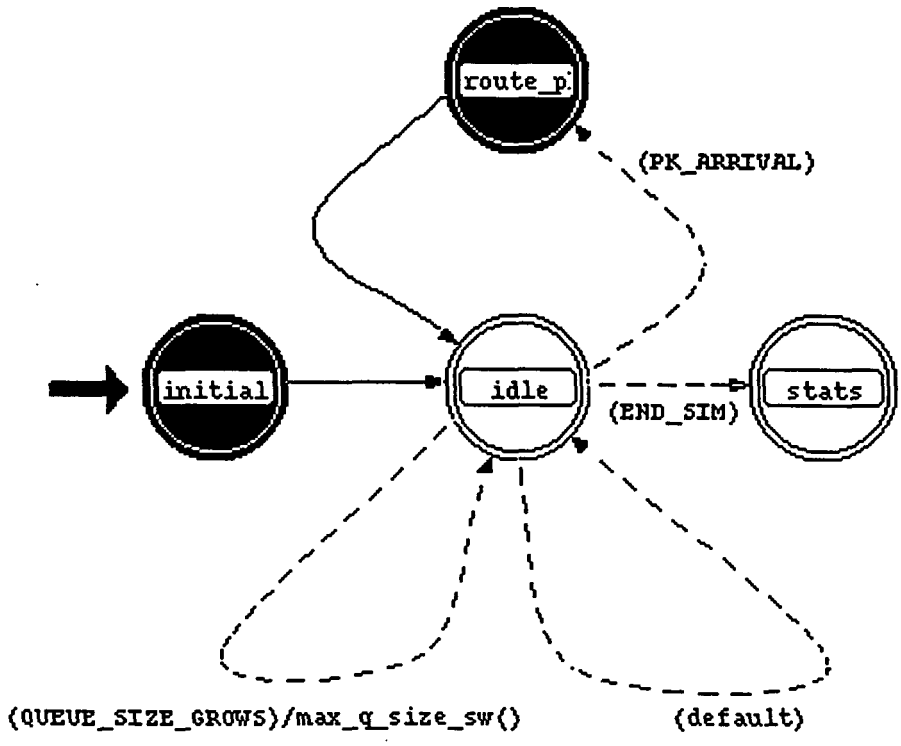


Figure 3.8: The ATM Switch Process Model

idle: In the *idle* state, *atm_sw_proc* is waiting for the arrival of an ATM packet, a statistic interrupt, or an end of simulation interrupt.

In the case that a packet arrives on one of the three point-to-point receivers, *atm_sw_proc* makes a transition to the state *route_pk*.

In the model, only one of the many possible interrupts of the statistical wires [14] is enabled. This interrupt occurs when the length of the waiting queue for the transmission to the other ATM switch rises (see Figure 3.9). Upon the arrival of the interrupt, the function *max_q_size_sw* is executed. The function is declared in the Function Block of *atm_sw_proc*. It reads the current queue size and compares it to its recorded maximum queue length. In the case of a

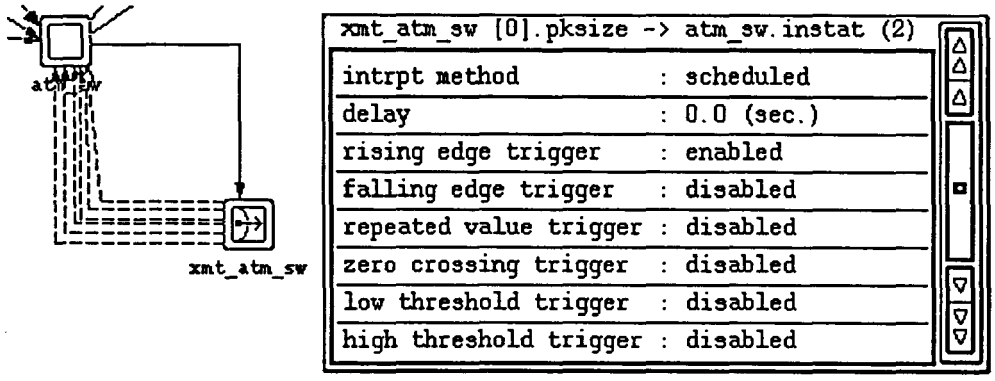


Figure 3.9: The Statistical Interrupts for the Queue Length Statistic

new highest value, the recorded maximum queue length is updated. After the execution of the function *max_q_size_sw*, *atm_sw_proc* returns to the *idle* state.

The other possible event is an end of simulation interrupt. This interrupt causes a transition to the state *stats*.

route_pk: As indicated by the name, this state is responsible for the routing of the received packets to their respective output. To accomplish this, the VPI field of the received packet is compared to the routing information at this node (see description of the *initial* state) and then sent to the corresponding point-to-point transmitter. After the transmission of the cell, *atm_sw_proc* returns to the *idle* state.

stats: In the state *stats*, a variety of scalar statistics is to be recorded. These statistics are the ATM switch throughput (in packets and Mb/s), the mean packet delay at the switch, the mean and maximum queue length on the connection to the other switch, and the utilization of the switch-to-switch link.

3.6 The FDDI Subnetworks

The OPNET models of the stations in the FDDI subnetworks are based on the FDDI example model provided by OPNET. It is described in [23]. A copy of the description is added to the thesis as Appendix C. In the following, only changes made to the existing FDDI model are shown. The name of changed models, packet formats and ICIs were formed keeping the original name and appending a _II. The formats of the MAC frame *fddi_mac_fr* and the interface control information *ici_mac_req* were changed to make the exchange of MAC frames between different FDDI rings feasible. A field, *dest_ring_id*, was added to both; a second field, *src_ring_id*, was introduced to *fddi_mac_fr*. Note that these two packet fields are artificial (therefore, the field length is set to zero). In a real life bridge approach, the bridge finds out about the destination ring address on its own, i.e., the addressing is transparent to the FDDI user. Since the focus of this work is not on the FDDI protocol or address resolution procedures, this simple solution has been adopted.

3.6.1 The FDDI Stations

The OPNET node model for the FDDI stations is that provided by OPNET. It is shown in Figure 3.10. However, there are changes in the process models of the FDDI stations. The differences between the process model *mac* used in the OPNET example model and the model described here are changes made to the logic to deal with the additional fields of *fddi_mac_fr* and *fddi_mac_req*. Different process models were used for the *llc_src* and the *llc_sink* processor modules in the *cbr_station* and the *vbr_station*. This has been done to be able to produce the different traffic characteristics and to file separate statistics for both. The different process models

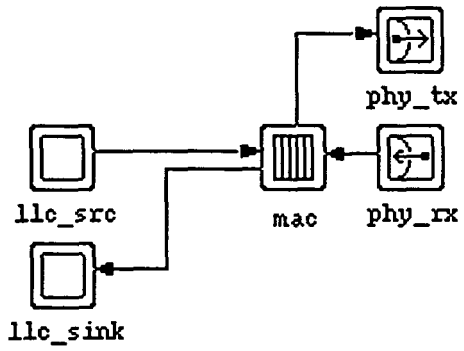


Figure 3.10: The Node Model of the FDDI Stations

are described in the remainder of this section.

fddi_gen_II: The process model *fddi_gen_II* is the process model used for *llc_src* of the *cbr_station*. It is based on the model *fddi_gen* of the OPNET FDDI example model. The most changes made correspond to the new fields in *fddi_mac_fr_II* and *fddi_mac_req_II*. Furthermore, the model attribute *traffic_dist* was introduced. *Traffic_dist* contains the function which determines the packet arrival distribution. Note that during simulations, this process model was only used to produce synchronous traffic with a constant arrival rate. This was achieved by the selection of the model parameters (see Appendix D), i.e., this model is more general than necessary to be used in this work.

fddi_sink_cbr: The process model *fddi_sink_cbr* is ruling the behavior of the processor module *llc_sink* in *cbr_station*. It is essentially the same as the original process model *fddi_sink*. It was changed to be able to record separate statistics for the CBR and VBR traffic.

fddi_gen_vbr: Probably one of the most difficult tasks in describing the behavior of VBR traffic sources is to find a suitable model to get accurate results. A recent paper [24] concludes that the widely used Poisson models are very likely to underestimate buffer requirements and cell delays. This is particularly true for the modeling of several independent sources. Independent sources modeled using the Poisson model result in a less bursty overall traffic characteristic. According to the findings in [24] the opposite is true, the aggregate traffic's burstiness is intensified.

As mentioned above, the VBR traffic in this model is produced using only one source. The principle of the process model *fddi_gen_vbr* is similar to the 'Interrupted Poisson Source' described in the literature (e.g., [25, 26]). The idea is that the traffic consists of busy and of idle periods. During busy periods, packets might be generated while no packets are produced during idle periods. The functions and parameters which determine the transition between the *idle* state and the *busy* state (and vise versa) as well as the generation of FDDI packets can be chosen using model attributes (see Section 3.3.3).

The OPNET process model *fddi_gen_vbr* is shown in Figure 3.11. It is used to produce the traffic characteristics outlined above. In the following its three states are described in more detail.

init: The *init* state comprises the same functions as described for the *init* state of *fddi_gen_II*. In addition, Marsaglia's random number generator is initialized using the two seed numbers given in the model attributes *vbr_gen_seed_I* and *vbr_gen_seed_II*. This additional random number gen-

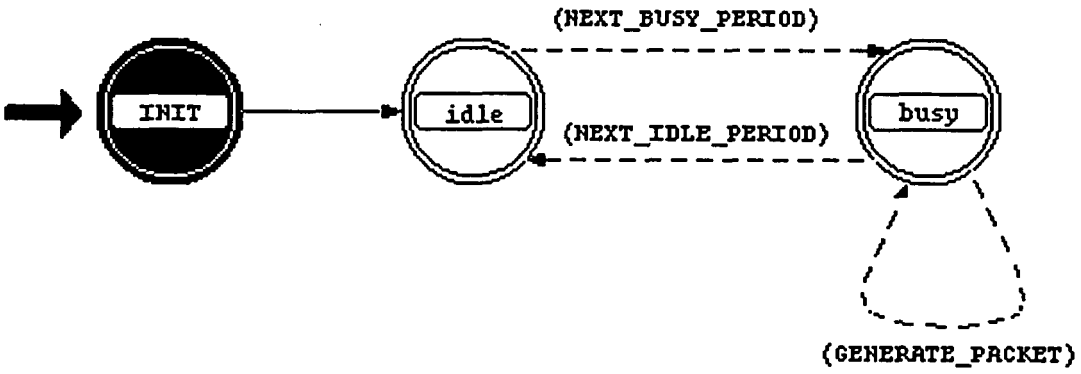


Figure 3.11: The Process Model of the VBR Traffic Generator

erator was used to be able to separate the outcome of the VBR traffic from all other random events. In OPNET, all processes using random numbers draw these numbers from the same source. This causes all random number outcomes to change if one parameter of one random distribution is changed (either direct or indirect). This was not desirable for some of the simulations where it was of interest to observe how different network configurations handle exactly the same traffic. Marsaglia's random number generator was chosen to reach this goal. It has a long range of random numbers and was shown to pass stringent tests for randomness [27].

Furthermore, the distributions for the transition from the *idle* state to the *busy* state (*busy_dist*) and the transition back from the *busy* state to the *idle* state (*idle_dist*) are set up. After initialization, *traffic_gen_vbr* makes an unconditional transition to the *idle* state. Note that for the conducted simulations only asynchronous VBR traffic was produced at this process model. This was achieved using the appropriate parameters

(see Appendix D). The described model is not restricted to this type of traffic.

idle: The *idle* state is the state representing the periods during which no packets are generated. No action is taken if the chosen arrival rate was zero. This means, there are no busy periods for this particular case. Otherwise, every time the *init* state is entered, an self interrupt is scheduled to start a new busy period. This self interrupt is determined by the distribution *busy_dist* (accessed by the distribution pointer *nxt_busy_ptr*). If the interrupt occurs, a state transition to the *busy* state is committed and another self interrupt is scheduled. This interrupt is ruled by the distribution *idle_dist* (accessed by the distribution pointer *nxt_idle_ptr*) and determines the simulation time at which the busy period ends, i.e., *fddi_gen_vbr* switches back to the *idle* state.

busy: Upon entering the *busy* state, an self interrupt is scheduled to determine the time a packet is to be produced. This is done according to the distribution *traffic_dist* (accessed by the distribution pointer *inter_dist_ptr*).

While in the *busy* state, two events of interest may occur. These two events are the arrival of a packet or the end of the busy period. In the case of a packet arrival, the same actions are taken as in *fddi_gen_II*. If the event is the end of the busy period, the pending interrupt for the next packet arrival is canceled and *traffic_gen_vbr* makes a transition back to the *idle* state.

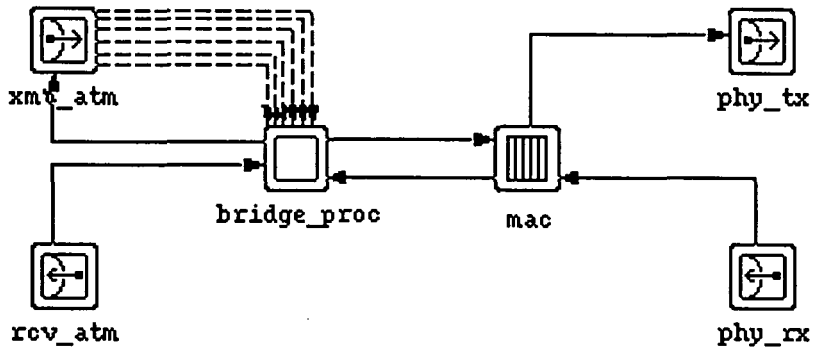


Figure 3.12: The Node Model of the ATM-FDDI Bridge

fddi_sink_vbr: The only difference between *llc_sink_vbr* and *llc_sink_cbr* is the naming of the statistics. Here the prefix vbr is used instead of cbr. The different names make it possible to distinguish the statistics for both stations.

3.6.2 The FDDI-ATM Bridge

The node model for *fddi_atm_link* is *bridge_node*. The model is shown in Figure 3.12. It is a combination of the node models of an ATM node and a FDDI station. The ATM part consists of the modules *xmt_atm*, *rcv_atm*, and *bridge_proc*. The modules *mac*, *phy_tx*, and *phy_rx* comprise the FDDI part. The *xtm_atm* and *rcv_atm* modules form the interface to the ATM switch. The process model residing in the processor module *bridge_proc* performs the functions of the ATM layer and the AAL.

As mentioned before, the protocol functions are not explicitly modeled according to their layered representation. Simulations were made using two different AAL functionalities, namely AAL type 3/4 and AAL type 5. The corresponding process models are *bridge_aal3_4_proc* and *bridge_aal5_proc*. The process models perform

three major functions:

- Segment packets submitted by the MAC entity and send the produced ATM cells to the ATM switch.
- Reassemble arriving ATM packets to reproduce the originally sent MAC frames and sent them to the processor module *mac*.
- Gather statistical information about the link between the ATM switch and the bridge.

The two process models look alike. Figure 3.13, therefore, represents both models. A more detailed description is provided below.

3.6.2.1 The Process Model bridge_aal3_4_proc. Several functions are defined in the Function Block of this model. The functions are intended to simplify the programs in the main body of the process model. Most of the functions are used to maintain a list to store information about partly received CPCS_PDUs. Details about the functions are mentioned in the state description, where appropriate. Otherwise, refer to Appendix B.

In the following each state of *bridge_aal3_4_proc* is discussed in detail.

init: The first task performed by the *init* state is to store the process model attribute *VPI_SET* in a state variable. It is later used to set the VPI field of the ATM packets sent to the ATM switch. Next, the state variable *multiplex_id* representing the multiplexing identifier (MID) of the SAR_PDU is set to zero. The identifier is used to distinguish different packet streams multiplexed into one ATM packet stream. Note that only one MID is needed for the chosen model

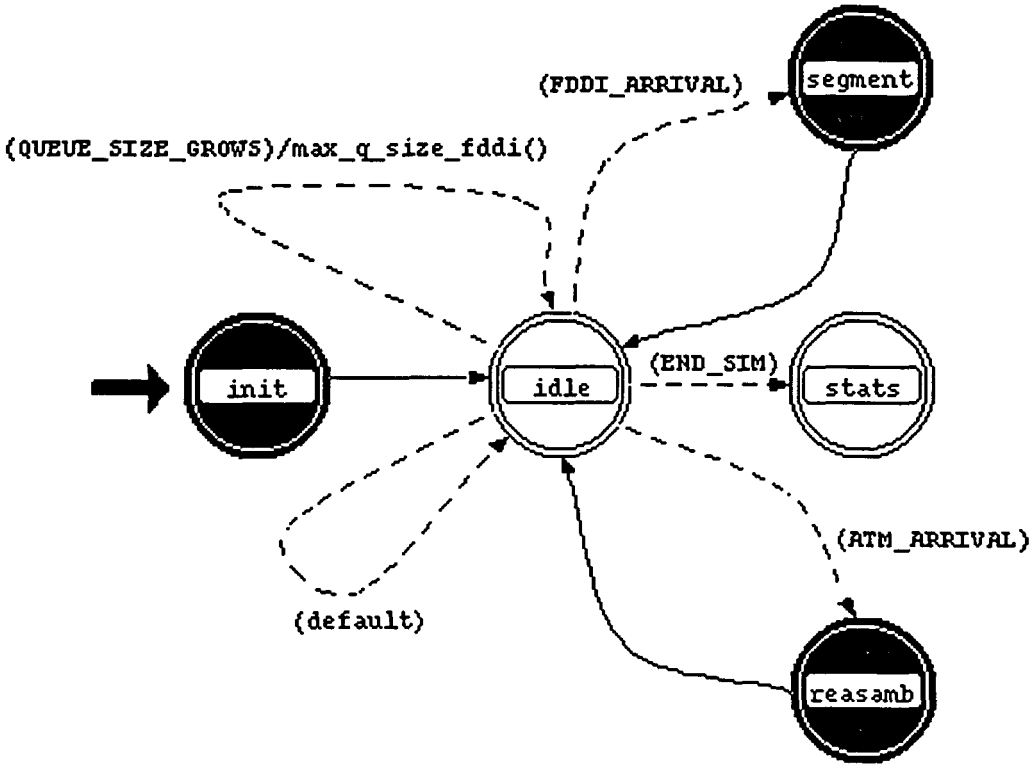


Figure 3.13: The Process Model of the FDDI-ATM Bridge

assumptions (for the connection between the two FDDI networks). Furthermore, the value of the variables *b_tag* and *e_tag* is set to zero. As was described in section 2.3.1, the only constraint for the begin_tag and end_tag fields of the CPCS_PDU packet frames is that the value of the fields has to be different for consecutive frames. The approach taken here is that the field values are increased by one for each new CPCS_PDU. Another necessary initialization is to set the start pointer of the list for partially received CPCS_PDUs to NIL. Since no cell has been sent or received, there is no entry in the list. Finally, the value of the maximum queue length for packets awaiting transmission to the

ATM switch is set to zero for the same reasons.

idle: The *idle* state is the steady state of the process model. Actions are taken for interrupts representing one of the following events:

- the arrival of a FDDI MAC frame from *mac*.
- the arrival of an ATM cell from the ATM switch.
- the grow in the length of the waiting queue at the link to the ATM switch.
- the end of the simulation.

Upon reception of a FDDI MAC frame, *bridge_aal3_4_proc* makes a transition to the *segment* state. Similarly, the reception of an ATM cell causes it to switch to the *reassemble* state. The statistic interrupt representing the grow of the waiting queue is handled the same way as described in *atm_sw_proc*. Finally, an end of simulation interrupt causes a state transition to the *stats* state.

segment: The purpose of the *segment* state is to encapsulate the received FDDI MAC frames and transform them into ATM cells according to AAL type 3/4. The process is shown in Figure 2.9 and was described in general in Section 2.3.1.

After reception of a FDDI MAC frame, the frame is encapsulated in a CPCS-PDU. The size of the padding field and the total size of the CPCS_PDU are calculated. Furthermore, a new value for the begin_tag and end_tag fields is provided. Then, the according fields of the CPCS_PDU are set. The PTI and BAsize fields of the header and the AL field of the trailer are omitted since they have no purpose for the simulation (their length is added to the *b_tag* and *e_tag* fields, respectively).

The remainder of the *segment* state is concerned with the division of the CPCS_PDUs into 44 bytes SAR_PDU payloads and their encapsulation. Logic is provided to assign the correct values to the SAR_PDU fields. The modeled fields are the segment type, the sequence number, the length indicator, and the multiplexing identifier fields. The CRC field is contained in the packet format *sar_pdu*, but no action is taken to determine an actual field value. It is set to some default value. Special care must be taken in simulating the SAR_PDU payload field. Note that the packet length of a simulated packet is rather a logical value than a physical value. Therefore, the task of segmenting a packet is not to divide a storage area into equal sized pieces. The goal to achieve is to produce the exact number of segments that correspond to the actual number of segments one would get from a real segmentation. In other words, in the simulation new packets are produced rather than an existing one divided into pieces. All the information that is contained in the original packet is sent with the last segment. This means, if a CPCS_PDU is to be segmented into N segments, than N packets are produced. The SAR_PDU payload field of the packets one through N-1 is empty, and the last packet contains a pointer to the original CPCS_PDU.

After all ATM cells are produced and sent to the ATM switch, *segment* makes an unconditional transition back to the *idle* state.

reassemble: The *reassemble* state performs the reverse functions of the *segment state*, i.e., it rebuilds a segmented CPCS_PDU, strips off the header and trailer, and sends the recovered FDDI MAC frame to *bridge_proc.* The reassembly of the CPCS_PDUs is done by means of a linked list. Each entry in the list

corresponds to one CPCS_PDU currently in the process of reassembly. A list element is identified by the multiplexing identifier (MID). Note that currently only one MID is needed (for the connection between the two FDDI networks). The more general case was assumed to be able to easily enhance the simulations, e.g., to destine traffic from the ATM nodes to the FDDI networks.

As mentioned in the description of the *segment* state, the simulation of the segmentation and reassembly process is a logical rather than a physical matter. Therefore, the reception of segments is only registered, but no actual data is stored. The registration is done by keeping track of the sequence number. In ATM, the sequence integrity is guaranteed. Thus, the sequence number of consecutive received segments of one CPCS_PDU must always be consecutive numbers. Otherwise, an error has occurred. Segments arriving out of order are discarded. Another error condition that is checked is if there is a BOM segment with a MID corresponding to a CPCS_PDU that has not been completely reassembled. In this case, the incompletely reassembled CPCS_PDU is discarded.

The original CPCS_PDU is restored using the information contained in the last segment (EOM). After the CPCS_PDU is reassembled, final error checking is performed. This is done in comparing the length of the reassembled CPCS_PDU with the length reported in the length indicator field of the CPCS_PDU. Furthermore, the begin and end tags are compared. If an error is detected, the CPCS_PDU is discarded. Otherwise, the FDDI MAC frame is decapsulated and sent to the FDDI subnetwork (*mac* in *fddi_atm_link*). In any case, after the reception of an EOM segment, the corresponding entry in the list is deleted.

If a received ATM cell is processed and all subsequent actions are taken, *bridge_aal3_4_proc* makes an unconditional transition back to the *idle* state.

stats: The *stats* state is responsible for the recording of several scalar statistics after the end of the simulation. These statistics are:

- the throughput in packets and Mb/s from the ATM switch to the FDDI subnetwork
- the maximum queue length and the mean packet delay experienced by ATM packets transmitted to the ATM switch
- the utilization of the link from the FDDI subnetwork to the ATM switch

3.6.2.2 The Process Model bridge_aal5_proc. The other process model for the processor module *bridge_proc* is *bridge_aal5_proc*. The process of segmentation and reassembly according to the specifications of AAL type 5 was described in general in section 2.3.2. No MID is defined in AAL type 5 SAR_PDUs. Therefore, only one CPCS_PDU can be reassembled at a time. This means considerable simplifications for the model. No list has to be maintained for partially received packets. As mentioned before, the process model looks exactly the same as *bridge_aal3_4_proc* and is shown in Figure 3.13. The states as coded for *bridge_aal5_proc* are described below:

init: the variables *address* (storing the value of the model attribute *VPI_SET*) and *max_quueue_length* are initialized as in *bridge_aal3_4_proc*. In addition, a new variable *num_of_segments* is set to zero. This variable contains the number of segments received from the CPCS_PDU currently in the process of reassembly. After initialization, an unconditional transition is made to the *idle* state.

idle: The *idle* state of *bridge_aal5_proc* is the same as in *bridge_aal3_4_proc.*

segment: First, a received FDDI MAC frame is encapsulated in a CPCS_PDU. Note that the CPCS_PDU format for AAL type 5 is different from the format used for AAL type 3/4 (see Figures 2.7 and 2.10). The CPCS_PDU packet fields needed for simulation are the Padding and Length fields. Their value is calculated, and the according packet fields are set. After setting of the fields, the CPCS_PDU is segmented, i.e., divided into 48 byte pieces. The Padding field assures that the CPCS_PDU is aligned on a 48 octet boundary. No SAR_PDU encapsulation has to be performed. Therefore, the segmentation process is simply:

1. get the number of needed segments (N)
2. produce N-1 ATM packets
   - set the PTI field in the cell header to zero
   - set the SAR_PDU to NIL
   - send the packets
3. produce the ATM cell for the last segment
   - set the PTI field in the cell header to one
   - set the SAR_PDU to point to the original CPCS_PDU
   - send the packet

After all cells are sent to the ATM switch, *bridge_aal5_proc* makes a transition back to the *idle* state.

reassemble: In the *reassemble* state, two different kinds of ATM packets may arrive. They are distinguished according to the value of the PTI field in the cell header.

A value of zero corresponds to a first or intermediate segment (BOM or COM in AAL type 3/4 terminology, respectively). A PTI value of one signals the reception of the last or the only segment sent (EOM or SSM, respectively).

All that has to be done is to count the number of received segments for the CPCS_PDU that is currently reassembled. The number is stored in the state variable *num_of_segments*. After reception of a cell with PTI set to one, the CPCS_PDU is reassembled. If the length reported in the length field fits the length of the reassembled frame (= no segments are gained or lost), the FDDI MAC frame is decapsulated and sent to *mac* in *bridge_nd*. Otherwise, the wrongly reassembled CPCS_PDU is discarded. Finally, an unconditional transition back to the *idle* state is made.

3.6.2.3 The Process Model fddi_mac_bridge. The process model *fddi_mac_bridge* is similar to *fddi_mac_II*. The MAC entities of the FDDI stations (*fddi_mac_II*) fetch only the packets from the ring which are destined for that particular station whereas *fddi_mac_bridge* fetches all packets which are destined for the other FDDI network. To accomplish this, the conditions for sending a received packet back on the ring (in the state *FR_RECEIVE* and for stripping the frame off the ring (transition macro *STRIP*) are modified. Furthermore, the received MAC frames are forwarded to the processor module *bridge_proc* rather than to a higher layer entity. On the other hand, packets reassembled at *bridge_proc* have to be sent on the FDDI ring. All that *fddi_mac_bridge* has to do is to queue the newly reassembled packets at the end of the queue of packets awaiting transmission to the FDDI ring and register its interest for data transmission, if not already done. This registration is needed for the token

acceleration mechanism described in [23] (see Appendix C). These actions are taken in the state *ENQUEUE*. This state was introduced to replace the state *ENCAPSULATE* used in *fddi_mac_II*. The state *ENCAPSULATE* was no longer needed since no new packets are produced at the node *fddi_atm_link*.

CHAPTER 4. CONDUCTED SIMULATIONS AND RESULTS

4.1 Overview

The final version of the developed OPNET model was presented in the previous chapter. In this chapter, an overview over the conducted simulations using this model will be given. An OPNET simulation program can be executed within the tool or from the UNIX shell. In the former case, the model attributes are entered in a simulation table. In the latter, case there are two options for communicating the model attributes to the program. These two options are using an environment file or including the parameters in the command line. The last approach was chosen for the conducted simulation runs. All necessary UNIX commands were included in c-shell script files. An example of such a shell script is shown in Figure 4.1. This offers a convenient way to run the same program with different parameters by simply changing one (or a few) parameter value(s) while leaving the rest unchanged. The drawback of this approach is that it is somewhat awkward to get an overview over the chosen values and their meaning. Therefore, an easier to read list of the chosen parameters for the simulation runs described later in this chapter is given in Appendix D.

As was described in the previous chapter, the process model *fddi_gen_vbr* was developed to simulate the behavior of a bursty traffic source. During the simula-

```
@ seed_mod=121 @ seed_vbr_I=911 @ seed_vbr_II=810@ seed_vbr_III=191 @ seed_vbr_IV=333
@ link_cap=110000000 @ vbr_arrival_rate=1500 @ cbr_arrival_rate=300
foreach atm_src_arg (1.082E-06 1.055E-06 1.029E-06 1.005E-06 9.815E-07 9.593E-07 9.38E-07 9.177E-07)
dt.sim -duration 0.4 -"top.FDDI_1.fddi_atm_link.xmt_atm[0].data rate" $link_cap -"top.FDDI_2.fddi_at
m_link.xmt_atm[0].data rate" $link_cap -verbose_sim TRUE -upd_int 0.1 -seed $seed_mod -os_file aal5_
atm_392_462Mb_fddi_ln_40_110Mb_vbr_48Mb_TIRT_0.001_sync_bw_0.5 -"top.ATM_1.src.interarrival args" $a
tm_src_arg -"top.ATM_4.src.interarrival args" 1.082E+06 -"top.ATM_2.xmt_fddi[0].data rate" $link_cap
 -"top.ATM_2.rcv_fddi[0].data rate" $link_cap -"top.ATM_3.xmt_fddi[0].data rate" $link_cap -"top.ATM
_3.rcv_fddi[0].data rate" $link_cap -"top.FDDI_1.vbr_station.llc_src.traffic_dist" exponential -"top
.FDDI_1.vbr_station.llc_src.vbr_gen_seed_I" $seed_vbr_I -"top.FDDI_1.vbr_station.llc_src.vbr_gen_see
d_II" $seed_vbr_II -"top.FDDI_1.vbr_station.llc_src.arrival rate" $vbr_arrival_rate -"top.FDDI_1.vbr
_station.llc_src.mean pk length" 32000 -top.FDDI_1.vbr_station.llc_src.idle_dist exponential -top.FD
DI_1.vbr_station.llc_src.idle_dist_arg 0.002 -top.FDDI_1.vbr_station.llc_src.busy_dist exponential -
top.FDDI_1.vbr_station.llc_src.busy_dist_arg 0.01 -top.FDDI_1.vbr_station.mac.T_Req 4.0 -"top.FDDI_1
.cbr_station.llc_src.arrival rate" 0.0 -"top.FDDI_1.cbr_station.llc_src.mean pk length" 32000 -"top.
FDDI_1.cbr_station.mac.sync bandwidth" 0.5 -top.FDDI_1.cbr_station.mac.T_Req 0.001 -"top.FDDI_1.fddi
_atm_link.mac.sync bandwidth" 0.5 -top.FDDI_1.fddi_atm_link.mac.T_Req 4.0 -"top.FDDI_1.vbr_station.l
lc_src.traffic_dist" exponential -"top.FDDI_2.vbr_station.llc_src.vbr_gen_seed_I" $seed_vbr_III -"to
p.FDDI_2.vbr_station.llc_src.vbr_gen_seed_II" $seed_vbr_IV -"top.FDDI_2.vbr_station.llc_src.arrival
rate" 0.0 -"top.FDDI_2.vbr_station.llc_src.mean pk length" 32000 -top.FDDI_2.vbr_station.llc_src.idl
e_dist exponential -top.FDDI_2.vbr_station.llc_src.idle_dist_arg 0.002 -top.FDDI_2.vbr_station.llc_s
rc.busy_dist exponential -top.FDDI_2.vbr_station.llc_src.busy_dist_arg 0.01 -"top.FDDI_2.vbr_station
.mac.T_Req" 4.0 -"top.FDDI_2.cbr_station.llc_src.arrival rate" 0.0 -"top.FDDI_2.cbr_station.llc_src.
mean pk length" 32000 -"top.FDDI_2.cbr_station.mac.sync bandwidth" 0.5 -top.FDDI_2.cbr_station.mac.T
_Req 0.001 -"top.FDDI_2.fddi_atm_link.mac.sync bandwidth" 0.5 -top.FDDI_2.fddi_atm_link.mac.T_Req 4.
0 -"top.FDDI_1.fddi_atm_link.rcv_atm[0].data rate" $link_cap -"top.FDDI_2.fddi_atm_link.rcv_atm[0].d
ata rate" $link_cap -station_latency 1E-07 -prop_delay 3.3E-06 -accelerate_token 1 -"spawn station" 1
@ link_cap = ($link_cap - 10000000)
end
```

Figure 4.1: An Example C-Shell Script

tions conducted in this work, the state transitions and cell generation followed the *exponential* distribution described in [16]. The ratio of the duration of the average idle period to the duration of the average busy period was used as a measurement for the burstiness. Increased burstiness was simulated in increasing the idle periods and increasing the probability of the packet arrival during the busy period by the same factor, i.e., the average data arrival rate remained the same. If not mentioned otherwise, the burstiness of the VBR traffic in the described simulations was chosen to be five.

It turned out that it was only possible to simulate packet arrivals in the order of 10^5–10^7 within reasonable program execution time. To be able to get useful

simulation results for events like cell loss probabilities in ATM, which occur with probabilities of 10^{-7}–10^{-9}, it is necessary to simulate about 10^{10}–10^{12} packets. The obtained results, however, are still useful to show some problems that may arise in an ATM based network but are not accurate enough to make a statement about real world resource requirements.

In the following sections, the different sets of conducted simulation runs are described in more detail and their results are discussed.

4.2 Comparison of AAL type 3/4 and AAL type 5

The principles of AAL type 3/4 and AAL type 5 were described in section 2.3 and the implementation in the model was shown in the previous chapter. This section documents simulation results for the two different AALs. The (CBR) traffic at the ATM node was chosen to be 422Mb/s and FDDI_LINE_CAP was chosen to be 80Mb/s. No CBR traffic was generated at the FDDI subnetworks. The VBR peak arrival rates were varied from 22Mb/s to 51Mb/s with a burstiness of five. For a complete list of the chosen model simulation attributes refer to Appendix D.

The Figures 4.2 and 4.3 show the maximum queue length at the ATM switch in relation to the VBR throughput for simulation runs using *bridge_aal3_4_proc* and *bridge_aal5_proc*, respectively. Not surprisingly, using the AAL type 3/4 resulted in longer maximum queue lengths. The queue lengths were slightly higher for small amounts of VBR traffic but increased with increasing VBR traffic. The differences were as high as eleven ATM packets for the above described simulations and might even be higher if more data is to be transmitted over the ATM network.

Note that exactly the same VBR traffic was generated for both simulations.

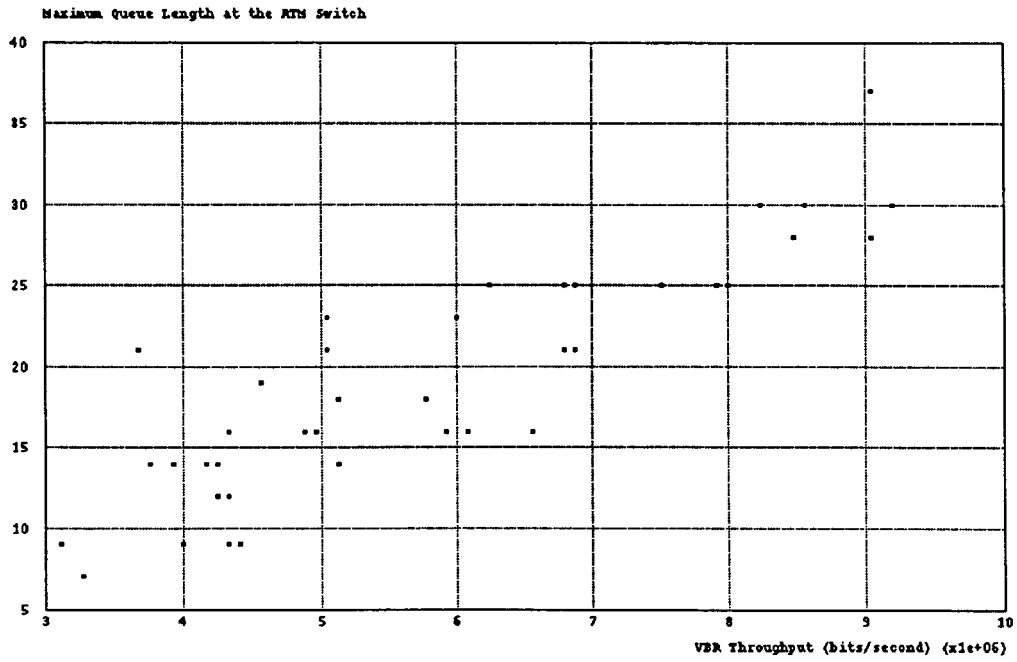


Figure 4.2: Maximum Queue Length at the ATM Switch as a Function of the VBR Throughput for CBR = 422Mb/s and VBR Peak Rate = 22-51Mb/s using AAL Type 3/4

Therefore, one can compare the outcome for every pair of programs executed with the same parameters but a different process model for *bridge_proc*. Such a pair is identified by exactly the same value for the VBR throughput since this value is determined by the FDDI throughput rather than the ATM throughput. The ATM throughput is higher for AAL type 3/4 due to the additional overhead for the SAR_PDU payload encapsulation. So, in order to transmit one 4000 byte FDDI MAC frame over an ATM network, it takes 92 cells using AAL type 3/4 compared to 84 cells using AAL type 5.

The extended encapsulation in AAL type 3/4 provides the means to pass partly reassembled CPCS_PDUs to higher layer entities (This feature was not modeled in

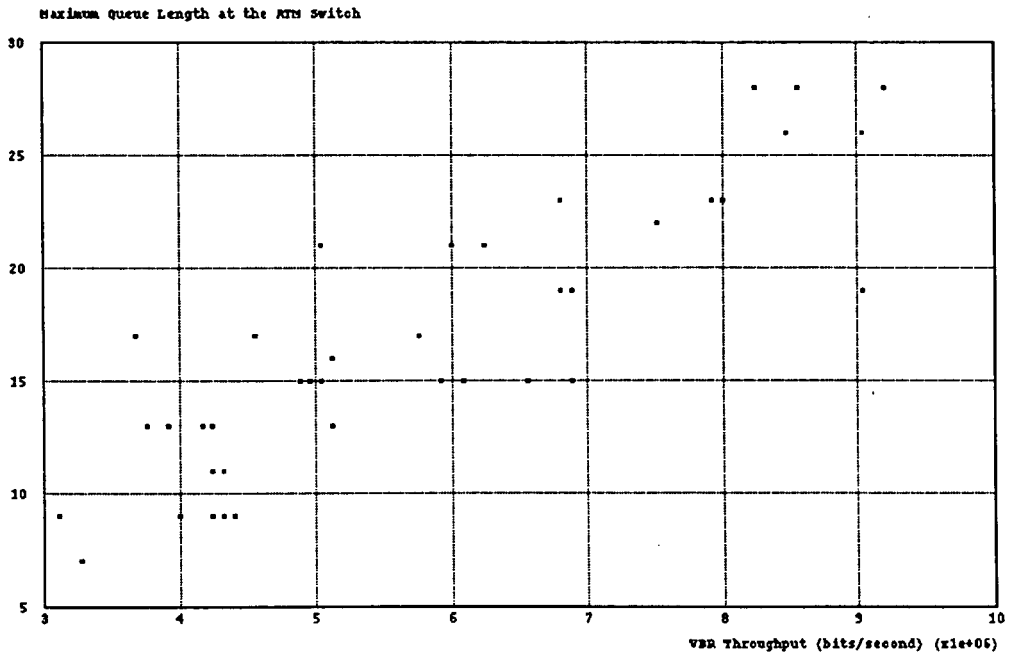


Figure 4.3: Maximum Queue Length at the ATM Switch as a Function of the VBR Throughput for CBR = 422Mb/s and VBR Peak Rate = 22-51Mb/s using AAL Type 5

this model, since a bridge could not use faulty information, anyway). These higher layer entities may then recover parts of the correctly received data. AAL type 5 doesn't provide this feature. One faulty segment results in the lost of the whole CPCS_PDU. In the case the last segment is lost, the following CPCS_PDU is lost, too. However, it was shown in [13] that the additional overhead of AAL type 3/4 pays off only for cell loss rates of approximately $2.6 * 10^{-3}$ and worse, which is far worse than what is expected from an ATM network.

An interesting feature of AAL type 5 is that it makes Selective Discarding feasible. Selective Discarding means that if one segment of a CPCS_PDU is to be discarded (due to congestion) all but the last segment of that CPCS_PDU are also discarded.

The last segment need to be kept to indicate the start of the next CPCS_PDU at the receiving entity. This strategy is very appealing in many ways. First, the remaining segments of a corrupted CPCS_PDU would have to be retransmitted even if they would arrive at their destination without error. Second, the discarding of these (worthless) segments reduces traffic during a congested period. Otherwise, they might cause congestion on subsequent switches and might lead to the discarding of other packets.

The Selective Discarding strategy is easy to implement: If a cell with AUU set to zero (in the PTI field of the cell header) is to be discarded, all subsequent cells of this VC are discarded until a cell with AUU set to one arrives. This strategy is not feasible (with respect to the computational burden) for AAL type 3/4 since the information about the beginning and the end of a segmented CPCS_PDU is contained in the data field of the ATM cell and not in the cell header [13].

The AAL type 5 approach outperformed the AAL type 3/4 approach. Furthermore, the AAL type 5 is expected to be used for LAN-LAN interconnections in BISDNs. Therefore, the AAL type 5 strategy was used for all other simulations.

4.3 Simulation of Bursty Traffic

The parameters to describe the VBR traffic should be accurate, on the one hand side, but have to be easy and fast to measure, on the other hand. Some parameters commonly used to describe bursty traffic are:

- the peak rate during the burst
- the average arrival rate

- the burst length
- the peak/average ratio
- the frequency of burst arrivals

If one adapts the widely used peak to average arrival rate ratio as a measurement for the burstiness, there are two ways to simulate increased burstiness. One can either increase the peak arrival rate (and keep the same average arrival) or one can decrease the average arrival rate, i.e., increase the duration of the idle periods.

Figure 4.3 shows the maximum queue length at the ATM switch in relation to the VBR throughput. The peak data arriving rate of the VBR traffic was varied from 22Mb/s to 51Mb/s and the ratio of idle to busy periods was chosen to be five.

Figure 4.4 represents the obtained simulation results for the same parameters accept that the ratio of idle to busy periods was increased to ten. As can be seen, the maximum queue lengths at the ATM switch are about the same for both simulations.

In another set of simulation runs, the VBR peak arrival rates were doubled, i.e., varied from 44Mb/s to 102Mb/s. The ratio of idle to busy periods was unchanged (=10). This results in the double peak and in the double average data arrival rate, i.e., the ratio of the peak arrival rate to average arrival rate remains the same. The maximum queue length in relation to the VBR throughput for this set up is shown in Figure 4.5. It is obvious to see that the maximum queue length for this set of simulation runs is considerably higher than in the two above mentioned sets of simulation runs. This suggests that the peak data arrival rate has an important impact on the network resources needed to handle a data burst. The duration of the idle periods made no difference for this simulations but might be an important

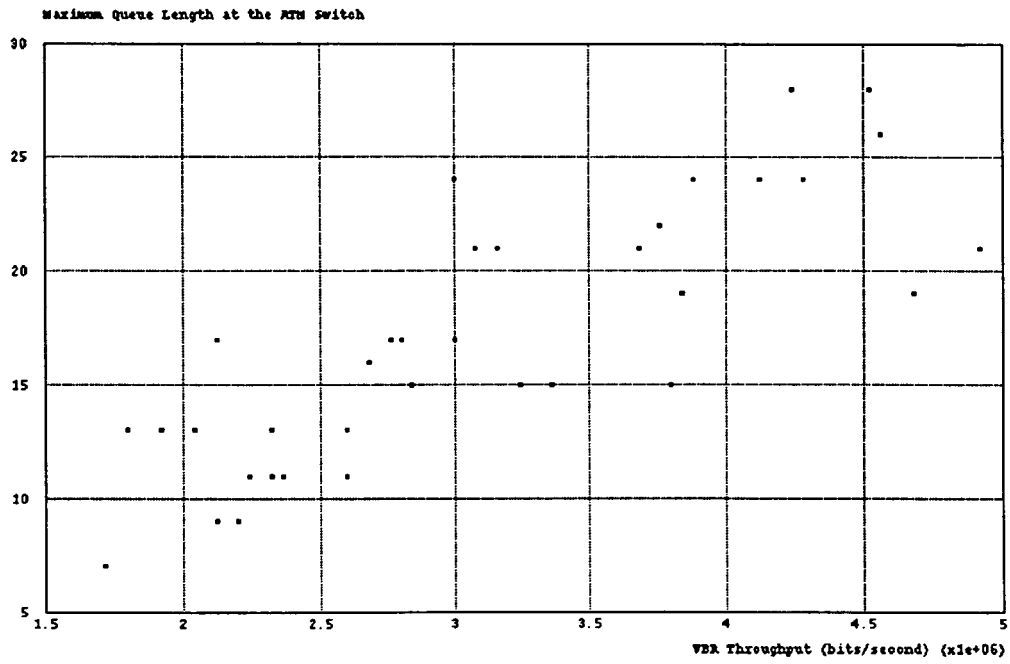


Figure 4.4: Maximum Queue Length at the ATM Switch as a Function of the VBR Throughput for CBR = 422Mb/s, VBR Peak Rate = 22-51Mb/s, and Burstiness = 10

parameter to determine the degree of statistical multiplexing that can be applied.

The mean burst length for the three simulations described above was the same. To find out more about the impact of the burst length on the maximum queue length at the ATM switch, a set of simulation programs were executed where the mean burst length was varied from 0.002–0.02 seconds. The peak arrival rate for these programs was chosen to be 60Mb/s. The maximum queue length at the ATM switch in relation to the mean burst length is shown in Figure 4.6. As can be seen in the figure, longer bursts may result in severe increase of the maximum queue lengths. These maximum queue lengths are even higher than those obtained with substantially higher VBR peak arrival rates. This confirms to findings of a study mentioned in [12]

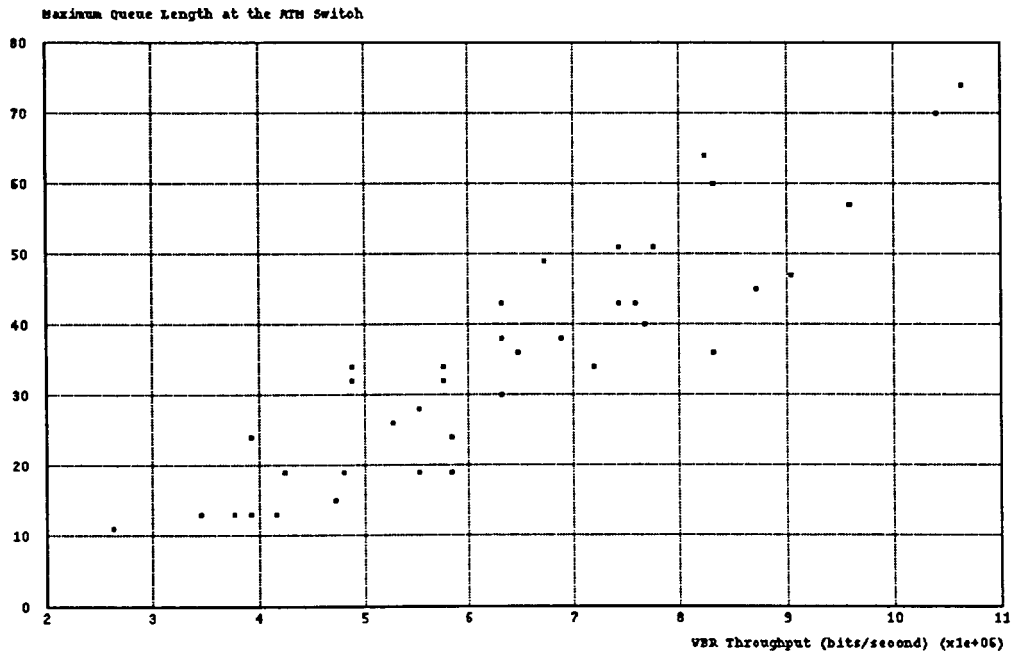


Figure 4.5: Maximum Queue Length at the ATM Switch as a Function of the VBR Throughput for CBR = 422Mb/s, VBR Peak Rate = 44-102Mb/s, and Burstiness = 10

which concludes that long intensive bursts highly increase the probability of queue overflows. The burst length and data arrival rates in the simulation discussed here result in a mean of ((0.002–0.02s)*60Mb/s) = 120–1200Mb = 15–150kbytes per burst, which is not unusual for applications like file transfer.

4.4 Combination of VBR Traffic and CBR Traffic at the ATM switch

In this set of simulation runs, only VBR traffic was generated in the FDDI subnetworks. In addition, only traffic in one direction was considered (i.e., no traffic on the VPs two and four—see Figure 3.3). The transmission capacity between the two ATM switches (SW_LINE_CAP) was chosen to be 500Mb/s. The packet arrival

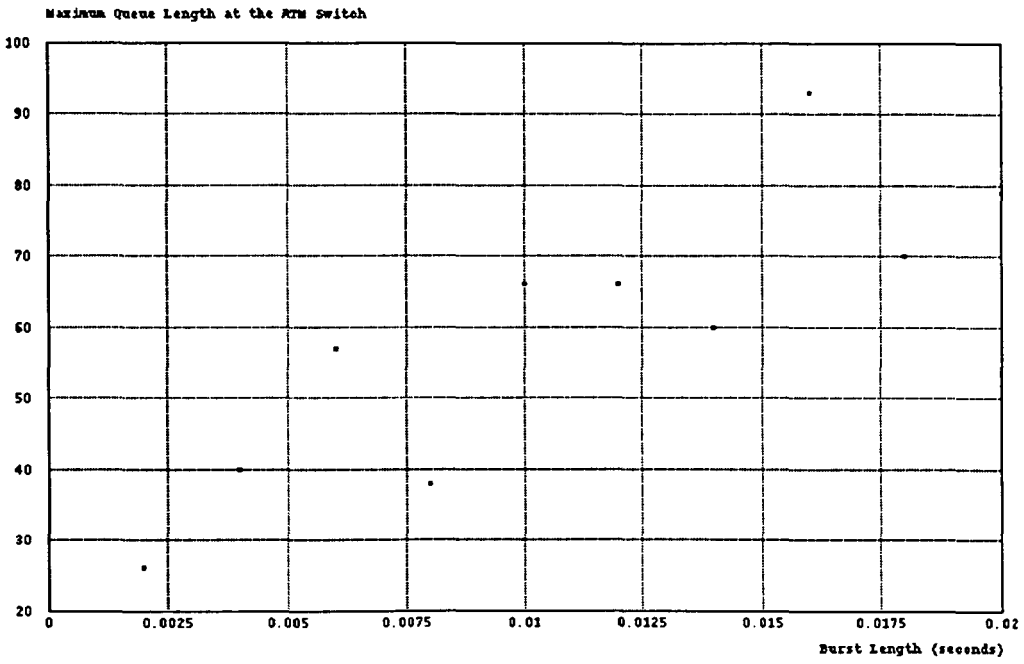


Figure 4.6: Maximum Queue Length at the ATM Switch as a Function of the Mean Burst Length for CBR = 422Mb/s and VBR Peak Rate = 60Mb/s

rates at *ATM_1* and at *vbr_station* were varied in several simulation runs. The focus was on the relationship between the CBR traffic and the VBR peak rate on the one side and SW_LINE_CAP on the other side. Note that the VBR peak rate with respect to the ATM network is determined by the capacity of the point-to-point link between the FDDI subnetwork *FDDI_1* and the ATM switch *ATM_2* (FDDI_LINE_CAP). A higher peak arrival rate at the FDDI station *vbr_station* only results in a longer burst at the ATM switch. However, the parameters were chosen so that the peak data rate at *vbr_station* was smaller than or equal to the above mentioned line capacity.

First let us consider the case where the CBR traffic and the VBR traffic together are equal to or less than SW_LINE_CAP. The maximum queue length at the ATM

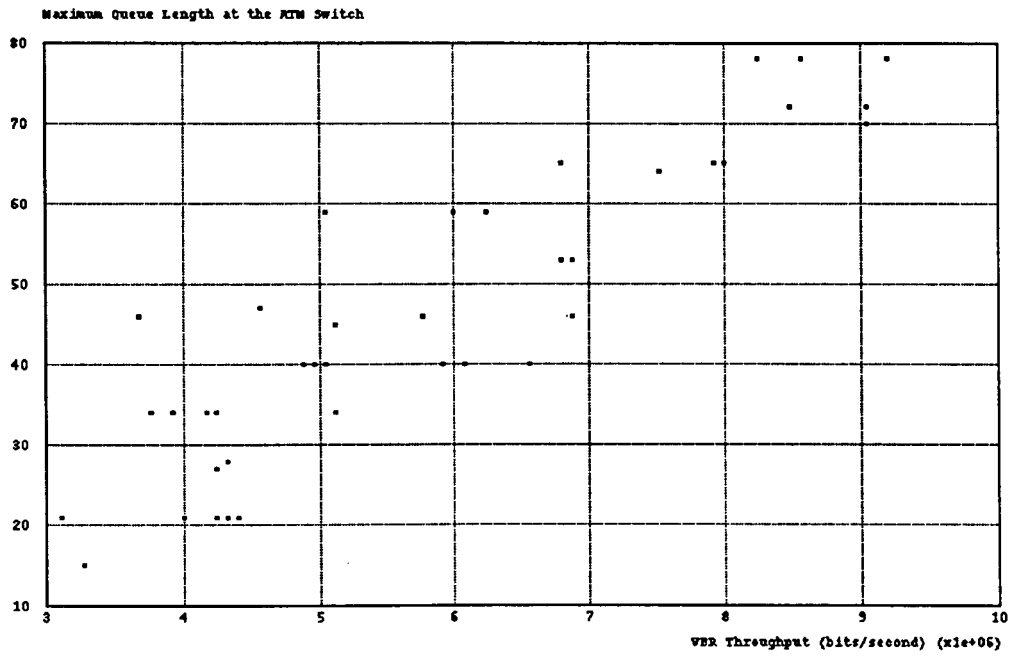


Figure 4.7: Maximum Queue Length at the ATM Switch as a Function of the VBR Throughput for CBR = 426Mb/s and VBR Peak Rate = 22-51Mb/s

switch for this particular case was always two packets, i.e., one packet arrived while a previous one had not yet finished transmission.

Figure 4.3 shows the results of an example were the combined traffic of *ATM_1* and *vbr_station* was slightly higher than SW_LINE_CAP, i.e., CBR traffic = 422Mb/s, FDDI_LINE_CAP = 80Mb/s, SW_LINE_CAP = 500Mb/s. For a complete list of the used parameters refer to Appendix D. As can be seen in the Figure, this configuration results in moderate to high queue lengths at the switch. If the CBR traffic is further increased, this results in very high queue lengths which can not be tolerated. In the case shown in Figure 4.7, the CBR was increased to 426Mb/s.

Similar simulations were made for

- CBR = 452–456Mb/s, FDDI_LINE_CAP = 50Mb/s, SW_LINE_CAP = 500Mb/s
- CBR = 442–446Mb/s, FDDI_LINE_CAP = 60Mb/s, SW_LINE_CAP = 500Mb/s
- CBR = 432–436Mb/s, FDDI_LINE_CAP = 70Mb/s, SW_LINE_CAP = 500Mb/s
- CBR = 412–416Mb/s, FDDI_LINE_CAP = 90Mb/s, SW_LINE_CAP = 500Mb/s
- CBR = 402–406Mb/s, FDDI_LINE_CAP = 100Mb/s, SW_LINE_CAP = 500Mb/s
- CBR = 392–396Mb/s, FDDI_LINE_CAP = 110Mb/s, SW_LINE_CAP = 500Mb/s

The simulations made using this parameters obtained similar results.

The results shown above strongly encourage a Peak Rate Allocation scheme for incorporation of VBR traffic into an ATM network. Peak Rate Allocation means that on any virtual connection (VC) the resources are allocated to be able to transmit data at the peak rate, i.e., this VC could as well support a CBR cell stream at the peak rate. Furthermore, the sum of the peak rates of the VCs on one transmission link is equal to or less than the maximum transmission rate of the link. ITU-TSS proposes Peak Rate Allocation schemes to be used in an early stage of ATM networks [12]. This strategy is easy to implement and easy to understand for the ATM network user [12, 28]. The link utilization, on the other hand, is poor if there is a considerable VBR portion of the overall network traffic. Although Peak Rate Allocation could guarantee that there are no queue overflows in the described model, this could be different in more complex network topologies [12]. However, this strategy is considered to be able to offer high performance in terms of cell lost probabilities.

Considerable research has been done to find solutions that achieve statistical gains in ATM networks compared to Peak Rate Allocation [12, 28, 29].

The authors in [29] claim to achieve gains of up to more than ten compared to Peak Rate Allocation while maintaining a low cell rate loss ratio ($<= 10^{-8}$). On the other hand, the proposed strategy results in higher processing overhead and higher call blocking probabilities.

Another approach, called Fast Buffer Reservation, also claims to result in 'dramatic improvements' over Peak Rate Allocation [28]. This approach would increase the hardware costs by approximately 10%. No additional signaling is necessary. The biggest drawback of this approach is that it requires non-standard ATM cell headers to put no further restrictions on the network.

4.5 Combination of VBR Traffic and CBR Traffic in the FDDI Subnetworks

As mentioned before, FDDI provides a priority scheme which gives CBR traffic a preference over VBR traffic. In the previous section, the CBR traffic in the FDDI subnetworks was assumed to be zero. In this section, the focus is on the influence of CBR traffic in the FDDI on the overall traffic to be transferred over the ATM network. To accomplish this, the simulation parameters were chosen to let the VBR peak arrival rate unchanged while the CBR data arrival rate was changed from 5–50Mb/s. The VBR traffic was set up to flow from *FDDI_1* to *FDDI_2*. The CBR traffic was set up to flow in the opposite direction. Note that the CBR traffic could as well be traffic which originates and ends on *FDDI_1*.

Figure 4.8 shows the resulting maximum queue lengths at the ATM switch *ATM_2*. It is obviously to see that, if there is a significant amount of CBR traffic, the bursty traffic is "tamed", and the maximum queue lengths shrink considerably. This

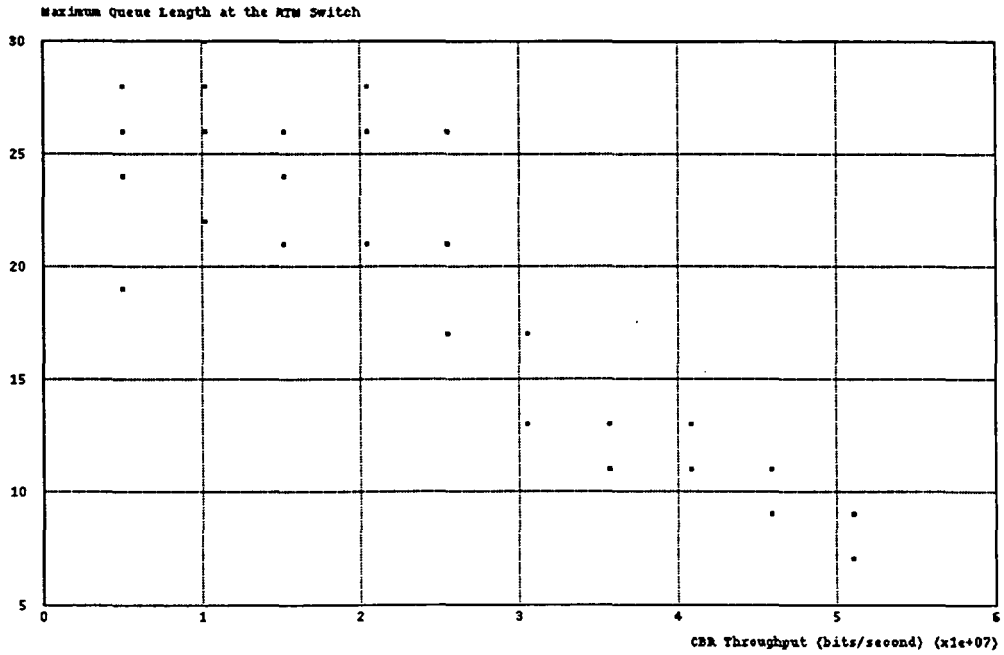


Figure 4.8: Maximum Queue Length at the ATM Switch as a Function of the CBR Throughput between the FDDI Networks for CBR (FDDI) = 5–50Mb/s, CBR(ATM) = 422Mb/s, and VBR Peak Rate = 51Mb/s

is another example that shows the difficulties to make useful statements about the VBR traffic. Even if one knew exactly the behavior of the VBR sources, no statement of their impact on the ATM network could be made since this is highly dependent on the CBR traffic on the FDDI ring. The CBR traffic is independent of the VBR traffic and may not be known in advance.

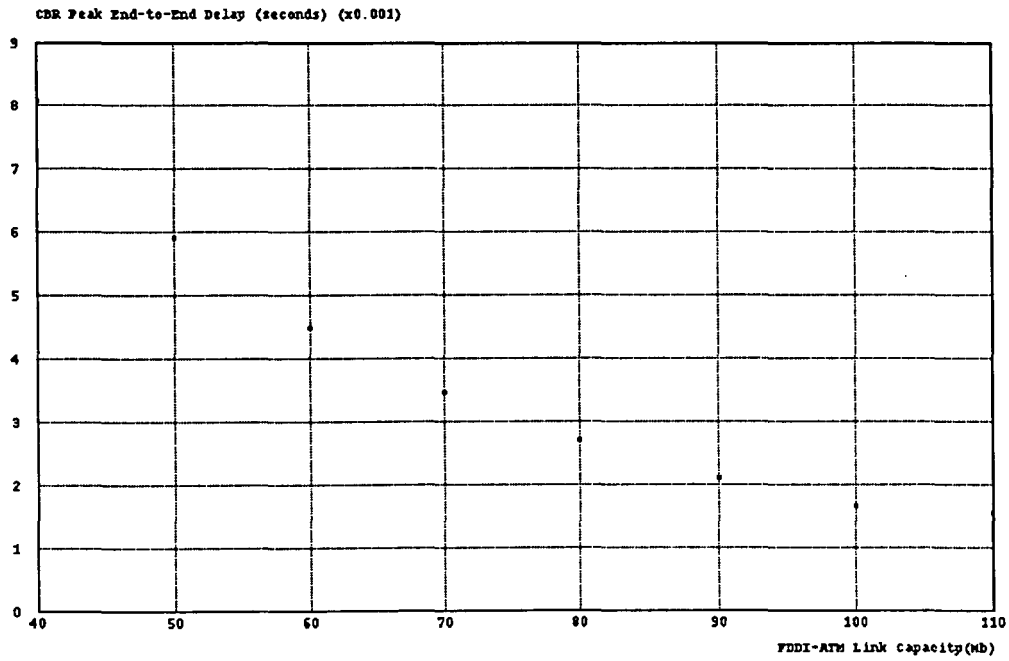


Figure 4.9: Simulation with varying Transmission Capacity of the FDDI Subnetwork to ATM Switch Communication Links

4.6 Variation of the Transmission Capacity of the FDDI Subnetwork to ATM Switch Communication Links

The maximum transmission rate of an FDDI network is 100Mb/s. To transmit 100Mb/s of FDDI traffic over a BISDN, using AAL type 5 adaptation, a transmission capacity of 110Mb/s is necessary. So, in order to be able to transmit FDDI traffic between the two FDDI subnetworks, without introducing a further queueing delay at the UNI, one has to choose FDDI_LINE_CAP to be 110Mb/s. Usually, the traffic to be exchanged between FDDIs is far less than this maximum transmission capacity. The question arises if it is justifiable to afford this high transmission capacity.

Figure 4.9 shows the maximum end-to-end delay experienced by the CBR traffic

exchanged between the two FDDI subnetworks in relation to the different values for FDDI_LINE_CAP. The VBR and CBR data was chosen to be 9.6Mb/s. The burstiness of the VBR traffic was five, i.e, the VBR peak traffic was approximately 48Mb/s. For a complete list of the simulation model attributes refer to Appendix D. Note that the traffic to be exchanged between the two FDDI subnetworks was exactly the same for all eight simulation runs. It can be seen that FDDI_LINE_CAP for the given traffic might be chosen to be less than the maximum transmission capacity without adding significantly to the maximum end-to-end delay. Choosing FDDI_LINE_CAP to be 80MB/s, for example, adds only slightly more than one millisecond to the maximum end-to-end delay of the CBR traffic. This suggests that it is worth considering a value of FDDI_LINE_CAP below the maximum transmission capacity of 110Mb/s if the timing constraints of the envisaged applications is not too stringent. If Peak Rate Allocation is used, the reduction of FDDI_LINE_CAP results in better switch-to-switch line utilizations.

CHAPTER 5. CONCLUSIONS

The combination of VBR traffic and CBR traffic puts considerable constraints on the network resources and management strategies that deal with it.

The bursty traffic is highly variable. This makes it very difficult to determine the parameters to describe the traffic accurately. For the conducted simulations, it turned out that the performance of the network was highly dependent on the peak data arrival rate (during a burst) and the burst length. By contrast, the average arrival rate, the peak to average arrival rate, and the burst interarrival rate did not significantly influence the simulation results. However, these parameters might be important to determine the possibility of statistical multiplexing. Other simulation results showed that the influence of a data burst on the ATM network can be influenced by LAN protocols and priority schemes. This fact makes it even more difficult to describe data bursts accurately. The obtained results strongly encourage the use of a Peak Rate Allocation scheme in early stage BISDNs.

The results obtained in a comparison between ATM adaptation layer type 3/4 and ATM adaptation layer type 5 suggest that ATM adaptation layer type 5 is very likely to become the standard used for LAN-LAN interconnections via ATM. This confirms to findings in [13] which were based on an analytical comparison.

The simulations concerned with the value of the transmission capacity of the

communication link between the LAN and the ATM switch showed that the transmission capacity might be less than the rate needed to transmit at the LAN speed for the transmission of moderate amounts of data within fair timing limits.

The simulations were limited by the order of packets that could be simulated within reasonable time. To overcome this shortcoming, it would be worth investigating whether strategies like Importance Sampling can be introduced to OPNET simulation models. Importance Sampling was shown to cut down execution time for calculating ATM cell blocking probabilities by orders of magnitude [30]. Since the same simulation program is executed several times with different simulation parameters and/or seeds for the random number generator to collect data for scalar statistics, executing the simulation programs with a programming scheme like PVM on a cluster of workstations would further enhance the order of packets which can be simulated.

There are still a lot of open questions concerned with the combination of VBR traffic and CBR traffic in the same network. One basic problem is to find an accurate model to describe the behavior of bursty traffic. Therefore, the feedback from early stage BISDN implementations will play an important role in solving these problems.

BIBLIOGRAPHY

[1] Rainer Händel and Manfred N. Huber. *Integrated Broadband Networks: An Introduction to ATM-Based Networks.* Addison-Wesley Publishers Ltd., Reading, Mass., 1991.

[2] Martin de Pricker. *Asynchronous Transfer Mode: Solution for Broadband ISDN.* Ellis Horwood Books in Computing Science. Series in Computer Communications and Networking. Ellis Horwood, New York, 1991.

[3] A. E. Eckberg. B-ISDN/ATM traffic and congestion control. *IEEE Network*, 6(5):28–37, September 1992.

[4] Shiro Sakata. B-ISDN multimedia workstation architecture. *IEEE Communications Magazine*, 31(8):64–67, August 1993.

[5] William Stallings. *ISDN: An Introduction*, chapter Broadband ISDN, pages 332–353. Integrated Services Digital Networks. Macmillian, New York, 1989.

[6] Adrian E. Eckberg, Bharat T. Doshi, and Richard Zoccolillo. Controlling congestion in B-ISDN/ATM: Issues and strategies. *IEEE Communications Magazine*, 29(9):64–70, September 1991.

[7] Tadanobu Okada, Hirokazu Ohnishi, and Naotaka Morita. Traffic control in asynchronous transfer mode. *IEEE Communications Magazine*, 29(9):58–62, September 1991.

[8] Setiadi Yazid and H. T. Mouftah. Congestion control methods for BISDN. *IEEE Communications Magazine*, 30(7):42–47, July 1992.

[9] James W. Roberts. Variable-bit-rate traffic control in B-ISDN. *IEEE Communications Magazine*, 29(9):50–56, September 1991.

[10] Chin-Tau Lea. What should be the goal for ATM. *IEEE Network*, 6(5):60–66, September 1992.

[11] Stephen M. Walters. A new direction for broadband ISDN. *IEEE Communications Magazine*, 29(9):39–42, September 1991.

[12] John Burgin and Dennis Dorman. Broadband ISDN resource management: The role of virtual paths. *IEEE Communications Magazine*, 29(9):44–48, September 1991.

[13] Grenville J. Armitage and Keith M. Adams. Packet reassembly during cell loss. *IEEE Network*, 7(5):26–34, September 1993.

[14] MIL 3, Inc., 3400 International Drive NW, Washington, DC 20008. *OPNET Modeling Manuel 2.0*, 1993. Release 2.4.

[15] MIL 3, Inc., 3400 International Drive NW, Washington, DC 20008. *OPNET External Interface Manuel 6.0*, 1993. Release 2.4.

[16] MIL 3, Inc., 3400 International Drive NW, Washington, DC 20008. *OPNET Simulation Kernal Manuel 5.0*, 1993. Release 2.4.

[17] MIL 3, Inc., 3400 International Drive NW, Washington, DC 20008. *OPNET Simulation Kernal Manuel 5.1*, 1993. Release 2.4.

[18] Andrew S. Tanenbaum. *Computer Networks.* Prentice-Hall, Englewood Cliffs, N. J., second edition, 1988.

[19] Scott L. Sutherland and John Burgin. B-ISDN interworking. *IEEE Communications Magazine*, 31(8):60–63, August 1993.

[20] Martin De Prycker. ATM switching on demand. *IEEE Network*, 6(2):25–28, March 1992.

[21] Achille Pattavina. Nonblocking architectures for ATM switching. *IEEE Communications Magazine*, 31(2):38–48, February 1993.

[22] Ellen Witte Zegura. Architectures for ATM switching systems. *IEEE Communications Magazine*, 31(2):28–37, February 1993.

[23] MIL 3, Inc., 3400 International Drive NW, Washington, DC 20008. *OPNET Example Models Manuel 8.0*, 1993. Release 2.4.

[24] Will E. Leland, Walter Willinger, Murad S. Taqqu, and Daniel V. Wilson. On the self-similar nature of ethernet traffic. In *SIGCOM '93: Conference Proceedings: Communication Architectures, Protocols, and Applications, September 13–17, 1993, San Francisco, California, USA*, pages 183–193. Association for Commputing Machinery, New York, 1993.

[25] Ibrahim W. Habib and Tarek N. Saadawi. Multimedia traffic characteristics in broadband networks. *IEEE Communications Magazine*, 30(7):48–54, July 1992.

[26] Daniel B. Schwartz. ATM scheduling with queueing delay predictions. In *SIGCOM '93: Conference Proceedings: Communication Architectures, Protocols, and Applications, September 13–17, 1993, San Francisco, California, USA*, pages 205–211. Association for Commputing Machinery, New York, 1993.

[27] George Marsaglia, Arif Zaman, and Wai Wan Tsang. Toward a universal random number generator. *Statistics & Probability Letters*, 8(1):35–39, January 1990.

[28] Jonathan S. Turner. Managing bandwith in ATM networks with bursty traffic. *IEEE Network*, 6(5):50–58, September 1992.

[29] Tomonori Aoyama, Ikuo Tokizawa, and Ken-Ichi Sato. ATM VP-based broadband networks for multimedia services. *IEEE Communications Magazine*, 31(4):30–39, April 1993.

[30] Qinglin Wang and Victor S. Frost. Efficient estimation of cell blocking probability for ATM systems. *IEEE/ACM Transactions on Networking*, 1(2):230–235, April 1993.

APPENDIX A. ABBREVIATIONS

AAL	ATM adaptation layer
AL	Alignment (field)
ATM	Asynchronous transfer mode
AUU	ATM-layer-user-to-ATM-layer-user indication
BAsize	Buffer allocation size
BISDN	Broadband integrated services digital network
Btag	Beginning tag
BOM	Begin of message
CAD	Computer-aided design
CAM	Computer-aided manufacturing
CBR	Constant-bit-rate
CCITT	Comité Consultatif International Télégraphique et Téléphonique
CLP	Cell loss priority
CPCS	Common part convergence sublayer
CPI	Common part identifier
CRC	Cyclic redundancy check
CS	Convergence sublayer
COM	Continuing of message
EOM	End of message
Etag	Ending tag
FDDI	Fiber-distributed data interface
GFC	Generic flow control
HDTV	High definition television
HEC	Header error control
ICI	Interface control information
ISDN	Integrated services digital network
ITU-TSS	International Telecommunications Union - Telecommunication Standardization Sector

LAN	Local area network
LI	Length indicator
LLC	Logical link control
MAC	Medium access control
MERMAID	Multimedia environment for remote multiple-attendee interactive decision-making
MID	Multiplexing identifier
NNI	Network-network interface
OAM	Operation and maintenance
PAD	Padding (field)
PDU	Protocol data unit
PM	Physical medium (sublayer)
PRM	Protocol reference model
PTI	Payload type identifier
PVM	Parallel virtual machine
RES	Reserved
SAR	Segmentation and reassembly
SDU	Service data unit
SN	Sequence number
SRC	Source
SSCS	Service specific convergence sublayer
SSM	Single-segment message
ST	Segment type
STD	State transition diagram
STM	Synchronous transfer mode
TC	Transmission convergence (sublayer)
TTRT	Target token rotation timer
UNI	User-network interface
UU	User-to-user indication
VBR	Varying-bit-rate
VC	Virtual connection
VCI	Virtual channel identifier
VLSI	Very-large-scale integration
VP	Virtual path
VPI	Virtual path identifier
XMT	Transmit

APPENDIX B. OPNET PROCESS MODEL REPORTS

Process Model Report: **atm_nd_proc**	Mon Mar 14 21:17:13 1994	Page 1 of 4
... ...		

Process Model Attributes

attribute	*value*	*type*	*default value*
VPI_SET	promoted	integer	2
VECTOR_STAT_ENABLE	promoted	toggle	disabled

Header Block

```
/* global variable */
double    atm_ete_peak_delay = 0.0;
int       scalar_stat_flag_atm_nd = 0;

/* packet stream definitions */
#define RCV_IN_STRM     0
#define SRC_IN_STRM     1
#define XMT_OUT_STRM    0

/* transition macros */
#define SRC_ARRIVAL ( op_intrpt_type () == OPC_INTRPT_STRM && \
            op_intrpt_strm () == SRC_IN_STRM )

#define RCV_ARRIVAL ( op_intrpt_type () == OPC_INTRPT_STRM && \
            op_intrpt_strm () == RCV_IN_STRM )

#define END_OF_SIM  op_intrpt_type() == OPC_INTRPT_ENDSIM
```

State Variable Block

```
Gshandle        \ete_gsh;
int             \address;
Objid           \module_id;
Boolean         \vec_stat_flag;
```

Temporary Variable Block

```
Packet  *pkptr;
double  ete_delay;
```

forced state **init**

attribute	*value*	*type*	*default value*
name	init	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **init**

```
ete_gsh = op_stat_global_reg ("ete_delay");
/* Get module ID */
module_id = op_id_self();
/* Get VPI from process attributes */
op_ima_obj_attr_get (module_id, "VPI_SET", &address);
op_ima_obj_attr_get (module_id, "VECTOR_STAT_ENABLE", &vec_stat_flag);
```

Process Model Report: **atm_nd_proc**	Mon Mar 14 21:17:13 1994	Page 2 of 4
...		
...		

transition **init -> idle**			
attribute	*value*	*type*	*default value*
name	tr_8	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **xmt**			
attribute	*value*	*type*	*default value*
name	xmt	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **xmt**

```
    /* get produced packet */
    pkptr = op_pk_get (SRC_IN_STRM);
    /* Set VPI field of packet */
    op_pk_nfd_set (pkptr, "VPI", address);
5   /* Send packet to transmitter */
    op_pk_send(pkptr, XMT_OUT_STRM);
```

transition **xmt -> idle**			
attribute	*value*	*type*	*default value*
name	tr_1	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **idle**			
attribute	*value*	*type*	*default value*
name	idle	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

transition **idle -> xmt**			
attribute	*value*	*type*	*default value*
name	tr_0	string	tr
condition	SRC_ARRIVAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

Process Model Report: **atm_nd_proc**	Mon Mar 14 21:17:13 1994	Page 3 of 4
... ...		

transition **idle -> idle**			
attribute	*value*	*type*	*default value*
name	tr_2	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> rcv**			
attribute	*value*	*type*	*default value*
name	tr_3	string	tr
condition	RCV_ARRIVAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> stats**			
attribute	*value*	*type*	*default value*
name	tr_9	string	tr
condition	END_OF_SIM	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **rcv**			
attribute	*value*	*type*	*default value*
name	rcv	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **rcv**

```
pkptr = op_pk_get (RCV_IN_STRM);
ete_delay = op_sim_time () - op_pk_creation_time_get (pkptr);
/* keep track of peak delay value */
if (ete_delay > atm_ete_peak_delay)
    atm_ete_peak_delay = ete_delay;
/* If the vector statistic is enabled record end-to-end delay */
if (vec_stat_flag == OPC_TRUE)
    op_stat_global_write (ete_gsh, ete_delay);
op_pk_destroy (pkptr);
```

transition **rcv -> idle**			
attribute	*value*	*type*	*default value*
name	tr_7	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

Process Model Report: **atm_nd_proc**	Mon Mar 14 21:17:14 1994	Page 4 of 4
... ...		

unforced state **stats**			
attribute	*value*	*type*	*default value*
name	stats	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

enter execs **stats**

```
/* At end of simulation, scalar statistics are written out */
/* Only one node has to do this */
if (!scalar_stat_flag_atm_nd){
    /* Set flag to signal that the scalar statistics are written out */
5   scalar_stat_flag_atm_nd = 1;
    op_stat_scalar_write ("ATM_Peak End-to-End Delay (sec.)", atm_ete_peak_delay);
}
```

Process Model Report: **atm_sw_proc**	Mon Mar 14 21:18:23 1994	Page 1 of 4
...		
...		

Process Model Attributes

attribute	*value*	*type*	*default value*
VPI_ATM_LOCAL	promoted	integer	0
VPI_ATM_REMOTE	promoted	integer	1
VPI_FDDI_LOCAL	promoted	integer	2
VPI_FDDI_REMOTE	promoted	integer	3
STAT_ENABLE	promoted	toggle	disabled

Header Block

```
/* Transition macros */
#define PK_ARRIVAL (op_intrpt_type () == OPC_INTRPT_STRM)

#define QUEUE_SIZE_GROWS (op_intrpt_type () == OPC_INTRPT_STAT && \
                          op_intrpt_stat () == QUEUE_LENGTH_STAT)

#define END_SIM (op_intrpt_type () == OPC_INTRPT_ENDSIM)

/* Packet Streams */
#define FDDI_OUTPUT_STRM          0
#define ATM_NODE_OUTPUT_STRM      1
#define ATM_SWITCH_OUTPUT_STRM    2

/* Statistic inputs */
#define MEAN_DELAY_STAT       0
#define MEAN_PKSIZE_STAT      1
#define QUEUE_LENGTH_STAT     2
#define PK_THRUPUT_STAT       3
#define UTILIZATION_STAT      4
#define BIT_THRUPUT_STAT      5
```

State Variable Block

```
Objid     \module_id;
int       \vpi_pk, \vpi1, \vpi2, \vpi3, \vpi4;
double    \max_queue_length;
```

Temporary Variable Block

```
Packet      *pkptr;
Boolean     scalar_write_flag;
void        max_q_size_sw ();
void        record_stats_sw ();
```

Function Block

```
/*****************************************************************/
/*                    max_q_size_sw ()                           */
/*****************************************************************/
/* This function dtermines the maximum queue of the ATM          */
/* transmiter channel length during the simulation run           */
/*****************************************************************/

void max_q_size_sw ()
    {
    double    queue_length;
```

...
...

```
    /* Read the new queue size */
    queue_length = op_stat_local_read (QUEUE_LENGTH_STAT);
    /* Check if it is a new maximum */
15  if (queue_length > max_queue_length)
        max_queue_length = queue_length;
}
```

unforced state **idle**			
attribute	*value*	*type*	*default value*
name	idle	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

transition **idle -> route_pk**			
attribute	*value*	*type*	*default value*
name	tr_0	string	tr
condition	PK_ARRIVAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> idle**			
attribute	*value*	*type*	*default value*
name	tr_10	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> idle**			
attribute	*value*	*type*	*default value*
name	tr_12	string	tr
condition	QUEUE_SIZE_GROWS	string	
executive	max_q_size_sw()	string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> stats**			
attribute	*value*	*type*	*default value*
name	tr_13	string	tr
condition	END_SIM	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **route_pk**			
attribute	*value*	*type*	*default value*
name	route_pk	string	st

Process Model Report: **atm_sw_proc**	Mon Mar 14 21:18:24 1994	Page 3 of 4
... ...		

enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **route_pk**

```
/* Get arrived packet */
pkptr = op_pk_get (op_intrpt_strm ());
/* Get VPI from packet field VPI */
op_pk_nfd_get (pkptr, "VPI", &vpi_pk);
/* Send packet to according output stream */
/* If the packet is from the remote FDDI-node */
/* send it to the local FDDI-node */
if (vpi_pk == vpi4)
    op_pk_send (pkptr, FDDI_OUTPUT_STRM);
/* If the packet is from the remote ATM-node */
/* send it to the local ATM-node */
if (vpi_pk == vpi2)
    op_pk_send (pkptr, ATM_NODE_OUTPUT_STRM);
/* If the packet is from the local FDDI- or ATM-node */
/* send it to the other ATM-switch */
if (vpi_pk == vpi3 || vpi_pk == vpi1)
    op_pk_send (pkptr, ATM_SWITCH_OUTPUT_STRM);
```

transition **route_pk -> idle**

attribute	*value*	*type*	*default value*
name	tr_4	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **initial**

attribute	*value*	*type*	*default value*
name	initial	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **initial**

```
/* Get module ID */
module_id = op_id_self();
/* Get the VPIs from the process attributes */
op_ima_obj_attr_get (module_id, "VPI_ATM_LOCAL", &vpi1);
op_ima_obj_attr_get (module_id, "VPI_ATM_REMOTE", &vpi2);
op_ima_obj_attr_get (module_id, "VPI_FDDI_LOCAL", &vpi3);
op_ima_obj_attr_get (module_id, "VPI_FDDI_REMOTE", &vpi4);

/* Initialize max queue length */
max_queue_length = 0.0;
```

Process Model Report: **atm_sw_proc**	Mon Mar 14 21:18:24 1994	Page 4 of 4
... ...		

transition **Initial -> idle**			
attribute	*value*	*type*	*default value*
name	tr_6	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **stats**			
attribute	*value*	*type*	*default value*
name	stats	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

enter execs **stats**

```
/* Record scalar statistics */
op_ima_obj_attr_get (module_id, "STAT_ENABLE", &scalar_write_flag);
if (scalar_write_flag == OPC_TRUE){
    /* Record final statistics */
    op_stat_scalar_write("ATM Switch Throughput (Mb/s)", (op_stat_local_read (BIT_THRUPUT_STAT) / 1000000));
    op_stat_scalar_write("Mean Packet Delay at ATM Switch", op_stat_local_read (MEAN_DELAY_STAT));
    op_stat_scalar_write("Mean Queue Length at ATM Switch", op_stat_local_read (MEAN_PKSIZE_STAT));
    op_stat_scalar_write("Maximum Queue Length at ATM Switch", max_queue_length);
    op_stat_scalar_write("Packet Throughput at ATM Switch", op_stat_local_read (PK_THRUPUT_STAT));
    op_stat_scalar_write("Utilization of Switch to Switch Link", op_stat_local_read (UTILIZATION_STAT));
}
```

Process Model Report: **bridge_aal3_4_proc**	Mon Mar 14 21:19:10 1994	Page 1 of 14
...		
...		

Process Model Attributes

attribute	*value*	*type*	*default value*
VPI_SET	promoted	integer	2
STAT_ENABLE	promoted	toggle	disabled

Header Block

```
/* packet stream definitions */
#define FDDI_IN_STRM        0
#define ATM_IN_STRM         1
#define FDDI_OUT_STRM       0
#define ATM_OUT_STRM        1

/* Statistical inputs definition */
#define MEAN_DELAY_STAT     0
#define MEAN_PKSIZE_STAT    1
#define QUEUE_LENGTH_STAT   2
#define PK_THRUPUT_STAT     3
#define UTILIZATION_STAT    4
#define BIT_THRUPUT_STAT    5

/* Packet size definitions */
#define ATM_PK_SIZE             424
#define SAR_PK_SIZE             384
#define SAR_PDU_PAYLOAD_SIZE    44
#define HEADER_SIZE             4
#define TRAILER_SIZE            4

/* Sgment type definitions */
#define BOM     10
#define COM     00
#define EOM     01
#define SSM     11

/* transition macros */
#define FDDI_ARRIVAL (op_intrpt_type () == OPC_INTRPT_STRM && \
        op_intrpt_strm () == FDDI_IN_STRM)

#define ATM_ARRIVAL (op_intrpt_type () == OPC_INTRPT_STRM && \
        op_intrpt_strm () == ATM_IN_STRM)

#define QUEUE_SIZE_GROWS (op_intrpt_type () == OPC_INTRPT_STAT && \
                op_intrpt_stat () == QUEUE_LENGTH_STAT)

#define END_SIM (op_intrpt_type () == OPC_INTRPT_ENDSIM)

typedef struct rcv_fddi_pk{
        int                     pk_id;
        int                     last_seq_number;
        struct rcv_fddi_pk      *prev_pk;
        struct rcv_fddi_pk      *next_pk;
}   pk_list;
```

Process Model Report: **bridge_aal3_4_proc**	Mon Mar 14 21:19:10 1994	Page 2 of 14
... ...		

State Variable Block

```
      int         \b_tag, \e_tag;
      int         \address, \multiplex_id;
      Objid       \module_id;
      pk_list*    \start_ptr;
 5    double      \max_queue_length;
      Boolean     \stat_flag;
```

Temporary Variable Block

```
      Packet   *pkptr, *cpcs_pdu_pkptr, *sar_pkptr;
      double   ete_delay_seg, ete_delay_fddi;
      int      rcv_b_tag, rcv_e_tag;
      int      fddi_pk_size;
 5    int      cpcs_pdu_payload_size;
      int      cpcs_pdu_total_pk_size;
      int      num_of_atm_pks;
      int      last_pk_payload_size;
      int      n;
10    int      seg_type, length_ind;
      int      mult_id, seq_number;
      int      pk_length;
      int      pad;
      Packet   *payload_ptr;
15    pk_list  *list_ptr;

      void     atm_pk_send(Packet*);
      void     set_sar_pk_fds(Packet*, int, int, int, Packet*, int);
      void     set_sar_pk_fds_ssm(Packet*, Packet*, int);
20    void     add_pk_to_list (int, int);
      pk_list  *find_pk (int);
      void     del_pk_from_list (pk_list*);
      void     max_q_size_fddi ();
      void     record_stats_fddi ();
25
```

Function Block

```
      /*****************************************************************/
      /*                        atm_pk_send()                          */
      /*****************************************************************/
      /* Function gets a pointer to a SAR-PDU                          */
 5    /* - encapsulate the SAR-PDU in a ATM packet                     */
      /* - sends the packet                                            */
      /*****************************************************************/

      void atm_pk_send (fkt_sar_pkptr)
10        Packet    *fkt_sar_pkptr;
          {
          Packet    *atm_pkptr;

          atm_pkptr = op_pk_create_fmt("atm_pk");
15        /* copy SAR packet to ATM packet data field */
          op_pk_nfd_set (atm_pkptr, "data", fkt_sar_pkptr);
          /* Set VPI field of packet */
```

```
    op_pk_nfd_set (atm_pkptr, "VPI", address);
    /* Set total ATM packet size */
    op_pk_total_size_set(atm_pkptr, ATM_PK_SIZE);
    /* Send packet to transmitter */
    op_pk_send(atm_pkptr, ATM_OUT_STRM);
}


/*************************************************************/
/*                      set_sar_pk_fds()                     */
/*************************************************************/
/* Function gets a pointer to a SAR-PDU and according field  */
/* values for segmented packets                              */
/* - fills the named fields of the specified SAR-PDU         */
/*  with the submitted values                                */
/*************************************************************/

void set_sar_pk_fds (sarptr, seg_type, seq_number, mid, payload_ptr, pk_length)
    Packet    *sarptr, *payload_ptr;
    int       seg_type, seq_number, mid, pk_length;
    {
    /* Set the fields of SAR packet */
    op_pk_nfd_set (sarptr, "segment_type", seg_type);
    op_pk_nfd_set (sarptr, "sequence_number", seq_number);
    op_pk_nfd_set (sarptr, "multiplexing_id", mid);
    /* Only the last segment gets the original data */
    if (seg_type == EOM)
        op_pk_nfd_set (sarptr, "sar_pdu_payload", payload_ptr);
    op_pk_nfd_set (sarptr, "length_indicator", pk_length);
    op_pk_total_size_set (sarptr, SAR_PK_SIZE);
}


/*************************************************************/
/*                      set_sar_pk_fds_ssm()                 */
/*************************************************************/
/* Function gets a pointer to a SAR-PDU and according field  */
/* values for single packets                                 */
/* - fills the named fields of the specified SAR-PDU         */
/*  with the submitted values                                */
/*************************************************************/

void set_sar_pk_fds_ssm (sarptr, payload_ptr, pk_length)
    Packet    *sarptr, *payload_ptr;
    int       pk_length;
    {
    /* Set the fields of SAR packet */
    op_pk_nfd_set (sarptr, "segment_type", SSM);
    op_pk_nfd_set (sarptr, "sar_pdu_payload", payload_ptr);
    op_pk_nfd_set (sarptr, "length_indicator", pk_length);
    op_pk_total_size_set (sarptr, SAR_PK_SIZE);
}


/*************************************************************/
/*                      find_end_of_list()                   */
/*************************************************************/
/* Gets a pointer to a list element                          */
/* - if the list element is the last list element            */
```

```
    /*  it returns the the pointer it got as an argument                    */
    /* - otherwise the function calls itself with the                       */
    /*  pointer to the next element in the list                             */
80  /*********************************************************************/

    pk_list *find_end_of_list (list_ptr)
        pk_list *list_ptr;
        {
85      pk_list *last_pk;
        /* debugging message */
        if (op_prg_odb_ltrace_active("list_test"))
            printf("Received order to find the end of the list - struct_ptr= %d\n", list_ptr );
        /* Check if it is the last element in the list */
90      if (list_ptr->next_pk == OPC_NIL){
            /* debugging message */
            if (op_prg_odb_ltrace_active("list_test"))
                printf("Found end of the list returned pointer= %d\n", list_ptr);
            /* Return the pointer to the last list element */
95          return list_ptr;
        }
        else
            {
            /* debugging message */
100         if (op_prg_odb_ltrace_active("list_test"))
                printf("Recursiv call structure_ptr= %d\n", list_ptr->next_pk);
            /* Recursiv call with pointer to the next list element as argument */
            last_pk = find_end_of_list (list_ptr->next_pk);
            /* debugging message */
105         if (op_prg_odb_ltrace_active("list_test"))
                printf("Recursiv call found end of list - returnrd %d\n", last_pk);
            /* Return pointer to the last list element */
        return last_pk;
        }
110 }


    /*********************************************************************/
    /*                        add_pk_to_list()                           */
115 /*********************************************************************/
    /* fills in structure for the new list element                        */
    /* and adds the new list element as the last list element             */
    /*********************************************************************/

120 void add_pk_to_list (pk_id, seq_number)
        int   pk_id, seq_number;
        {
        pk_list *last_pk, *struct_ptr;

125     /* debugging message */
        if (op_prg_odb_ltrace_active("list_test"))
            printf("Received order to add packet to list pk_id: %d seq_number: %d\n", pk_id, seq_number);
        /* Allocate memory for the new list element */
        struct_ptr = (pk_list *) malloc (sizeof(pk_list));
130     /* Fill in variables of the new list element */
        struct_ptr->pk_id = pk_id;
        struct_ptr->last_seq_number = seq_number;
        /* The new list element is now the last one in the list */
        struct_ptr->next_pk = OPC_NIL;
135     /* debugging message */
```

```
...
...
```

```
    if (op_prg_odb_ltrace_active("list_test")){
        printf("Filled in values in structure \n");
        printf("pk_id = %d seq_no = %d\n",struct_ptr->pk_id, struct_ptr->last_seq_number);
    printf("Assigned pointer to structure = %d\n",struct_ptr);
140 }
    /* Check if the list is empty */
    if (start_ptr == OPC_NIL){
        /* debugging message */
        if (op_prg_odb_ltrace_active("list_test"))
145         printf("I am the only packet - start_pointer was OPV_NIL\n");
        /* The packet is the first one */
        /* - set the pointer to the first list element */
        start_ptr = struct_ptr;
        /* - set the pointer to the next list element to NULL */
150     /*  (since there is no other element in the list) */
        struct_ptr->prev_pk = OPC_NIL;
    }
    else{
        /* The list is not empty */
155     /* debugging message */
        if (op_prg_odb_ltrace_active("list_test")){
            printf("There is a packet in the list - called function to find the end of the list\n");
            printf("Parameter start_ptr= %d\n", start_ptr);
        }
160     /* Find the end of the list */
        last_pk = find_end_of_list (start_ptr);
        /* Register our list element as next list element */
        last_pk->next_pk = struct_ptr;
        /* Register the former end of the list as our precessor */
165     struct_ptr->prev_pk = last_pk;
    }
}


170 /*************************************************************/
    /*                      find_pk_in_list()                    */
    /*************************************************************/
    /* Checks if the specified list element                      */
    /* belongs to the specified packed identifier                */
175 /* - YES = return pointer to the list element                */
    /* - NO  = recursiv call (check next list element)           */
    /*************************************************************/

    pk_list *find_pk_in_list (list_ptr, pk_id)
180     pk_list *list_ptr;
        int     pk_id;
        {
        pk_list *element_ptr;

185     /* debugging message */
        if (op_prg_odb_ltrace_active("list_test"))
            printf("Got order to find packet in the list\n pk_id= %d structure_ptr= %d\n", pk_id, list_ptr);
        /* Check if the Pointer is NULL - packet is not in the list */
        if (list_ptr == OPC_NIL){
190         /* debugging message */
            if (op_prg_odb_ltrace_active("list_test"))
                printf("Could not find packet - reached end of the list \n");
            return OPC_NIL;
        }
```

```
195     else{
            /* Check if this is the element we are looking for */
            if (list_ptr->pk_id == pk_id){
                /* debugging message */
                if (op_prg_odb_ltrace_active("list_test"))
200                 printf("Found packet - returned structure pointer %d\n", list_ptr);
                /* Return the pointer to the list element */
                return list_ptr;
            }
            else{
205             /* debugging message */
                if (op_prg_odb_ltrace_active("list_test"))
                    printf("Recursiv call - struct_ptr = %d pk_id = %d\n", list_ptr->next_pk, pk_id);
                /* Check the next list element */
                element_ptr = find_pk_in_list (list_ptr->next_pk, pk_id);
210             /* debugging message */
                if (op_prg_odb_ltrace_active("list_test"))
                    printf("Recursiv call found packet - returned pointer = %d\n", element_ptr);
                return element_ptr;
            }
215     }
}


/*************************************************************/
220 /*                      find_pk()                            */
/*************************************************************/
/* Checks if there is a list entry for the given pk_id        */
/* - YES = return the pointer to the list element             */
/* - NO  = return NULL pointer                                */
225 /*************************************************************/

pk_list *find_pk (pk_id)
    int   pk_id;
    {
230 pk_list *list_ptr;

    /* debugging message */
    if (op_prg_odb_ltrace_active("list_test"))
        printf("Got order to find packet pk_id= %d\n", pk_id);
235 /* Check if the list is empty */
  if (start_ptr == OPC_NIL){
        /* debugging message */
        if (op_prg_odb_ltrace_active("list_test"))
            printf("Could not find packet - list is empty \n");
240     /* Return NULL pointer - no element in the list */
        return OPC_NIL;
    }
    else{
        /* debugging message */
245     if (op_prg_odb_ltrace_active("list_test"))
            printf("Called function find_pk_in_list\n - parameter pk_id= %d structure pointer= %d\n", p
        /* Since the list is not empty - search for our packet */
        list_ptr = find_pk_in_list (start_ptr, pk_id);
        /* debugging message */
250     if (op_prg_odb_ltrace_active("list_test"))
            printf("Procedure found packet - returned structure_ptr = %d\n", list_ptr);
        return list_ptr;
    }
```

```
}

/*************************************************************/
/*                      del_pk_from_list()                   */
/*************************************************************/
/* Deletes a list element from the list                      */
/* - update entries in the neighboring list elements         */
/* - free the allocated memory                               */
/*************************************************************/

void del_pk_from_list (list_ptr)
    pk_list *list_ptr;
    {
    /* debugging message */
    if (op_prg_odb_ltrace_active("list_test"))
        printf("Received order to delete packet from list - list_ptr= %d\n", list_ptr);
    /* Check if specified packet is the first packet in the list */
    if (list_ptr->prev_pk == OPC_NIL){
        /* debugging message */
        if (op_prg_odb_ltrace_active("list_test"))
            printf("I was the first packet in the list - set start_ptr = %d\n", list_ptr->next_pk);
        /* Update start_ptr (pointer to the first list element) */
        start_ptr = list_ptr->next_pk;
    }
    else{
        /* debugging message */
        if (op_prg_odb_ltrace_active("list_test"))
            printf("Set the next_pk_ptr of list %d to %d\n", list_ptr->prev_pk, list_ptr->next_pk);
        /* Update our precessor */
        list_ptr->prev_pk->next_pk = list_ptr->next_pk;
    }
    /* Check if specified packet is not the last packet in the list */
    if (list_ptr->next_pk != OPC_NIL){
        /* debugging message */
        if (op_prg_odb_ltrace_active("list_test"))
            printf("I was not the last packet in the list\n");
            printf("Therefore I set prev_pk_ptr of %d to %d\n", list_ptr->next_pk, list_ptr->prev_pk);
        /* Update our successor */
        list_ptr->next_pk->prev_pk = list_ptr->prev_pk;
    }
    /* deallocate the memory */
    free(list_ptr);
}

/*************************************************************/
/*                      max_q_size_fddi ()                   */
/*************************************************************/
/* This function dtermines the maximum queue of the ATM      */
/* transmiter channel length during the simulation run       */
/*************************************************************/

void max_q_size_fddi ()
    {
  double      queue_length;

    /* Read the new queue size */
    queue_length = op_stat_local_read (QUEUE_LENGTH_STAT);
    /* Check if it is a new maximum */
```

...
...

```
    if (queue_length > max_queue_length)
        max_queue_length = queue_length;
315 }
```

forced state **init**

attribute	*value*	*type*	*default value*
name	init	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **init**

```
    /* Get module ID */
    module_id = op_id_self();
    /* Get VPI from process attributes */
    op_ima_obj_attr_get (module_id, "VPI_SET", &address);
 5  /* Get the value of the statistics flag */
    op_ima_obj_attr_get (module_id, "STAT_ENABLE", &stat_flag);
    /* Initialize the value of the begin and end tag */
    b_tag = 0;
    e_tag = 0;
10  /* Initialize multiplexing identifier */
    multiplex_id = 0;
    /* Initialize pointer to list of partially received fddi packets */
    start_ptr = OPC_NIL;
    /* Initialize maximum queue size */
15  max_queue_length = 0;
```

transition **init -> idle**

attribute	*value*	*type*	*default value*
name	tr_8	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **segment**

attribute	*value*	*type*	*default value*
name	segment	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **segment**

```
    /* Get the FDDI MAC frame */
    pkptr = op_pk_get (FDDI_IN_STRM);
    /* Get the size of the packet */
    fddi_pk_size = op_pk_total_size_get (pkptr);
 5  /* Change the total size to size in octets */
    fddi_pk_size = (fddi_pk_size / 8);
```

...
...

```
    /* Create CPCS_PDU packet */
    cpcs_pdu_pkptr = op_pk_create_fmt ("cpcs_pdu_aal3_4");
    /* Determine the size of the padding field */
10  pad = fddi_pk_size % 4;
    /* We need different values for the begin and end tags */
    b_tag++;
    e_tag++;
    /* Set the CPCS_PDU packet fields */
15  op_pk_nfd_set (cpcs_pdu_pkptr, "begin_tag", b_tag);
    op_pk_nfd_set (cpcs_pdu_pkptr, "cpcs_pdu_payload", pkptr);
    op_pk_nfd_set (cpcs_pdu_pkptr, "padding", pad);
    op_pk_nfd_set (cpcs_pdu_pkptr, "end_tag", e_tag);
    op_pk_nfd_set (cpcs_pdu_pkptr, "length", fddi_pk_size);
20
    /* Calculate the CPCS_PDU packet length */
    cpcs_pdu_total_pk_size = (fddi_pk_size + pad + HEADER_SIZE + TRAILER_SIZE );
    /* Determine segmentation parameters */
    /* - Number of ATM packets */
25  num_of_atm_pks = ((cpcs_pdu_total_pk_size + SAR_PDU_PAYLOAD_SIZE - 1) / SAR_PDU_PAYLOAD_SIZE);
    /* - Number of useful octets in the last ATM packet */
    last_pk_payload_size = (((cpcs_pdu_total_pk_size - 1) % SAR_PDU_PAYLOAD_SIZE) + 1);
    /* handle cases of a single ATM message */
    /* NOTE that this is not a valid assumption for FDDI */
30  /* the minimum packet size of FDDI packets is 64 octets */
    if (num_of_atm_pks == 1) {
        /* Create SAR packet and set fields */
        sar_pkptr = op_pk_create_fmt ("sar_pdu");
        set_sar_pk_fds_ssm (sar_pkptr, cpcs_pdu_pkptr, last_pk_payload_size);
35      /* form ATM packet and send it */
        atm_pk_send (sar_pkptr);
    }

    /* handle cases when more than one packet is to be sent */
40  if (num_of_atm_pks >= 2){
        /* We need a new multiplexing identifier */
        multiplex_id++;
        /* Create first SAR packet */
        sar_pkptr = op_pk_create_fmt ("sar_pdu");
45      set_sar_pk_fds(sar_pkptr, BOM, 0, multiplex_id, OPC_NIL, SAR_PDU_PAYLOAD_SIZE);
        /* create first ATM packet and send it */
        atm_pk_send (sar_pkptr);
        /* Create message 2 to (n-1) and send them */
        for (n = 1; n < (num_of_atm_pks - 1); n++){
50          /* Create next SAR packet andset fields */
            sar_pkptr = op_pk_create_fmt ("sar_pdu");
            set_sar_pk_fds(sar_pkptr, COM, n, multiplex_id, OPC_NIL, SAR_PDU_PAYLOAD_SIZE);
            /* create ATM packet and send it */
            atm_pk_send (sar_pkptr);
55      }
        /* Create last SAR packet */
        sar_pkptr = op_pk_create_fmt ("sar_pdu");
        set_sar_pk_fds(sar_pkptr, EOM, (num_of_atm_pks - 1), multiplex_id, cpcs_pdu_pkptr, last_pk_payload_size);
        /* create last ATM packet and send it */
60      atm_pk_send (sar_pkptr);
    }
```

Process Model Report: **bridge_aal3_4_proc**	Mon Mar 14 21:19:14 1994	Page 10 of 14
... ...		

transition **segment -> idle**			
attribute	*value*	*type*	*default value*
name	tr_15	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **idle**			
attribute	*value*	*type*	*default value*
name	idle	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

transition **idle -> segment**			
attribute	*value*	*type*	*default value*
name	tr_14	string	tr
condition	FDDI_ARRIVAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> reasamble**			
attribute	*value*	*type*	*default value*
name	tr_18	string	tr
condition	ATM_ARRIVAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> idle**			
attribute	*value*	*type*	*default value*
name	tr_24	string	tr
condition	QUEUE_SIZE_GROWS	string	
executive	max_q_size_fddi()	string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> stats**			
attribute	*value*	*type*	*default value*
name	tr_26	string	tr
condition	END_SIM	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

Process Model Report: **bridge_aal3_4_proc**	Mon Mar 14 21:19:14 1994	Page 11 of 14
... ...		

transition **idle -> idle**			
attribute	*value*	*type*	*default value*
name	tr_28	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **reasamble**			
attribute	*value*	*type*	*default value*
name	reasamble	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **reasamble**

```
    /* Get received ATM packet */
    pkptr = op_pk_get (ATM_IN_STRM);
    /* Extract data field */
    op_pk_nfd_get (pkptr, "data", &sar_pkptr);
 5  /* destroy the packet */
    op_pk_destroy (pkptr);
    /* Get the message type of the received data */
    op_pk_nfd_get (sar_pkptr, "segment_type", &seg_type);
    /* Check if it is a single message */
10  if (seg_type == SSM){
        /* Restore encapsulated CPCS_PDU packet */
        op_pk_nfd_get (sar_pkptr, "sar_pdu_payload", &cpcs_pdu_pkptr);
        op_pk_nfd_get (sar_pkptr, "length_indicator", &length_ind);
        /* Destroy SAR-PDU payload to free memory */
15      op_pk_destroy (sar_pkptr);
        /* Calculate the length of the CPCS_PDU packet */
        cpcs_pdu_total_pk_size = (length_ind * 8);
        /* Get the value of the padding field to determine the length of the MAC frame */
        op_pk_nfd_get (cpcs_pdu_pkptr, "padding", &pad);
20      /* Restore the MAC frame */
        op_pk_nfd_get (cpcs_pdu_pkptr, "cpcs_pdu_payload", &payload_ptr);
        /* Destroy the CPCS_PDU packet */
        op_pk_destroy (cpcs_pdu_pkptr);
        /* Calculate the length of the MAC frame */
25      fddi_pk_size = (cpcs_pdu_total_pk_size - HEADER_SIZE - TRAILER_SIZE -pad);
        /* Set the packet length of the MAC frame */
        op_pk_total_size_set (payload_ptr, fddi_pk_size);
        /* Send the restored packet to the FDDI */
        op_pk_send (payload_ptr, FDDI_OUT_STRM);
30  }

    /* Check if it is the begin of a message */
    if (seg_type == BOM){
        /* Get information of segment */
35      op_pk_nfd_get (sar_pkptr, "multiplexing_id", &mult_id);
        op_pk_nfd_get (sar_pkptr, "sequence_number", &seq_number);
        /* Check if there already is an entry in the list */
        if((list_ptr=find_pk(mult_id))!=OPC_NIL){
            /* Delete the old list entry since there an error has occurred */
```

```
40              del_pk_from_list(list_ptr);
            }
            /* Make an entry in the list */
            add_pk_to_list(mult_id, seq_number);
            /* Destroy SAR-PDU payload to free memory */
45          op_pk_destroy (sar_pkptr);
        }

        /* Check if it is the continuing of a message */
        if (seg_type == COM){
50          op_pk_nfd_get (sar_pkptr, "multiplexing_id", &mult_id);
            /* if the packet is not in the list discard segment */
            if ((list_ptr = find_pk(mult_id)) == OPC_NIL){
                if (op_prg_odb_ltrace_active("list_test"))
                    printf("Could not find according packet - COM");
55              op_pk_destroy (sar_pkptr);
            }
            else{
                /* The packet is in the list */
                op_pk_nfd_get (sar_pkptr, "sequence_number", &seq_number);
60              /* If not all previous segments were received discard segment */
                if (list_ptr->last_seq_number != (seq_number - 1)){
                    if (op_prg_odb_ltrace_active("list_test"))
                        printf("Previous packet missing (COM) - discarded packet\n");
                    op_pk_destroy (sar_pkptr);
65              }
                else{
                    /* Update the list entry */
                    list_ptr->last_seq_number = seq_number;
                    if (op_prg_odb_ltrace_active("list_test"))
70                      printf("COM set last seq_number = %d\n", list_ptr->last_seq_number);
                    /* Destroy SAR-PDU payload to free memory */
                    op_pk_destroy (sar_pkptr);
                }
            }
75      }

        /* Check if it is the end of a message */
        if (seg_type == EOM){
            op_pk_nfd_get (sar_pkptr, "multiplexing_id", &mult_id);
80          /* if the packet is not in the list discard segment */
            if ((list_ptr = find_pk(mult_id)) == OPC_NIL){
                if (op_prg_odb_ltrace_active("list_test"))
                    printf("Could not find according packet - EOM\n");
                op_pk_destroy (sar_pkptr);
85          }
            else{
                /* Packet is in the list */
                op_pk_nfd_get (sar_pkptr, "sequence_number", &seq_number);
                /* If not all previous segments were received discard segment */
90              if (list_ptr->last_seq_number != (seq_number - 1)){
                    if (op_prg_odb_ltrace_active("list_test"))
                        printf("Previous packet missing (EOM) - discarded packet\n");
                    op_pk_destroy (sar_pkptr);
                }
95              else{
                    /* Restore encapsulated CPCS_PDU packet */
                    op_pk_nfd_get (sar_pkptr, "sar_pdu_payload", &cpcs_pdu_pkptr);
                    op_pk_nfd_get (sar_pkptr, "length_indicator", &length_ind);
```

```
...
...
                /* Destroy SAR-PDU payload to free memory */
100             op_pk_destroy (sar_pkptr);
                /* Calculate the length of the CPCS_PDU packet */
                cpcs_pdu_total_pk_size = (((seq_number * SAR_PDU_PAYLOAD_SIZE) + length_ind) * 8);
              /* Get the value of the padding field to determine the length of the MAC frame */
                op_pk_nfd_get (cpcs_pdu_pkptr, "padding", &pad);
105             /* Calculate the length of the MAC frame */
                fddi_pk_size = (cpcs_pdu_total_pk_size - HEADER_SIZE - TRAILER_SIZE - pad);
                /* Get the reported length of the CPCS_PDU payload field */
                op_pk_nfd_get (sar_pkptr, "length", &cpcs_pdu_payload_size);
                /* Check if the reported length matches the length of the reassembled frame */
110             if(fddi_pk_size != cpcs_pdu_payload_size){
                    /* Destroy wrongly reassembled CPCS_PDU */
                    op_pk_destroy (cpcs_pdu_pkptr);
                }
                else{
115                 /* Get the values of the begin and end tags */
                    op_pk_nfd_get (cpcs_pdu_pkptr, "begin_tag", &rcv_b_tag);
                    op_pk_nfd_get (cpcs_pdu_pkptr, "end_tag", &rcv_e_tag);
                    /* Chek if they are matching */
                    if (rcv_b_tag != rcv_e_tag){
120                     /* Destroy the packet because they are not matching */
                        op_pk_destroy (cpcs_pdu_pkptr);
                    }
                    else{
                        /* Restore the MAC frame */
125                     op_pk_nfd_get (cpcs_pdu_pkptr, "cpcs_pdu_payload", &payload_ptr);
                        /* Destroy the CPCS_PDU packet to free memory */
                        op_pk_destroy (cpcs_pdu_pkptr);
                        /* Calculate the length of the MAC frame */
                        fddi_pk_size = (cpcs_pdu_total_pk_size - HEADER_SIZE - TRAILER_SIZE -pad);
130                     /* Set the packet length of the MAC frame */
                        op_pk_total_size_set (payload_ptr, fddi_pk_size);
                        /* Send the restored packet to the FDDI */
                        op_pk_send (payload_ptr, FDDI_OUT_STRM);
                    }
135             }
            }
            del_pk_from_list (list_ptr);
        }
    }
140
```

transition **reasamble -> idle**			
attribute	*value*	*type*	*default value*
name	tr_19	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **stats**			
attribute	*value*	*type*	*default value*
name	stats	string	st

Process Model Report: **bridge_aal3_4_proc**	Mon Mar 14 21:19:16 1994	Page 14 of 14
... ...		

enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

enter execs **stats**

```
    /* Check if statistic recording is enabled */
    if (stat_flag == OPC_TRUE){
        /* Record final statistics */
        op_stat_scalar_write("FDDI Throughput (Mb/s)", (op_stat_local_read (BIT_THRUPUT_STAT) / 1000000));
5       op_stat_scalar_write("Mean Packet Delay at FDDI Node", op_stat_local_read (MEAN_DELAY_STAT));
        op_stat_scalar_write("Mean Queue Length at FDDI Node", op_stat_local_read (MEAN_PKSIZE_STAT));
        op_stat_scalar_write("Maximum Queue Length at FDDI Node", max_queue_length);
        op_stat_scalar_write("Packet Throughput at FDDI Node", op_stat_local_read (PK_THRUPUT_STAT));
        op_stat_scalar_write("Utilization of FDDI to ATM Switch Line", op_stat_local_read (UTILIZATION_STAT));
10  }
```

Process Model Report: **bridge_aal5_proc**	Mon Mar 14 21:20:38 1994	Page 1 of 7
... ...		

Process Model Attributes

attribute	*value*	*type*	*default value*
VPI_SET	promoted	integer	2
STAT_ENABLE	promoted	toggle	disabled

Header Block

```
/* packet stream definitions */
#define FDDI_IN_STRM        0
#define ATM_IN_STRM         1
#define FDDI_OUT_STRM       0
#define ATM_OUT_STRM        1

/* Statistical inputs definition */
#define MEAN_DELAY_STAT     0
#define MEAN_PKSIZE_STAT    1
#define QUEUE_LENGTH_STAT   2
#define PK_THRUPUT_STAT     3
#define UTILIZATION_STAT    4
#define BIT_THRUPUT_STAT    5

/* Packet size definitions */
#define ATM_PK_SIZE                 424
#define SAR_PDU_PAYLOAD_SIZE        48
#define CPCS_PDU_TRAILER_SIZE       8

/* transition macros */
#define FDDI_ARRIVAL (op_intrpt_type () == OPC_INTRPT_STRM && \
            op_intrpt_strm () == FDDI_IN_STRM)

#define ATM_ARRIVAL (op_intrpt_type () == OPC_INTRPT_STRM && \
            op_intrpt_strm () == ATM_IN_STRM)

#define QUEUE_SIZE_GROWS (op_intrpt_type () == OPC_INTRPT_STAT && \
                        op_intrpt_stat () == QUEUE_LENGTH_STAT)

#define END_SIM (op_intrpt_type () == OPC_INTRPT_ENDSIM)
```

State Variable Block

```
int         \num_of_segments;
int         \address;
Objid       \module_id;
double      \max_queue_length;
Boolean     \stat_flag;
```

Temporary Variable Block

```
Packet  *pkptr, *cpcs_pdu_pkptr, *payload_pkptr;
int     fddi_total_pk_size;
int     cpcs_pdu_total_pk_size;
int     num_of_atm_pks;
int     n, a, b;
int     pad, pti;

void    atm_pk_send(Packet*, int);
```

...
...

```
    void      max_q_size_fddi ();
10  void      record_stats_fddi ();
```

Function Block

```
    /*****************************************************************/
    /*                          atm_pk_send()                        */
    /*****************************************************************/
    /* Function gets a pointer to a SAR-PDU                          */
 5  /* - encapsulate the SAR-PDU in a ATM packet                     */
    /* - sends the packet                                            */
    /*****************************************************************/

    void atm_pk_send (fkt_sar_pkptr, pti)
10      Packet    *fkt_sar_pkptr;
        int       pti;
        {
        Packet    *atm_pkptr;

15      atm_pkptr = op_pk_create_fmt("atm_pk");
        /* copy SAR packet to ATM packet data field */
        /* NOTE that for simulation purposes this is only */
        /* neccessary for the last segment */
        if (pti == 1){
20          op_pk_nfd_set (atm_pkptr, "data", fkt_sar_pkptr);
        }
        /* Set VPI field of the ATM packet */
        op_pk_nfd_set (atm_pkptr, "VPI", address);
        /* Set the PTI field of the ATM packet */
25      op_pk_nfd_set (atm_pkptr, "pti", pti);
        /* Set total ATM packet size */
        op_pk_total_size_set(atm_pkptr, ATM_PK_SIZE);
        /* Send packet to transmitter */
        op_pk_send(atm_pkptr, ATM_OUT_STRM);
30  }


    /*****************************************************************/
    /*                          max_q_size_fddi ()                   */
35  /*****************************************************************/
    /* This function dtermines the maximum queue of the ATM          */
    /* transmiter channel length during the simulation run           */
    /*****************************************************************/

40  void max_q_size_fddi ()
        {
      double      queue_length;

        /* Read the new queue size */
45      queue_length = op_stat_local_read (QUEUE_LENGTH_STAT);
        /* Check if it is a new maximum */
        if (queue_length > max_queue_length)
            max_queue_length = queue_length;
    }
```

Process Model Report: **bridge_aal5_proc**	Mon Mar 14 21:20:39 1994	Page 3 of 7
... ...		

forced state **init**			
attribute	*value*	*type*	*default value*
name	init	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **init**

```
/* Get module ID */
module_id = op_id_self();
/* Get VPI from process attributes */
op_ima_obj_attr_get (module_id, "VPI_SET", &address);
/* Get the value of the statistics flag */
op_ima_obj_attr_get (module_id, "STAT_ENABLE", &stat_flag);
/* Initialize number of segments of partially received fddi packets */
num_of_segments = 0;
/* Initialize maximum queue size */
max_queue_length = 0;
```

transition **init -> idle**			
attribute	*value*	*type*	*default value*
name	tr_8	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **segment**			
attribute	*value*	*type*	*default value*
name	segment	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **segment**

```
/* Get produced packet */
pkptr = op_pk_get (FDDI_IN_STRM);
/* Get the size of the packet */
fddi_total_pk_size = op_pk_total_size_get (pkptr);
/* Change the total size to size in octets */
fddi_total_pk_size = (fddi_total_pk_size / 8);
/* Create CPCS_PDU packet */
cpcs_pdu_pkptr = op_pk_create_fmt ("cpcs_pdu_aal5");
/* Determine the size of the padding field */
a = fddi_total_pk_size % SAR_PDU_PAYLOAD_SIZE;
b = SAR_PDU_PAYLOAD_SIZE - CPCS_PDU_TRAILER_SIZE;
if (a <= b){
    pad = b - a;
}
else{
    pad = SAR_PDU_PAYLOAD_SIZE - a + b;
```

```
    }
    /* Set the CPCS_PDU packet fields */
    op_pk_nfd_set (cpcs_pdu_pkptr, "cpcs_pdu_payload", pkptr);
    op_pk_nfd_set (cpcs_pdu_pkptr, "padding", pad);
    op_pk_nfd_set (cpcs_pdu_pkptr, "length", fddi_total_pk_size);
    cpcs_pdu_total_pk_size = (fddi_total_pk_size + pad + CPCS_PDU_TRAILER_SIZE);
    /* Determine number of ATM packets */
    num_of_atm_pks = (cpcs_pdu_total_pk_size / SAR_PDU_PAYLOAD_SIZE);
    /* handle cases of a single ATM message */
    /* NOTE that this is not a valid assumption for FDDI */
    /* the minimum packet size of FDDI packets is 64 octets */
    if (num_of_atm_pks == 1) {
        /* form ATM packet and send it */
        atm_pk_send (cpcs_pdu_pkptr, 1);
    }

    /* handle cases when more than one packet is to be sent */
    if (num_of_atm_pks >= 2){
        /* Create message 1 to (n-1) and send them */
        for (n = 1; n <= (num_of_atm_pks - 1); n++){
            /* create ATM packet and send it */
            atm_pk_send (OPC_NIL, 0);
        }
        /* create last ATM packet and send it */
        atm_pk_send (cpcs_pdu_pkptr, 1);
    }
```

transition **segment -> idle**			
attribute	*value*	*type*	*default value*
name	tr_15	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **idle**			
attribute	*value*	*type*	*default value*
name	idle	string	st
enter execs	(empty)	textlist	(empty)
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

transition **idle -> segment**			
attribute	*value*	*type*	*default value*
name	tr_14	string	tr
condition	FDDI_ARRIVAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

Process Model Report: **bridge_aal5_proc**	Mon Mar 14 21:20:40 1994	Page 5 of 7
... ...		

transition **idle -> reasamble**			
attribute	*value*	*type*	*default value*
name	tr_18	string	tr
condition	ATM_ARRIVAL	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> idle**			
attribute	*value*	*type*	*default value*
name	tr_24	string	tr
condition	QUEUE_SIZE_GROWS	string	
executive	max_q_size_fddi()	string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> stats**			
attribute	*value*	*type*	*default value*
name	tr_25	string	tr
condition	END_SIM	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **idle -> idle**			
attribute	*value*	*type*	*default value*
name	tr_26	string	tr
condition	default	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

forced state **reasamble**			
attribute	*value*	*type*	*default value*
name	reasamble	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **reasamble**

```
   /* Get received ATM packet */
   pkptr = op_pk_get (ATM_IN_STRM);
   /* Get the message type of the received data */
   op_pk_nfd_get (pkptr, "pti", &pti);
5
   /* Check if it is the begin or continuing of a message */
   if (pti == 0){
     /* Record reception and destroy ATM packet */
       num_of_segments++;
10     op_pk_destroy (pkptr);
   }
```

...
...

```
else{
    /* It is a single message or end of message */
    num_of_segments++;
    /* Restore encapsulated CPCS_PDU packet and destroy ATM packet */
    op_pk_nfd_get (pkptr, "data", &cpcs_pdu_pkptr);
    op_pk_destroy (pkptr);
    /* Get the CPCS_PDU parameters */
    op_pk_nfd_get (cpcs_pdu_pkptr, "length", &fddi_total_pk_size);
    op_pk_nfd_get (cpcs_pdu_pkptr, "padding", &pad);
    cpcs_pdu_total_pk_size = (fddi_total_pk_size + pad + CPCS_PDU_TRAILER_SIZE);
    /* Check if segments are lost or inserted */
    if (cpcs_pdu_total_pk_size != (num_of_segments * SAR_PDU_PAYLOAD_SIZE)){
        op_pk_destroy (cpcs_pdu_pkptr);
        num_of_segments = 0;
    }
    else{
        /* Restore encapsulated FDDI packet and destroy CPCS_PDU packet */
        op_pk_nfd_get (cpcs_pdu_pkptr, "cpcs_pdu_payload", &payload_pkptr);
        op_pk_destroy (cpcs_pdu_pkptr);
        /* Change length of fddi packet to size in bits */
        fddi_total_pk_size = fddi_total_pk_size * 8;
        op_pk_total_size_set (payload_pkptr, fddi_total_pk_size);
        /* Send the restored packet to the FDDI */
        op_pk_send (payload_pkptr, FDDI_OUT_STRM);
        /* Reset number of received segments */
        num_of_segments = 0;
    }
}
```

transition **reasamble -> idle**			
attribute	*value*	*type*	*default value*
name	tr_19	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **stats**			
attribute	*value*	*type*	*default value*
name	stats	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	unforced	toggle	unforced

enter execs **stats**

```
/* Check if statistic recording is enabled */
if (stat_flag == OPC_TRUE){
    /* Record final statistics */
        op_stat_scalar_write("FDDI Throughput (Mb/s)", (op_stat_local_read (BIT_THRUPUT_STAT) / 1000000));
        op_stat_scalar_write("Mean Packet Delay at FDDI Node", op_stat_local_read (MEAN_DELAY_STAT));
        op_stat_scalar_write("Mean Queue Length at FDDI Node", op_stat_local_read (MEAN_PKSIZE_STAT));
        op_stat_scalar_write("Maximum Queue Length at FDDI Node", max_queue_length);
        op_stat_scalar_write("Packet Throughput at FDDI Node", op_stat_local_read (PK_THRUPUT_STAT));
```

...
...

```
10  }
```

op_stat_scalar_write("Utilization of FDDI to ATM Switch Line", **op_stat_local_read** (UTILIZATION_STAT

Process Model Report: **fddi_gen_vbr**	Mon Mar 14 21:21:19 1994	Page 1 of 8
... ...		

Process Model Attributes

attribute	*value*	*type*	*default value*
low dest address	promoted	integer	-1
high dest address	promoted	integer	-1
arrival rate	promoted	double	1.0 (pk/sec)
mean pk length	promoted	double	1,024 (bits)
async_mix	promoted	double	0.5
dest_ring_id	promoted	integer	0
traffic_dist	promoted	string	constant
idle_dist	promoted	string	constant
idle_dist_arg	promoted	double	1.0
busy_dist	promoted	string	constant
busy_dist_arg	promoted	double	1.0
vbr_gen_seed_I	promoted	integer	101
vbr_gen_seed_II	promoted	integer	201

Header Block

```
#define MAC_LAYER_OUT_STREAM        0

/* define possible service classes for frames */
#define FDDI_SVC_ASYNC              0
#define FDDI_SVC_SYNC               1

/* define token classes */
#define FDDI_TK_NONRESTRICTED       0
#define FDDI_TK_RESTRICTED          1

/* Define transition macros */
#define NEXT_IDLE_PERIOD    (op_intrpt_code() == 2)
#define NEXT_BUSY_PERIOD    (op_intrpt_code() == 1)
#define GENERATE_PACKET     (op_intrpt_code() == 0)
```

State Variable Block

```
Distribution*       \inter_dist_ptr;
Distribution*       \len_dist_ptr;
Distribution*       \dest_dist_ptr;
Distribution*       \nxt_busy_ptr;
Distribution*       \nxt_idle_ptr;

Objid               \mac_objid;
Objid               \my_id;

Evhandle            \pk_intrpt;

int                 \low_dest_addr;
int                 \high_dest_addr;
int                 \station_addr;
int                 \ring_id;
int                 \dest_ring_id;

char                \traffic_dist[25];
char                \busy_dist[25];
char                \idle_dist[25];
```

...
...

```
double          \arrival_rate;
double          \busy_dist_arg;
double          \idle_dist_arg;
double          \mean_pk_len;
double          \async_mix;

Ici*            \mac_iciptr;
```

Temporary Variable Block

```
Packet      *pkptr;
int         pklen;
int         dest_addr;
int         i, restricted;
int         seed_I, seed_II;

void        start_random_number(int, int);
double      next_random_number();
double      uniform_rand_val;
```

Function Block

```
/*
 * Title:       random_number
 * Last Mod:        Fri Mar 18 08:52:13 1988
 * Author:          Vincent Broman
 *          <broman@schroeder.nosc.mil>
 */

#define P 179
#define PM1 (P - 1)
#define Q (P - 10)
#define STATE_SIZE 97
#define MANTISSA_SIZE 24
#define RANDOM_REALS 16777216.0
#define INIT_C   362436.0
#define INIT_CD  7654321.0
#define INIT_CM 16777213.0

static unsigned int ni;
static unsigned int nj;
static double u[STATE_SIZE];
static double c, cd, cm;


static unsigned int collapse (anyint, size)
int anyint;
unsigned int size;
/*
 * return a value between 0 and size-1 inclusive.
 * this value will be anyint itself if possible,
 * otherwise another value in the required interval.
 */
{
  if (anyint < 0)
     anyint = - (anyint / 2);
  while (anyint >= size)
```

...
...

```
    anyint /= 2;
  return (anyint);
}

void start_random_number (seed_a, seed_b)
int seed_a;
int seed_b;
/*
 * This procedure initialises the state table u for a lagged
 * Fibonacci sequence generator, filling it with random bits
 * from a small multiplicative congruential sequence.
 * The auxilliaries c, ni, and nj are also initialized.
 * The seeds are transformed into an initial state in such a way that
 * identical results are guaranteed across a wide variety of machines.
 */
{
double s, bit;
unsigned int ii, jj, kk, mm;
unsigned int ll;
unsigned int sd;
unsigned int elt, bit_number;

  sd = collapse (seed_a, PM1 * PM1);
  ii = 1 + sd / PM1;
  jj = 1 + sd % PM1;
  sd = collapse (seed_b, PM1 * Q);
  kk = 1 + sd / PM1;
  ll = sd % Q;
  if (ii == 1 && jj == 1 && kk == 1)
    ii = 2;

  ni = STATE_SIZE - 1;
  nj = STATE_SIZE / 3;
  c  = INIT_C;
  c /= RANDOM_REALS;        /* compiler might mung the division itself */
  cd = INIT_CD;
  cd /= RANDOM_REALS;
  cm = INIT_CM;
  cm /= RANDOM_REALS;

  for (elt = 0; elt < STATE_SIZE; elt += 1) {
    s = 0.0;
    bit = 1.0 / RANDOM_REALS;
    for (bit_number = 0; bit_number < MANTISSA_SIZE; bit_number += 1) {
      mm = (((ii * jj) % P) * kk) % P;
      ii = jj;
      jj = kk;
      kk = mm;
      ll = (53 * ll + 1) % Q;
      if (((ll * mm) % 64) >= 32)
        s += bit;
      bit += bit;
    }
    u[elt] = s;
  }
}
```

...
...

```
 95  double next_random_number()
     /*
      * Return a uniformly distributed pseudo random number
      * in the range 0.0 .. 1.0-2**(-24) inclusive.
      * There are 2**24 possible return values.
100   * Side-effects the non-local variables: u, c, ni, nj.
      */
     {
     double uni;

105    if (u[ni] < u[nj])
         uni = u[ni] + (1.0 - u[nj]);
       else
         uni = u[ni] - u[nj];
       u[ni] = uni;
110
       if (ni > 0)
         ni -= 1;
       else
         ni = STATE_SIZE - 1;
115
       if (nj > 0)
         nj -= 1;
       else
         nj = STATE_SIZE - 1;
120
       if (c < cd)
         c = c + (cm - cd);
       else
         c = c - cd;
125
       if (uni < c)
         return (uni + (1.0 - c));
       else
         return (uni - c);
130  }
```

forced state **INIT**			
attribute	*value*	*type*	*default value*
name	INIT	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(empty)	textlist	(empty)
status	forced	toggle	unforced

enter execs **INIT**

```
     /* determine id of own processor to use in finding attrs */
     my_id = op_id_self ();

     /* Set up the seed for the random number generator */
  5  op_ima_obj_attr_get (my_id, "vbr_gen_seed_I", &seed_I);
     op_ima_obj_attr_get (my_id, "vbr_gen_seed_II", &seed_II);
     /* Initialize Marsaglia's random number generator */
     start_random_number(seed_I, seed_II);

 10  /* determine address range for uniform destination assignment */
```

...
...

```
    op_ima_obj_attr_get (my_id, "low dest address", &low_dest_addr);
    op_ima_obj_attr_get (my_id, "high dest address", &high_dest_addr);

    /* determine object id of connected 'mac' layer process */
15  mac_objid = op_topo_assoc (my_id, OPC_TOPO_ASSOC_OUT,
            OPC_OBJMTYPE_MODULE, MAC_LAYER_OUT_STREAM);

    /* determine the station and ring address assigned to it */
    /* which are also the addresses of this station */
20  op_ima_obj_attr_get (mac_objid, "station_address", &station_addr);
    op_ima_obj_attr_get (mac_objid, "ring_id", &ring_id);

    /* setup a distribution for generation of addresses */
    dest_dist_ptr = op_dist_load ("uniform_int", low_dest_addr,
25                                      high_dest_addr);

    /* determine the ring identification of the destination */
    op_ima_obj_attr_get (my_id, "dest_ring_id", &dest_ring_id);

30  /* also determine the arrival rate for packet generation */
    op_ima_obj_attr_get (my_id, "arrival rate", &arrival_rate);

    /* also determine the traffic distribution function */
    op_ima_obj_attr_get (my_id, "traffic_dist", traffic_dist);
35
    /* determine the mix of asynchronous and synchronous */
    /* traffic. This is expressed as the proportion of */
    /* asynchronous traffic. i.e a value of 1.0 indicates */
    /* that all the produced traffic shall be asynchronous. */
40  op_ima_obj_attr_get (my_id, "async_mix", &async_mix);

    /* set up a distribution for arrival generations */
    if (arrival_rate != 0.0)
        {
45      /* arrivals are distributed, with given mean */
        inter_dist_ptr = op_dist_load (traffic_dist, 1.0 / arrival_rate, 0.0);

        /* determine the distribution for packet size */
        op_ima_obj_attr_get (my_id, "mean pk length", &mean_pk_len);
50
        /* set up corresponding distribution */
        len_dist_ptr = op_dist_load ("constant", mean_pk_len, 0.0);

        /* Get the distribution parameters for the busy period from the process attributes */
55      op_ima_obj_attr_get (my_id, "busy_dist", busy_dist);
        op_ima_obj_attr_get (my_id, "busy_dist_arg", &busy_dist_arg);

        /* Load the distribution of the busy periods */
        nxt_busy_ptr = op_dist_load (busy_dist, busy_dist_arg, 0.0);
60
        /* Get the distribution parameters for the idle period from the process attributes */
        op_ima_obj_attr_get (my_id, "idle_dist", idle_dist);
        op_ima_obj_attr_get (my_id, "idle_dist_arg", &idle_dist_arg);

65      /* Load the distribution of the idle periods */
        nxt_idle_ptr = op_dist_load (idle_dist, idle_dist_arg, 0.0);

        /* set up an interface control information (ICI) structure */
        /* to communicate parameters to the mac layer process */
```

Process Model Report: **fddi_gen_vbr**	Mon Mar 14 21:21:21 1994	Page 6 of 8

...
...

```
70      /* (it is more efficient to set one up now and keep it */
        /* as a state variable than to allocate one on each packet xfer) */
        mac_iciptr = op_ici_create ("fddi_mac_req_II");
    }
```

transition **INIT -> idle**			
attribute	*value*	*type*	*default value*
name	tr_10	string	tr
condition		string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **busy**			
attribute	*value*	*type*	*default value*
name	busy	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(See below.)	textlist	(See below.)
status	unforced	toggle	unforced

enter execs **busy**

```
    /* Get a random number (value is in the zero-one interval) */
    uniform_rand_val = next_random_number();
    /* Schedule interrupt to generate packet */
    pk_intrpt = op_intrpt_schedule_self (op_sim_time() + op_dist_outcome_ext (inter_dist_ptr, uniform_rand_val), 0);
5
```

exit execs **busy**

```
    if (op_intrpt_code() == 0){
        /* Get a random number(value is in the zero-one interval) */
        uniform_rand_val = next_random_number();
        /* determine the length of the packet to be generated */
5       pklen = op_dist_outcome_ext (len_dist_ptr, uniform_rand_val);

        /* determine the destination */
        /* dont allow this station's address as a possible outcome */
        gen_packet:
10      /* Get a random number(value is in the zero-one interval) */
        uniform_rand_val = next_random_number();
        dest_addr = op_dist_outcome_ext (dest_dist_ptr, uniform_rand_val);
        if ((dest_addr != -1) && (dest_addr == station_addr) && (dest_ring_id == ring_id))
            goto gen_packet;
15
        /* create a packet to send to mac */
        pkptr = op_pk_create_fmt ("fddi_llc_fr");

        /* assign its overall size. */
20      op_pk_total_size_set (pkptr, pklen);

        /* assign the time of creation */
        op_pk_nfd_set (pkptr, "cr_time", op_sim_time ());
```

```
25      /* place the destination address and the destination ring */
        /* identification into the ICI */
        /* (the protocol_type field will default) */
        op_ici_attr_set (mac_iciptr, "dest_addr", dest_addr);
        op_ici_attr_set (mac_iciptr, "dest_ring_id", dest_ring_id);
30
        /* assign the priority, and requested token class */
        /* also assign the service class */
        if (op_dist_uniform (1.0) <= async_mix)
            {
35          op_ici_attr_set (mac_iciptr, "svc_class", FDDI_SVC_ASYNC);
            }
        else{
            op_ici_attr_set (mac_iciptr, "svc_class", FDDI_SVC_SYNC);
            }
40
        /* Request only nonrestricted tokens after transmission */
        op_ici_attr_set (mac_iciptr, "tk_class", FDDI_TK_NONRESTRICTED);

        op_ici_attr_set (mac_iciptr, "pri", 0);
45
        /* send the packet coupled with the ICI */
        op_ici_install (mac_iciptr);
        op_pk_send (pkptr, MAC_LAYER_OUT_STREAM);
    }
50  else
        op_ev_cancel(pk_intrpt);
```

transition **busy -> busy**			
attribute	*value*	*type*	*default value*
name		string	tr
condition	GENERATE_PACKET	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

transition **busy -> idle**			
attribute	*value*	*type*	*default value*
name	tr_12	string	tr
condition	NEXT_IDLE_PERIOD	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

unforced state **idle**			
attribute	*value*	*type*	*default value*
name	idle	string	st
enter execs	(See below.)	textlist	(See below.)
exit execs	(See below.)	textlist	(See below.)
status	unforced	toggle	unforced

Process Model Report: fddi_gen_vbr	Mon Mar 14 21:21:21 1994	Page 8 of 8
... ...		

enter execs **idle**

```
/* Schedule next busy period if packets are to be generated */
if (arrival_rate != 0.0){
    /* Get a random number(value is in the zero-one interval) */
    uniform_rand_val = next_random_number();
    /* Schedule next busy period */
    op_intrpt_schedule_self (op_sim_time() + op_dist_outcome_ext (nxt_busy_ptr, uniform_rand_val), 1);
}
```

exit execs **idle**

```
/* Get a random number(value is in the zero-one interval) */
uniform_rand_val = next_random_number();
/* Schedule next idle period */
op_intrpt_schedule_self (op_sim_time() + op_dist_outcome_ext (nxt_idle_ptr, uniform_rand_val), 2);
```

transition **idle -> busy**

attribute	*value*	*type*	*default value*
name	tr_11	string	tr
condition	NEXT_BUSY_PERIOD	string	
executive		string	
color	RGB333	color	RGB333
drawing style	spline	toggle	spline

APPENDIX C. DESCRIPTION OF THE OPNET FDDI EXAMPLE MODEL

FDDI.0 Background / Operational Description

The content of this chapter and the OPNET model that it describes are primarily based upon the *American National Standard for Information Systems Specification X3.139-1987- Fiber Distributed Data Interface Token Ring Media Access Control.* This document can be obtained from:

American National Standards Institute, Inc.
1430 Broadway New York, NY 10018

The Fiber Distributed Data Interface (FDDI) provides general purpose networking at **100 Mbits/sec** transmission rates for large numbers of communicating stations configured in a ring topology. Use of ring bandwidth is controlled through a timed token rotation protocol, wherein stations must receive a token and meet with a set of timing and priority criteria before transmitting frames. In order to accommodate network applications in which response times are critical, FDDI provides for deterministic availability of ring bandwidth by defining a synchronous transmission service. Asynchronous frame transmission requests dynamically share the remaining ring bandwidth.

A key parameter of the FDDI media access protocol is the **Target Token Rotation Time** (**TTRT**). The **TTRT** is a parameter which is in effect, global to all stations on the ring since its value is agreed upon by all stations at ring initialization. The **TTRT** is the expiration value for a **Token Rotation Timer** (**TRT**), which is maintained by each station. This timer holds the time since the token was last captured by a station (modulo **TTRT**). Expiration of **TRT** causes a station to increment a state variable called `Late_Ct`, which indicates if the token's arrival has exceeded **TTRT**. Stations that capture a token and have a non-zero `Late_Ct` may use the token only for synchronous transmissions. `Late_Ct` is reset to zero when a token is captured. This protocol limits to **TTRT** the total amount of asynchronous transmission by all stations during a full rotation of the token. The total synchronous bandwidth allocated to all stations in the ring is also required to be less than **TTRT**. The maximum time for a full token rotation is therefore twice **TTRT** and if the ring is operating properly, `Late_Ct` should not exceed one. Stations requiring a maximum response time **T_max** should choose a **TTRT** of at most **T_max/2**.

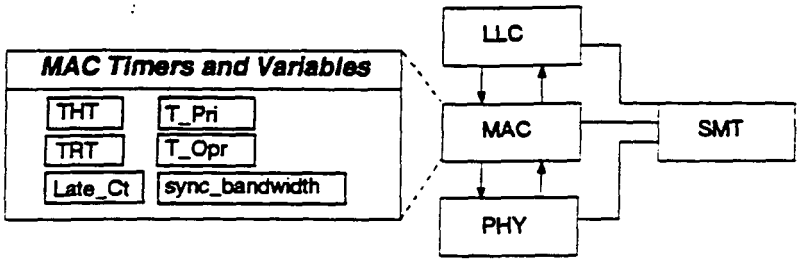
Each station maintains a **Token Holding Timer** (**THT**), which limits asynchronous transmission while still allowing **TRT** to progress independently. **THT** contains the value of **TRT** when asynchronous frame transmission begins and is enabled during the transmission of each asynchronous frame. The difference between the value of **THT** and **TTRT** represents the amount of asynchronous transmission still available to the station. When **THT** expires (i.e., reaches **TTRT**), the asynchronous transmission in progress (if any) is allowed to complete. The time for which transmission occurs beyond the expiration of **THT** is called the *residual transmission time*. The residual transmission time is always less than the maximum frame transmission time.

FDDI also allows priorities to be associated with frames queued for transmission. The priorities are defined in terms of threshold values for **THT**. Frames whose thresholds are exceeded by **THT** are withheld from transmission. By assigning lower threshold values to a frame, its transmission is more likely to be blocked, thus freeing bandwidth for stations with higher priority requests.

In order to support bursts of high volume and continuous traffic on the ring, FDDI supports a restricted mode in which a specially marked token is monopolized by two stations that reserve all asynchronous bandwidth on the ring until they again release a *non-restricted* token. During this time, however, other stations are allowed to use the restricted token for transmission of synchronous frames.

The diagram below illustrates some of the important elements contained in the FDDI-MAC entity, as well as the basic context in which it operates.

FDDI MAC Operational Context



Most of the elements shown within the **MAC** have already been discussed. **T_Opr** simply represents the operative value of the **TTRT**, which is agreed upon at ring initialization. All stations must have the same value of **T_Opr** for the ring to operate properly. **T_Pri** represents an array of **THT** thresholds used to establish priority classes as mentioned above. This array maps integer priority levels into thresholds which are compared with the value of **THT** before transmitting asynchronous frames. **sync_bandwidth** represents the synchronous bandwidth allocation for the station, normalized to **TTRT**.

The diagram also illustrates **MAC**'s relationship with some of the surrounding entities. **MAC** receives frames from **LLC**, which are intended for transmission to a peer **LLC** entity. These frames are transmitted via **PHY** in accordance with the rules outlined above. **MAC** delivers frames received from **PHY**, and destined for this station, to **LLC**. The station management entity, **SMT**, is a supervisory entity that controls and monitors **LLC**, **MAC**, and **PHY**. Each transfer of data across entity interfaces is coupled with the specification of control information that is fully described in the standard.

FDDI.1 Model Scope and Limitations

The previous section briefly described the basic mechanisms and operational context of the FDDI **MAC** entity. This section discusses the implementation choices made in constructing an OPNET model of **MAC**. Because the model is intended for the purpose of simulation, and particularly for performance estimation, certain parts of the protocol have been simplified or omitted. It is important to understand which mechanisms are modeled in order to gauge whether the model is applicable for a particular simulation study.

The first restriction is that the ring initialization and recovery processes are not modeled explicitly. While the model could be extended to address these areas (it is provided in source code form), its primary usefulness is in obtaining measurements of steady state performance. The initial alignment of station timers and bidding process by which all **MAC** entities negotiate the **TTRT** is performed in an essentially static manner, as shall be seen in the detailed description of the models below. This also applies to the distribution of synchronous transmission bandwidth.

A second restriction has to do with the modeling of error conditions and in general, the role of the **SMT** entity. In its current form, the model makes no attempt to implement the mechanisms related to the detection of damaged frames, or the reporting of errors to **SMT**. The interface between **MAC** and **SMT** is, in fact, not presently incorporated into the implementation of **MAC**.

The FDDI model incorporates a simulation acceleration feature for modeling the passing of the token from station to station. When the ring experiences an idle period with no transmissions, the token may be passed many times in a very short period, thus generating many simulation events and consuming large amounts of real time while producing data that is of little interest. In order to jump over these periods, a procedure is employed whereby **MAC** modules register their interest in using the token and also yield the token through a centrally managed set of variables. When an idle period is encountered, token passing is blocked. It is later re-injected into the ring as soon as a station again has a need for it.

The model does incorporate the interfaces between **MAC** and **LLC**, as well as those between **PHY** and **MAC**. These are described in the next section entitled *Model Interfaces*. In addition, the primary data transfer features of FDDI are modeled explicitly, including synchronous and asynchronous transmission, definable priority levels for asynchronous frames, and restricted tokens. The effects of station latency and propagation delay are also incorporated into the model. The parameters that may be easily controlled by the user, without having to modify the internals of any of the provided models include:

- the number of stations attached to the ring
- synchronous bandwidth allocation at each station
- requested value of the **TTRT** by each station (**T_Req**)

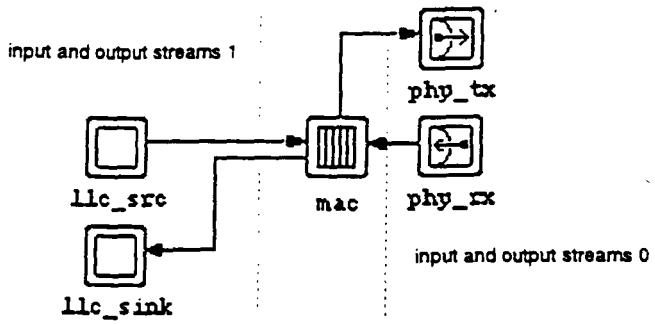
- the address of the station that launches the token as the simulation begins
- the delay incurred by frames and tokens as they traverse a station's ring interface
- the propagation delay separating stations on the ring
- the rate of exponentially distributed frame generation at each station and the size of generated frames
- the mix of asynchronous and synchronous traffic generated at each station
- the range of destination addresses for the frames generated at each station

Implicit in the parameters listed above is one other simplification made by the model: the station latency and inter-station propagation delay are assumed to be uniform across the ring. This is primarily done to conveniently support the simulation acceleration option described later.

FDDI.2 Model Interfaces

There are simple requirements for the connection of **MAC** to **PHY** and **LLC**. These have to do with the physical port numbers, which in OPNET are called *stream indices*, to which the packet streams attach. The `fddi_mac` process model expects to attach to the **PHY** entity via input and output stream `0`, and expects to attach to the **LLC** entity via input and output stream `1`. This is illustrated in the following diagram:

MAC Physical Interfaces



MAC is not concerned with the format of frames transferred to it by LLC, since it simply encapsulates these packets into MAC frames before sending them into the ring. Similarly, it decapsulates these frames from the MAC frames that it receives from PHY before forwarding them to the LLC. For the purposes of the example model described later in section *FDDI.4*, a packet format called "`fddi_llc_fr`" is used for the frames generated by LLC. However, frames of any other format could be sent to the MAC. It is important to note that the processor module `mac` does not enforce the frame size limitations specified in the standard, and that it is the responsibility of the higher level models to generate requests that will result in conformant MAC frame sizes.

FDDI.3 Model Internal Structure

This section describes the internal structure of the MAC entity, represented by the processor module `mac` as shown above. The interface is essentially dictated by the process model `fddi_mac`, which resides in the processor module `mac` and forms the core element of the FDDI example model. `fddi_mac` will be described in this section, beginning with the indication primitives and interrupt definitions, and then the State Transition Diagram (STD) and each of the states will be discussed.

FDDI.3.0 Indication Primitives and Interrupt Definitions

The transfer of frames to and from the MAC entity are the only primitives defined in this model. The following indication primitives described in the specification need not be explicitly modeled since other mechanisms provided by OPNET operate in an equivalent manner, or they are not part of the scope of the example model:

MA_UNITDATA.indication

This primitive corresponds to the notification to **LLC** of a frame arrival, which has been determined by **MAC** to be addressed for this station. In OPNET, this mechanism is replaced by a **stream interrupt**, which is delivered to the **LLC** when **MAC** forwards a packet on the packet stream that connects them.

MA_UNITDATA_STATUS.indication

This primitive is used by **MAC** to report the status of a frame transmission request to the **LLC**. In this example model, there is no implementation of mechanisms related to failure on the part of any component to perform its specified service, so the **LLC** can assume that its service requests are satisfied by **MAC**.

PH_UNITDATA_STATUS.indication,
PH_UNITDATA.indication

These primitives correspond to the notification to **MAC** by **PHY** of the decoding symbol arriving from the physical media, and by **MAC** to **PHY** of the transfer of a symbol. In this model, all transactions between **MAC** and **PHY** are treated with a frame by frame granularity. To do otherwise would be prohibitive in terms of simulation performance.

The primitives that are modeled are:

MA_UNITDATA.request

This primitive corresponds to submission by **LLC** to **MAC** of data to be transmitted to a peer **LLC** entity. While several Service Data Units (**SDU**'s) may be grouped into a single invocation of this primitive, in this example model, a **stream interrupt** will be associated with the transfer of each frame between the **LLC** and the **MAC**. Control information associated with each transfer of an **SDU** is also provided by this primitive. In this example model, the control fields are packaged into an OPNET **Interface Control Information** (**ici**) structure whose format is **fddi_mac_req**. The attributes present in this format are **"svc_class"**, **"dest_addr"**, **"pri"**, and **"tk_class"**. The integer attribute **"svc_class"** represents the class of service requested for the frame transmission and a value of **0** corresponds to asynchronous transmission while a

value of 1 corresponds to synchronous transmission. The integer attribute "dest_addr" specifies the destination address for the recipient(s) of the frame. The "pri" attribute is the priority class of the frame and is only meaningful for asynchronous transmission requests. It is used as the key into the T_Pri array to obtain THT thresholds that may cause the transmission of the frame to be deferred. Finally, the integer attribute "tk_class" can be used to indicate that a restricted token should be issued by MAC after processing of the request. A value of 0 indicates a non-restricted token, while a value of 1 indicates a restricted token should be issued. This attribute is meaningful only for asynchronous transmissions.

MA_UNITDATA.indication

This primitive corresponds to transfer of data from MAC to LLC. It occurs when MAC has captured a frame that is addressed for the local station. In the model, this event is implemented via a stream interrupt that occurs when MAC sends a packet to LLC over the packet stream that connects them. In addition to the delivery of the actual data in the form of an OPNET packet, there are control fields associated with this event. As in the case of the transfer from the LLC to the MAC, these fields are grouped into an ici structure. This ici structure abides by the format fddi_mac_ind. The attributes contained in this format are two integers named "src_addr" and "dest_addr", which represent the addresses of the originating and receiving stations, respectively.

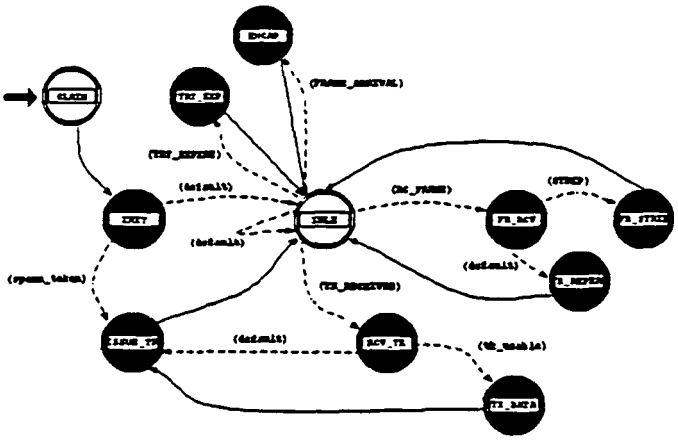
FDDI.3.1 Process STD and State Definitions

The process model fddi_mac is specified by a State Transition Diagram (STD) that manages the timers and state variables associated with a single MAC entity. The responsibilities of the fddi_mac process include forming MAC frames that encapsulate data received from LLC, repeating frames destined for other stations, decapsulating data from frames destined for this station and passing this data to LLC, stripping frames originated by this station, maintaining THT and TRT timers, determining token usability, and transmitting MAC frames into the ring according to the rules defined in the FDDI MAC specification.

The fddi_mac process must execute within a queue module and be connected to lower and higher layer entities as described in the fddi_station example node model later in this chapter. fddi_mac uses a single subqueue of the queue module

in which it is located to hold frames that are waiting to become eligible for transmission. The actions and resources defined in `fddi_mac` are fully specified in the process model code, which can be viewed from the `opnet` Process Editor. The code found in these listings is lined with extensive comments so that in most cases, it should be self-explanatory. The STD is shown below, followed by a discussion of each state.

fddi_mac Process Model State Diagram



CLAIM state

This is the initial state of the process model and is entered upon receipt of a **begin simulation interrupt** which is delivered by the Simulation Kernel when the simulation starts. Its primary purpose is to emulate the negotiation that takes place in an FDDI ring with regard to the value of TTRT. All `fddi_mac` processes in the ring can compare their requested value of TTRT, `T_Req`, by means of a global variable `Fddi_T_Opr`, which holds the lowest value yet requested. After the last station exits the `CLAIM` state, `Fddi_T_Opr` has become the operative value of TTRT, which is then used to regulate the use of bandwidth in the ring. A semaphore represented by the variable `Fddi_Claim_Start` is used so that the first station to enter the `CLAIM` state will automatically place its `T_Req` in `Fddi_T_Opr` without performing a comparison, since at this early stage `Fddi_T_Opr` has no defined value. Each process also requests that it again be interrupted after the claim phase has completed so that further initializations, which may be dependent upon the final selected value of TTRT, can be performed. After exiting the `CLAIM` state, the `fddi_mac` process always transfers control to the `INIT` state upon receiving its next interrupt.

INIT state

This state is entered by every fddi_mac process after the completion of the claim phase. These is no delay in simulated time since the last action in the CLAIM state is to request a self interrupt for the process with zero time delay. However, this mechanism guarantees that all processes have executed their CLAIM state before *any* process enters its INIT state. All actions of the INIT state are held within its enter executives. INIT is not reentrant because its actions are all related to initialization and need not be performed more than once. Most of the actions performed in INIT have to do with assignment of values to state and global simulation variables. These include the station latency Fddi_St_Latency, the inter-station propagation delay Fddi_Prop_Delay, the combined inter-station delay Fddi_Tk_Hop_Delay, and the priority table Fddi_T_Pri.

The process model also defines a timer object in the header block and a series of procedures for manipulating and querying timers. In INIT two such timers are created for TRT and THT. The THT timer is initialized to expire one TTRT period later, and a self interrupt is set to occur at that time. Throughout the process model, whenever TRT is set, a corresponding self interrupt is requested, so that at the time of expiration, the Late_Ct variable can be incremented. Also note that when the self interrupt is requested, an event handle is obtained and placed in the state variable TRT_handle, so that, if an early token is captured, the request can be repealed, and a new one issued. The Late_Ct variable is reset and the state variable restricted is set to 0, since the station is not at this point in a restricted mode.

In INIT, fddi_mac also acquires knowledge of its own station's address and places this value in the state variable my_address. An ici is created which abides by the format "fddi_mac_to_llc" defined via the Parameter Editor. This ici is stored in the state variable to_llc_ici_ptr, and is used to specify control information fields when delivering service data units to the LLC. The fddi_mac registers itself in a global table that maps station addresses to process object ID's. This table is used by an acceleration mechanism that bypasses idle periods to avoiding letting the token freely circulate and generate a large number of events. Also associated with this mechanism is the tk_registered variable which indicates if this station has registered its desire to make use of the token should it become available. This acceleration mechanism is only used if the global simulation variable accelerate_token is set to 1; this is normally done in an external environment file supplied to the simulation upon execution. The assignment of this attribute is placed in the global variable Fddi_Tk_Accelerate. The sync_bandwidth state variable, which represents the synchronous bandwidth usable on each token capture (normalized to TTRT), is initialized. Note that there is no logic in place to verify that the sum of these assignments for all stations does not exceed one.

The final statements of INIT have to do with the initial generation of a token for the ring. A single station in the ring is designated as the spawning station, which has the responsibility of creating an OPNET packet representing the token and releasing it into the ring. The address of this station is specified in the global simulation attribute "spawn station". The value of this attribute is loaded into the variable spawn_station so that it can be compared to my_address. If a match oc-

curs, a token is created with the format `fddi_mac_tk`, and its fields are initialized. Also, in case of a match, the variable `spawn_token` is set to 1. This will cause a transition to the `ISSUE_TK` state after completing the actions of `INIT` where the token will be sent into the ring. For other stations, where a match does not occur, the next transition will be to the `IDLE` state instead. The last action of `INIT` resets the value of the `accum_bandwidth` variable, which keeps track of the total amount of bandwidth scheduled for transmission since a token was captured. This value is used to schedule packet transmissions so that they will occur with the proper delays. In the case of the initial transmission of the token, there is no delay due to other transmissions.

ISSUE_TK state

This state is entered whenever `fddi_mac` needs to issue a token. Its primary action is therefore to forward the OPNET packet that represents the token (held in the variable `tk_pkptr`) with the proper delay, which reflects the accumulated bandwidth consumed by previous frame transmissions, and the propagation delay which the token will experience as it travels to the next down-stream neighbor. In addition, `ISSUE_TK` checks for the special condition whereby the station is releasing the token without having performed any frame transmissions, and the station has no data to transmit. If this condition is met, the procedure ***fddi_tk_indicate_no_data()*** is called. This procedure is defined in the function block, and checks for the condition where all stations on the ring have no data to transmit. If any station has data to transmit, the token will be forwarded normally, otherwise the token will be blocked until a station registers its intent to use it. This mechanism provides significant improvements in simulation efficiency, particularly in simulations where the network traffic is well below network capacity. After executing the statements in the `ISSUE_TK` state, `fddi_mac` always transitions to the `IDLE` state.

IDLE state

This state is the branching point for event processing in the steady state operation of `fddi_mac`. The interrupts to which the process will respond while in the `IDLE` state are **stream interrupts** signaling the arrival of service data units from LLC, **stream interrupts** signaling the arrival of frames or tokens from PHY, **self interrupts** representing the expiration of THT, and **remote interrupts** requesting that a token be generated and inserted into the ring. Upon receiving any of these interrupts, `fddi_mac` executes the code present in the exit executives of the `IDLE` state. This code prepares variables used on the transition conditions, and handles the case of a **remote interrupt**. All other interrupt types are processed by leaving `IDLE` and going to the appropriate destination state. In the case of a **stream interrupt**, `IDLE` distinguishes between arrivals from PHY and arrivals from LLC, and sets the variable `phy_arrival` accordingly. In addition, in the case of **stream interrupts** from PHY, `IDLE` sets the variable `frame_control` to indicate if a token or a frame has arrived. If a **remote interrupt** is trapped, the event corresponds to a re-introduction of the token

into the ring after an idle period where no stations had data to transmit. During such periods, if the token acceleration mechanism is enabled, the token transfers between stations are not modeled explicitly in order to economize simulation events. Once a station registers interest in using the token again (i.e., it produces data for transmission), the token acceleration mechanism computes the station at which the token would be present, based on the token hop delay, and the duration of the idle interval. This station is notified via a **remote interrupt** that it should generate a new token and send it into the ring.

TRT_EXP state

This state is entered when `fddi_mac` receives a **self interrupt** indicating that the **TRT** timer has expired. At this point, the timer is reset to expire one **TTRT** into the future and a corresponding **self interrupt** is requested. Also, the `Late_Ct` variable is incremented to indicate the lateness of the token. After completing the enter executives of this state, `fddi_mac` returns to the `IDLE` state where it waits for the next interrupt.

ENCAP state

This state is entered when `fddi_mac` receives a Service Data Unit (**SDU**) from **LLC**. This event is delivered in the form of a **stream interrupt** on the port arriving from **LLC**, as described in the *Model Interfaces* section of this chapter. The primary actions of this state are to acquire the arriving data and associated control information and to use these to create a **MAC** frame suitable for transmission to a peer **MAC** entity via **PHY**. The resulting **MAC** frame is enqueued until a later time when it becomes eligible for transmission in the `TX_DATA` state.

The first actions implemented in this state serve to obtain the OPNET packet and **ici** that represent the service data unit, and the control fields that comprise the frame transmission request. These are placed in the variables `pdu_ptr`, and `ici_ptr`, respectively. The following statements extract the control information fields that specify service class (asynchronous or synchronous), destination address, and for asynchronous frames, priority class and the class of the token that will be issued after transmission (restricted or non-restricted). These values are placed in the variables `svc_class`, `dest_addr`, `req_pri`, and `req_tk_class`, respectively.

The passed **SDU** and the control information are used to form a **MAC** frame that is held in the variable `mac_frame_ptr`. The fields of the frame, as defined in the packet format `"fddi_mac_fr"` are assigned. The **SDU** that is to be communicated to an **LLC** in a remote station is encapsulated in the `"info"` field of the new frame. For asynchronous requests, the requested token class and priority fields are also assigned. The frame control field `"fc"` is set so that other stations will recognize the arriving packet as a frame rather than a token, and finally the frame is inserted into the fifo which queues transmission requests.

The remaining actions of the ENCAP state are related to the token acceleration mechanism mentioned earlier. In this mechanism, stations that wish to use the token register their need by calling the procedure *fddi_tk_register()*, which is defined in the function block. A station must call this procedure as it transitions from having no data to send to having data to send. There is no need however for a station to register if it is currently registered. The state variable tk_registered is used to prevent unnecessary registrations. If the token is currently blocked when *fddi_tk_register()* is called, it will be reinserted into the ring at the appropriate location so that transmission requests may be serviced.

RCV_TK state

This state is entered by fddi_mac upon receipt of a token from PHY. This event corresponds to a stream interrupt from PHY, and subsequent acquisition of a packet with the proper frame control field. These conditions are represented by the TK_RECEIVED transition conditional that departs from the IDLE state.

The first actions taken in this state obtain the class of the token (restricted or non-restricted), and in the case of a restricted token, the address of the station for which the token is usable is also extracted. A variable, tk_usable, which indicates at the end of the RCV_TK state executives, whether the token may be used by this station, is initially set to 0.

A series of conditions are tested in order to determine if the token can be considered usable. The first condition is that there must be at least one frame enqueued for transmission. If the first frame enqueued (the one at the head of the queue) is synchronous, then the token is necessarily usable and no further criteria need be met. If instead the frame is asynchronous, then in order for the token to be usable, Late_Ct must be zero, the token class must be non-restricted or the station must be involved in the restricted exchange, and finally the frame's priority class must be high enough that the corresponding THT threshold (given by the T_Pri array) is not exceeded by the current value of TRT.

When the token is captured by a station, regardless of whether it is usable or not, timer adjustments must be made. In the case of a usable and early token, the contents of TRT are transferred to THT and the THT timer is disabled. TRT is reset to time the new rotation of the token. Also, the self interrupt previously associated with the expiration of TRT is canceled and a new one is requested to correspond to the new setting of the timer. If, on the other hand, the token is late but usable for a synchronous transmission request, then THT is set to its expired value and disabled (this will prevent asynchronous transmissions from occurring when the TX_DATA state is entered), and Late_Ct is cleared. In the case where the token is not usable but is early, the THT timer is reset and a new self interrupt is requested to replace the previously scheduled one. If instead the token is late, then the only action taken is to set Late_Ct to zero. In either case, if the token is not usable, the variable accum_bandwidth, which keeps track of bandwidth consumption since the arrival of the token, is set to the station latency so that the token will be appropriately delayed when for-

warded to the next station.

TX_DATA state

This state is entered by `fddi_mac` when the token is captured and a determination is made that it is usable. This determination is made in the RCV_TOKEN state according to the logic presented above. The role of the TX_DATA state is to dequeue and send frames into the ring until the token is no longer usable by this station, at which time it is forwarded down-stream.

As specified in the FDDI standard, frames are dequeued in a first-in-first out order and may not be transmitted out of order, regardless of class of service. In order to simplify the implementation, advantage is taken of the ability to schedule the transfer of packets at arbitrary future times. Thus, once entered, the TX_DATA state dequeues as many frames as can be sent according to the prescribed transmission rules, and forwards these with appropriate delays. Simulation time does not advance during this processing, and so the progress of the THT timer is emulated by using an accumulator variable, `tht_value`. From the point of view of entities receiving the packets, all events are perceived as though the packets were individually sent at distinct times since the Simulation Kernel delivers each packet at separate instants. This method avoids complexity in the STD, and significantly reduces the number of simulation events, which in turn, shortens run times.

In order to keep track of the amount of time spent transmitting to date, so that new transmissions can be properly scheduled, the variable `accum_bandwidth` is increased at each frame transmission by the frame transmission time. This variable is initialized to zero at the top of TX_DATA. A separate accumulator, `accum_sync`, keeps track of only synchronous bandwidth, since a fixed limit is imposed on this type of transmission.

The central element of TX_DATA is a transmission loop whose main condition for continuation is that there still are frames in the input queue. Other exit conditions for the loop are tested within its body. At the top of the loop, the first frame in the queue is removed and its service class is extracted. Depending on whether the frame is synchronous or asynchronous, it is processed differently. Synchronous frames are allowed to be transmitted provided that their transmission does not cause the station's synchronous bandwidth allocation to be exceeded. This test is therefore performed before transmitting the frame. The test involves computing the frame's transmission time based on its length and the transmission data rate. This transmission time is stored in the variable `tx_time` which is added to the variable `accum_sync`. The sum cannot exceed the state variable `sync_bandwidth`, set at initialization, if the frame is to be sent. If the frame cannot be transmitted, it is replaced at the head of the queue and because no further transmission requests can be honored (the frames must be served in a FIFO order), the transmission loop is exited. If instead, there is sufficient remaining synchronous bandwidth for the frame to be transmitted, it is scheduled for transmission with a delay comprising the already consumed bandwidth and the inter-station propagation delay (transmission delay of

each frame is accounted for at the time where a frame is received by its destination). Also, the accumulators `accum_bandwidth` and `accum_sync` are increased to reflect the new transmission.

For asynchronous frames, transmission is allowed if the value of THT represented by the `tht_value` variable has not exceeded `Fddi_T_Opr` and the priority level of the frame has a corresponding threshold which is not exceeded by `tht_value`. Unlike synchronous frames, asynchronous frames that meet these criteria are allowed to be sent even if the criteria are violated during the frame transmission. Thus it is possible for an asynchronous frame to complete transmission and have `tht_value` exceeding `Fddi_T_Opr`. However, this will be the last asynchronous frame transmission.

Asynchronous frames that carry a requested token class of *restricted* may cause `fddi_mac` to enter a restricted transmission mode. Similarly, frames that specify a non-restricted token, will cause the process to exit restricted transmission mode. This is currently the only method to affect the restricted transmission status of `fddi_mac`. In restricted mode, the token that is issued after transmission is usable only by a specifically designated peer station on the ring.

As with synchronous frames, the asynchronous frame, if transmittable, is forwarded with a delay that reflects already consumed bandwidth and propagation delay. Also, the `accum_bandwidth` variable and `tht_value` are increased to reflect the new transmission.

Finally, if after exiting the transmission loop, the station has no more frames to transmit (the queue is empty), the station must deregister its interest in the token so that if all stations in the network are data-less, the token can be blocked as part of the token acceleration mechanism mentioned earlier.

FR_RCV state

In this state a frame has been received from PHY. This determination is made in the `IDLE` state which responds to a **stream interrupt** arriving from PHY and analyzes the frame control field, `"fc"`, of the arriving packet. The only action performed in `FR_RCV` is to extract the source address of the packet so that a determination can be made with regard to stripping the frame from the ring. This decision corresponds to the two transitions that depart from `FR_RCV`, which lead either to `FR_STRIP` or `FR_REPEAT`.

FR_REPEAT state

In this state, a frame has been received that was originated by a station other than this one. The FDDI specification calls for the frame to be repeated until it reaches its originating station. However, in simulation, there is no need for the frame to proceed beyond its destination unless group addresses are being used.

Therefore, if the frame's destination matches this station's address, the frame is effectively stripped from the ring. Also, the LLC data encapsulated within the "info" field of the frame is decapsulated. This data is forwarded to LLC with a delay equal to its transmission time, since this is not accounted for upon transmission. An ici is composed which supplies the source and destination address values to LLC.

In the case where the frame's destination is not this station, the frame is repeated onto the ring, and propagation delay and station latency are accounted for.

FR_STRIP state

In this state a frame has been received that was originated by this station. The FDDI specification calls for the frame to be stripped from the ring. The frame is therefore discarded rather than repeated and fddi_mac returns to the IDLE state to await the next interrupt.

FDDI.4 Example Usage

This section describes an example model that encompasses the FDDI MAC model. First, each file that is part of the FDDI example models is explained. Then, the fddi_station node is described where a MAC entity is placed within a traffic source, a traffic sink, and a PHY entity. Then, this node is used to define 32 stations in an FDDI ring described in the *FDDI.4.2 Network Description* section.

FDDI.4.0 Files

This section gives a brief overview of the files found in the <*opdir*>/stdmod/-fddi directory. The models should be easily understandable by the user who wishes to analyze their internals. It is also possible to treat these abstractly and simply work with their parameters.

The files are listed in alphabetical order:

fddi.ef	An environment file specifying the values of simulation attributes for the fddi_net_n simulations.
fddi.os	An output scalar file that contains measurements of throughput and mean end-to-end delay for a range of TTRT values between 0.5 milliseconds and 60 milliseconds. This file can be used in the analysis tool to produce plots of throughput or delay versus TTRT. This file was produced by running the fddi_script shell script.

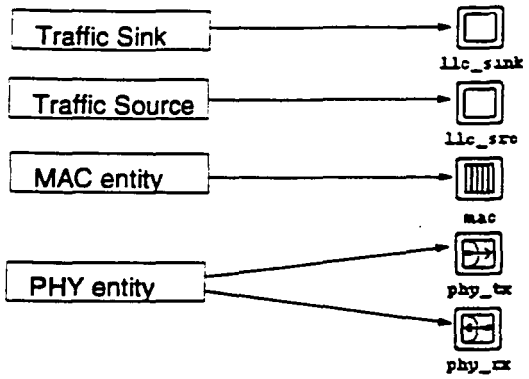
`fddi.pb.m`	A probe file containing specifications of data collection. This probe file can be optionally specified with the `"probe"` environment attribute when executing `fddi_net_n.sim` simulations.
`fddi_build.em.c`	An Ems based application that can be used to generate rings constructed with the `fddi_station` node model. This application can be compiled with the `m3_mkems` program described in the *OPNET External Interfaces Manual / 6.0*. When executed, it accepts a single argument which is the number of stations in the ring.
`fddi_gen.pr.m`	A process model that provides a simple example of higher-layer interfacing with the `fddi_mac` process. This process acts as a poisson frame source and has several attributes that can be modified to control its rate of frame generation, the size of generated frames, and the destination addresses for these frames. When compiled, this process generates the additional files `fddi_gen.pr.c` and `fddi_gen.pr.o`.
`fddi_llc_fr.pf.m`	A packet format specification for frames generated by the `fddi_gen` process model and passed to `fddi_mac`. These frames are encapsulated in the `"info"` field of frames that travel on the FDDI ring.
`fddi_mac.pr.m`	A process model that represents the MAC entity of an fddi station. This process model must operate within a queue module and interfaces with the lower layer PHY entity and the higher layer LLC entity via packet streams and Ici structures. When compiled, this process generates the additional files `fddi_mac.pr.c` and `fddi_mac.pr.o`.
`fddi_mac_fr.pf.m`	A packet format specification for frames formed by the `fddi_mac` process. These frames are the ones passed between MAC entities on the FDDI ring.
`fddi_mac_req.ic.m`	An Ici format that specifies the control information that may be passed by the LLC to the MAC when transmission requests are generated.
`fddi_mac_tk.pf.m`	A packet format specification used to represent a token that circulates on the FDDI ring.

`fddi_mac_ind.ic.m`	An `ici` format that specifies the control information that may be passed by **MAC** to **LLC** when data is received by **MAC** and provided to the local **LLC** entity.
`fddi_net_32.nt.m`	A network model containing 32 `fddi_station` nodes arranged in a ring. This model is produced by executing the `Emu` program `fddi_build.em.x`.
`fddi_script`	A C Shell script that executes the `fddi_net_32` model 18 times for a range of **TTRT** values between 0.5 milliseconds and 60.0 milliseconds.
`fddi_sink.pr.m`	A simple process that acts as a place holder for user-defined higher level processes that would receive data from **MAC**. This process simply discards packets while maintaining and reporting a few statistics related to ring throughput and delay. When compiled, this process generates the files `fddi_sink.pr.c` and `fddi_sink.pr.o`.
`fddi_station.nd.m`	An example node model centered around the `fddi_mac` process. It comprises a transmitter and a receiver representing the **PHY** entity, a queue representing the **MAC** entity, and two processors that together represent the **LLC** entity. This model is the basic building block for the `fddi_net_n` network models.
`propdel_zero.ps.c`	A pipeline model constructed to force point-to-point links used in the FDDI ring model to use a propagation delay of zero within the transceiver pipeline, thus allowing propagation delay to be modeled at a higher level. When compiled, this process model produces the file `propdel_zero.ps.o`.
`txdel_zero.ps.c`	A pipeline model constructed to force point-to-point links used in the FDDI ring model to use a transmission delay of zero within the transceiver pipeline, thus allowing transmission delay to be modeled at a higher level. When compiled, this process model produces the file `txdel_zero.ps.o`.

FDDI.4.1 Node Description: fddi_station

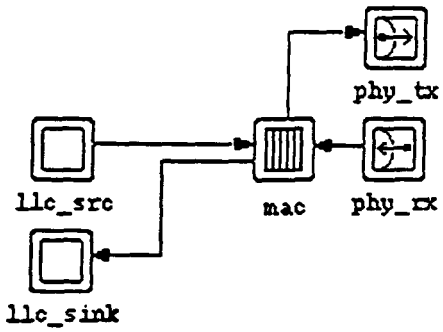
This section describes the basic component of the FDDI network model, which is a model of an FDDI station, including a traffic source, a traffic sink, a MAC entity, and a PHY entity. These entities are modeled in terms of the modules that are provided by OPNET's Node Editor to form a basic example of an FDDI-based communication node called `fddi_station`. The correspondence between the entities is shown in the following diagram.

Mapping of FDDI station entities to OPNET Modules



The modules are connected via OPNET packet streams over which tokens or frames can be forwarded. Both tokens and data frames are represented with OPNET packets defined by the frame formats `fddi_llc_fr`, `fddi_mac_fr`, and `fddi_mac_tk`, described previously in this chapter. The `mac` processor receives inputs from the processor `llc_src` and the receiver module `phy_rx`. The processor `llc_sink` and the transmitter `phy_tx` receive inputs from `mac`. The layout of the node is shown below.

fddi_station Node Model



The `phy_tx` and `phy_rx` modules serve as the physical interface to the ring transmission medium. Frames and tokens are received by `mac` from `phy_rx`, which is connected to the point-to-point link emanating from the next up-stream neighbor. Similarly, frames and tokens are forwarded by `mac` to `phy_tx`, which is connected to the link leading to the next down-stream neighbor. In most OPNET models, the primary attributes of the point-to-point transmitter and receiver would be the data rate assignments for their channel objects. However, in this FDDI model, as mentioned earlier, the computation of transmission delay returns a fixed value of zero, thereby making the `"data rate"` attribute irrelevant.

As shown in the mapping above, the queue module `mac` occupies the place of the **MAC** entity in the station and has the responsibility of token and timer management, frame capture and repetition, and queueing of transmission requests. The behavior of this queue is prescribed by the `fddi_mac` process model described in a later section of this chapter.

The processor `llc_src` is so named because of its physical relationship with `mac` to which it provides frames for transmission. It does not have the functionality of an actual **LLC** beyond correctly interfacing with the **MAC** entity. While it is primarily intended to serve as an example of how to interface with `mac`, it may also be used as a convenient but simple message source for FDDI models. Its behavior is prescribed by the `fddi_gen` process model.

Finally, the processor `fddi_sink` provides a simple destination for frames captured by and addressed for the station, and forwarded by the `mac` processor. It serves as a place holder for higher level processes that may be developed as part of larger modeling efforts. Its actions, which consist primarily of packet disposal and statistic collection, are specified by the process model `fddi_sink`.

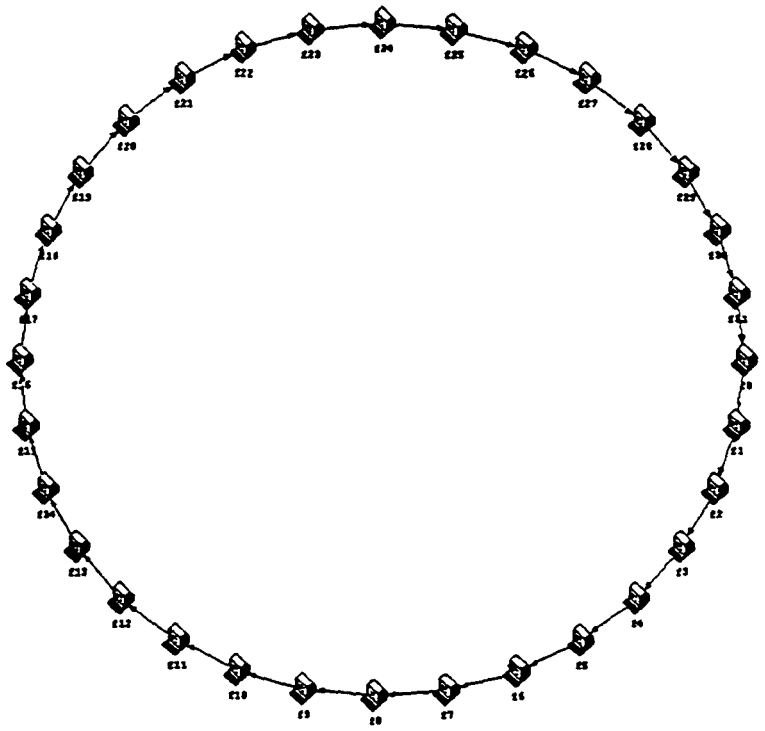
FDDI.4.2 Network Description: fddi_net_32

The ring topology and number of stations are the primary specifications comprised in the network level model, since the inter-station propagation delay has been

made a global simulation attribute, common to all inter-station connections. A sample network model, `fddi_net_32` containing 32 stations within a subnetwork is provided to illustrate the usage of the lower level models.

Because FDDI rings may often contain large numbers of stations, it is convenient to generate the associated network models in an automated fashion, specifying only the number of stations. An application called `fddi_build.em.x`, based on the External Model Access (Ema) package, is provided for this purpose. This application is almost entirely derived from the `ring_build` program described in *Chapter Ema* of the *OPNET External Interfaces Manual / 6.0*, and consequently is only minimally documented here. `fddi_build.em.x` accepts a single required argument which is the number of stations on the network. It arranges stations of the type `fddi_station` in a circular ring and creates links between adjacent stations. This program was used to generate the example network `fddi_net_32` which is shown in the following diagram.

fddi_net_32 Network Model



The source code for the network building program is supplied in the file `fddi_build.em.c` so that the form of the models may be conveniently modified. Some parameters that can easily be changed are the icon used to represent stations, the

strings used to name the stations, and the physical dimensions and shape of the network.

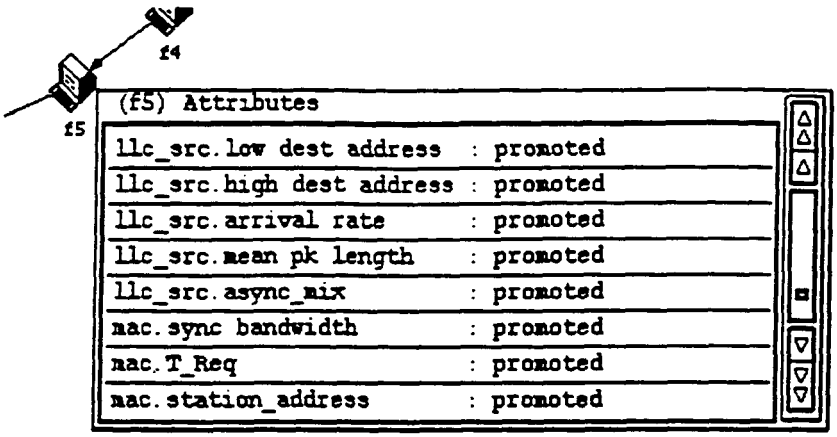
An important specification that occurs at the network level (and therefore in the **Ema** program which generates the network model) is the assignment of procedures that model the operation of the links. There is currently no dedicated physical layer object in OPNET for modeling ring architectures. Consequently, point-to-point links are used and their internal operation and transmission timing mechanisms are modified to reflect the behavior of FDDI interfaces. The procedures that implement the point-to-point link models are called Transceiver Pipeline procedures and these are discussed in detail in the *Chapter Comec* chapter of the *OPNET Modeling Manual / 2.0.*

Two mechanisms in the point-to-point link, used for modeling transmission delay and propagation delay, have been modified to implement the FDDI ring physical layer. The default point-to-point link pipeline uses a procedure called ***dpt_txdel()*** to compute transmission delay on the basis of a packet's length, and the **"data rate"** attribute of the channel of interest. The transmission delay computed by ***dpt_txdel()*** is used by the Simulation Kernel to schedule the delivery of the transmitted packet at the output of the receiver in the link's destination node. In other words, no entity in the destination node is aware of the arrival of the packet until its last bit has arrived. This is clearly inappropriate for FDDI, where frames and tokens must be repeated after only a small number of symbols have been received by a station. The approach used here is to create a new procedure called ***txdel_zero()***, which always returns a transmission delay of zero, and to leave the modeling of station latency to the **MAC** process model, since all frames and tokens must be made known to this entity. The source code for the **txdel_zero** pipeline procedure is provided in the file **<*opdir*>/stdmod/fddi/txdel_zero.ps.c**.

Because the assumption has been made that inter-station propagation delays are uniform across the network, the pipeline mechanism used to inject propagation delay is also disabled. This is done by inserting the pipeline procedure ***propdel_zero()*** in place of the default propagation delay model ***dpt_propdel()***. Propagation delay modeling is instead done as packets are forwarded by the **MAC** entity. As with **txdel_zero**, source code is provided in the **<*opdir*>/stdmod/fddi** directory.

Once constructed, the Network Editor allows a number of attributes to be set for each station via menus. It is also possible (this is the default and often most convenient method) to set these attributes via an environment file which is interpreted at the time of simulation. The attributes that are promoted from lower level models and are available on a per station basis are illustrated below. Each attribute is discussed following the illustration.

Some FDDI Station Attributes viewed in Network Editor



llc.src.low dest address

This attribute is promoted from the **fddi_station** node model and originates in the **fddi_gen** process model. It can be used to control the lower bound for destination assignment when the **llc_src** module within the station generate new frames for transmission. It affects the addressing of both synchronous and asynchronous frames.

llc.src.high dest address

This attribute is promoted from the **fddi_station** node model and originates in the **fddi_gen** process model. It can be used to control the upper bound for destination assignment when the **llc_src** module within the station generates new frames for transmission. It affects the addressing of both synchronous and asynchronous frames.

llc.src.arrival rate

This attribute is promoted from the **fddi_station** node model and originates in the **fddi_gen** process model. It can be used to control the rate at which the **llc_src** module within the station generates frames

for transmission. The value of this attribute specifies the aggregate rate comprising both synchronous and asynchronous traffic, and is specified in frames per second.

`llc.src.mean pk length`
This attribute is promoted from the `fddi_station` node model and originates in the `fddi_gen` process model. It can be used to control the length of both asynchronous and synchronous frames generated for transmission by the `llc_src` module.

`llc.src.async_mix` This attribute is promoted from the `fddi_station` node model and originates in the `fddi_gen` process model. Its value varies between zero and one and specifies the proportion of asynchronous frames generated by the `llc_src` module within the station. A value of zero specifies that `llc_src` shall queue only synchronous frames for transmission while a value of one specifies instead that it shall generate only asynchronous frames.

`mac.sync bandwidth` This attribute is promoted from the `fddi_station` node model and originates in the `fddi_mac` process model. Its value varies between zero and one and specifies the proportion of synchronous bandwidth allocated to this station relative to the total synchronous allocation for the entire ring. The sum of this attribute for all the stations in the ring should not exceed one (note, that at present the model does not enforce this).

`mac.T_Req` This attribute is promoted from the `fddi_station` node model and originates in the `fddi_mac` process model. It represents the requested value of TTRT on the part of the station. The `fddi_mac` process model will select the minimum value of this attribute among all stations to install in the variable `Fddi_T_Opr`, which holds the operative value of TTRT.

`mac.station_address` This attribute is the unique identification of each station, and is used for addressing and stripping transmitted frames in the ring.

APPENDIX D. DESCRIPTION OF THE SIMULATION PARAMETERS

This appendix comprises a complete list of the chosen simulation model attributes for the simulations described in the main text. The parameters are identified by the Figure numbers which show the results of this particular set of simulations. The chosen parameter are described in more detail for the first set. In subsequent sets only the differences to the first set are shown.

Simulation Parameters Figure 4.2 and Figure 4.3

- duration = 0.4 (seconds)
- seed = 121, 234, 310, 444
- top.ATM_1.src.interarrival args = 1.00474E-06 (1/packet arrival rate, =422Mb/s)
- top.ATM_1.proc.VPI_SET = 3
- top.ATM_1.proc.VECTOR_STAT_ENABLE = disabled
- top.ATM_1.src.interarrival pdf = constant
- top.ATM_1.rcv[0].data rate = 500E+06 (bits/second)
- top.ATM_1.xmt[0].data rate = 500E+06 (bits/second)
- top.ATM_4.src.interarrival args = 1.00474E+06 (=no packets)
- top.ATM_4.proc.VPI_SET = 4
- top.ATM_4.proc.VECTOR_STAT_ENABLE = disabled
- top.ATM_4.src.interarrival pdf = constant

- top.ATM_4.rcv[0].data rate = 500E+06 (bits/second)
- top.ATM_4.xmt[0].data rate = 500E+06 (bits/second)
- top.ATM_2.atm_sw.VPI_ATM_LOCAL = 3
- top.ATM_2.atm_sw.VPI_ATM_REMOTE = 4
- top.ATM_2.atm_sw.VPI_FDDI_LOCAL = 1
- top.ATM_2.atm_sw.VPI_FDDI_REMOTE = 2
- top.ATM_2.atm_sw.STAT_ENABLE = enabled
- top.ATM_2.xmt_fddi[0].data rate = 80E+06 (bits/second)
- top.ATM_2.rcv_fddi[0].data rate = 80E+06 (bits/second)
- top.ATM_2.xmt_atm_nd[0].data rate = 500E+06 (bits/second)
- top.ATM_2.rcv_atm_nd[0].data rate = 500E+06 (bits/second)
- top.ATM_2.xmt_atm_sw[0].data rate = 500E+06 (bits/second)
- top.ATM_2.rcv_atm_sw[0].data rate = 500E+06 (bits/second)
- top.ATM_3.atm_sw.VPI_ATM_LOCAL = 4
- top.ATM_3.atm_sw.VPI_ATM_REMOTE = 3
- top.ATM_3.atm_sw.VPI_FDDI_LOCAL = 2
- top.ATM_3.atm_sw.VPI_FDDI_REMOTE = 1
- top.ATM_3.atm_sw.STAT_ENABLE = disabled
- top.ATM_3.xmt_fddi[0].data rate = 80E+06 (bits/second)
- top.ATM_3.rcv_fddi[0].data rate = 80E+06 (bits/second)
- top.ATM_3.xmt_atm_nd[0].data rate = 500E+06 (bits/second)
- top.ATM_3.rcv_atm_nd[0].data rate = 500E+06 (bits/second)
- top.ATM_3.xmt_atm_sw[0].data rate = 500E+06 (bits/second)
- top.ATM_3.rcv_atm_sw[0].data rate = 500E+06 (bits/second)

- top.FDDI_1.fddi_atm_link.xtm_atm[0].data rate = 80E+06 (bits/second)
- top.FDDI_1.fddi_atm_link.rcv_atm[0].data rate = 80E+06 (bits/second)
- top.FDDI_1.fddi_atm_link.bridge_proc.VPI_SET = 1
- top.FDDI_1.fddi_atm_link.bridge_proc.STAT_ENABLE = enabled
- top.FDDI_1.fddi_atm_link.mac.station_address = 0
- top.FDDI_1.fddi_atm_link.mac.ring_id = 1
- top.FDDI_1.fddi_atm_link.mac.sync bandwidth = 0.5 (*100%)
- top.FDDI_1.fddi_atm_link.mac.T_Req = 4.0 (seconds)
- top.FDDI_1.vbr_station.llc_src.traffic_dist = exponential
- top.FDDI_1.vbr_station.llc_src.vbr_gen_seed_I = 911, 284, 595, 412
- top.FDDI_1.vbr_station.llc_src.vbr_gen_seed_II = 810, 212, 611, 693
- top.FDDI_1.vbr_station.llc_src.arrival rate = 700–1600 (packets/second, step = 100)
- top.FDDI_1.vbr_station.llc_src.mean pk length = 32000 (bits)
- top.FDDI_1.vbr_station.llc_src.idle_dist = exponential
- top.FDDI_1.vbr_station.llc_src.idle_dist_arg = 0.002 (seconds)
- top.FDDI_1.vbr_station.llc_src.busy_dist = exponential
- top.FDDI_1.vbr_station.llc_src.busy_dist_arg = 0.01 (seconds)
- top.FDDI_1.vbr_station.llc_src.low dest address = 1
- top.FDDI_1.vbr_station.llc_src.high dest address = 1
- top.FDDI_1.vbr_station.llc_src.async_mix = 1.0 (only asynchronous data)
- top.FDDI_1.vbr_station.llc_src.dest_ring_id = 2
- top.FDDI_1.vbr_station.mac.T_Req = 4.0 (seconds)
- top.FDDI_1.vbr_station.mac.station_address = 1

- top.FDDI_1.vbr_station.mac.ring_id = 1
- top.FDDI_1.vbr_station.mac.sync bandwidth = 0.0 (*100%)
- top.FDDI_1.cbr_station.llc_src.arrival rate 0.0 (packets/second)
- top.FDDI_1.cbr_station.llc_src.mean pk length = 32000 (bits)
- top.FDDI_1.cbr_station.mac.sync bandwidth = 0.5 (*100%)
- top.FDDI_1.cbr_station.mac.T_Req = 0.001 (seconds)
- top.FDDI_1.cbr_station.llc_src.low dest address = 2
- top.FDDI_1.cbr_station.llc_src.high dest address = 2
- top.FDDI_1.cbr_station.llc_src.traffic_dist = constant
- top.FDDI_1.cbr_station.llc_src.async_mix = 0.0 (only synchronous data)
- top.FDDI_1.cbr_station.llc_src.dest_ring_id = 2
- top.FDDI_1.cbr_station.mac.station_address = 2
- top.FDDI_1.cbr_station.mac.ring_id = 1
- top.FDDI_2.vbr_station.llc_src.traffic_dist = exponential
- top.FDDI_2.vbr_station.llc_src.vbr_gen_seed_I = 191, 186, 343, 543
- top.FDDI_2.vbr_station.llc_src.vbr_gen_seed_II = 333, 432, 999, 842
- top.FDDI_2.vbr_station.llc_src.arrival rate = 0.0 (packets/second)
- top.FDDI_2.vbr_station.llc_src.mean pk length = 32000 (bits)
- top.FDDI_2.vbr_station.llc_src.idle_dist = exponential
- top.FDDI_2.vbr_station.llc_src.idle_dist_arg = 0.002 (seconds)
- top.FDDI_2.vbr_station.llc_src.busy_dist = exponential
- top.FDDI_2.vbr_station.llc_src.busy_dist_arg = 0.01 (seconds)
- top.FDDI_2.vbr_station.llc_src.low dest address = 1
- top.FDDI_2.vbr_station.llc_src.high dest address = 1

- top.FDDI_2.vbr_station.llc_src.async_mix = 1.0 (only asynchronous data)
- top.FDDI_2.vbr_station.llc_src.dest_ring_id = 1
- top.FDDI_2.vbr_station.mac.T_Req = 4.0 (seconds)
- top.FDDI_2.vbr_station.mac.station_address = 1
- top.FDDI_2.vbr_station.mac.ring_id = 2
- top.FDDI_2.vbr_station.mac.sync bandwidth = 0.0 (*100%)
- top.FDDI_2.cbr_station.llc_src.arrival rate = 0.0 (packets/second)
- top.FDDI_2.cbr_station.llc_src.mean pk length = 32000 (bits)
- top.FDDI_2.cbr_station.mac.sync bandwidth = 0.5 (*100%)
- top.FDDI_2.cbr_station.mac.T_Req = 0.001 (seconds)
- top.FDDI_2.cbr_station.llc_src.low dest address = 2
- top.FDDI_2.cbr_station.llc_src.high dest address = 2
- top.FDDI_2.cbr_station.llc_src.traffic_dist = constant
- top.FDDI_2.cbr_station.llc_src.async_mix = 0.0 (only synchronous data)
- top.FDDI_2.cbr_station.llc_src.dest_ring_id = 1
- top.FDDI_2.cbr_station.mac.station_address = 2
- top.FDDI_2.cbr_station.mac.ring_id = 2
- top.FDDI_2.fddi_atm_link.mac.station_address = 0
- top.FDDI_2.fddi_atm_link.mac.ring_id = 2
- top.FDDI_2.fddi_atm_link.mac.sync bandwidth = 0.5 (*100%)
- top.FDDI_2.fddi_atm_link.mac.T_Req = 4.0 (seconds)
- top.FDDI_2.fddi_atm_link.bridge_proc.VPI_SET = 2
- top.FDDI_2.fddi_atm_link.bridge_proc.STAT_ENABLE = enabled
- top.FDDI_2.fddi_atm_link.xtm_atm[0].data rate = 80E+06 (bits/second)

- top.FDDI_2.fddi_atm_link.rcv_atm[0].data rate = 80E+06 (bits/second)
- station_latency = 1E-07 (seconds)
- prop_delay = 3.3E-06 (seconds)
- accelerate_token = 1 (Enables the token acceleration mechanism)
- spawn station = 1 (station with station_id = 1 issues the token at the beginning of the simulation)

Simulation Parameters Figure 4.4

- top.FDDI_1.vbr_station.llc_src.busy_dist_arg = 0.02 (seconds)

Simulation Parameters Figure 4.5

- top.FDDI_1.vbr_station.llc_src.busy_dist_arg = 0.02 (seconds)
- top.FDDI_1.vbr_station.llc_src.arrival rate = 1400–3200 (packets/second, step = 200)

Simulation Parameters Figure 4.6

- duration = 0.8 (seconds)
- seed = 444
- top.FDDI_1.vbr_station.llc_src.vbr_gen_seed_I = 412
- top.FDDI_1.vbr_station.llc_src.vbr_gen_seed_II = 693
- top.FDDI_1.vbr_station.llc_src.arrival rate = 1875
- top.FDDI_1.vbr_station.llc_src.idle_dist_arg = 0.002–0.02 (seconds)
- top.FDDI_1.vbr_station.llc_src.busy_dist_arg = 0.01–0.1 (seconds, to keep burstiness = 5)

Simulation Parameters Figure 4.7

- top.ATM_1.src.interarrival args = 9.953E-07 (1/packet arrival rate)

Simulation Parameters Figure 4.8

- top.FDDI_1.vbr_station.llc_src.arrival rate = 1600 (packets/second)
- top.FDDI_2.cbr_station.llc_src.arrival rate 160–1600 (packets/second, step = 160)

Simulation Parameters Figure 4.9

- top.FDDI_1.cbr_station.llc_src.arrival rate 300 (packets/second)
- top.FDDI_1.vbr_station.llc_src.arrival rate = 1500 (packets/second)
- top.ATM_1.src.interarrival args = 1.082E-06, 1.055E-06, 1.029E-06, 1.005E-06, 9.815E-07, 9.593E-07, 9.38E-07, 9.177E-07 (1/packet arrival rate, =392-462Mb/s)
- top.ATM_2.xmt_fddi[0].data rate = 40E+06–110E+06 (bits/second, step = 10E+06)
- top.ATM_2.rcv_fddi[0].data rate = 40E+06–110E+06 (bits/second, step = 10E+06)
- top.ATM_3.xmt_fddi[0].data rate = 40E+06–110E+06 (bits/second, step = 10E+06)
- top.ATM_3.rcv_fddi[0].data rate = 40E+06–110E+06 (bits/second, step = 10E+06)
- top.FDDI_2.fddi_atm_link.xtm_atm[0].data rate = 40E+06–110E+06 (bits/second, step = 10E+06)
- top.FDDI_2.fddi_atm_link.rcv_atm[0].data rate = 40E+06–110E+06 (bits/second, step = 10E+06)