Rochester Institute of Technology

RIT Scholar Works

Theses

8-1-2000

The Design, modeling and simulation of switching fabrics: For an ATM network switch

Dmitriy Molokov

Follow this and additional works at: <https://scholarworks.rit.edu/theses>

Recommended Citation

Molokov, Dmitriy, "The Design, modeling and simulation of switching fabrics: For an ATM network switch" (2000). Thesis. Rochester Institute of Technology. Accessed from

This Thesis is brought to you for free and open access by RIT Scholar Works. It has been accepted for inclusion in Theses by an authorized administrator of RIT Scholar Works. For more information, please contact ritscholarworks@rit.edu.

THE DESIGN, MODELING AND SIMULATION OF SWITCHING
FABRICS
FOR AN ATM NETWORK SWITCH

by

Dmitriy Molokov

A Thesis Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE
in
Computer Engineering

Approved by:

Principal Advisor: _____
Dr. Muhammad Shaaban

Committee Member: _____
Dr. James Heliotis

Committee Member: _____
Dr. Andreas Savakis

Department of Computer Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
August, 2000

RELEASE PERMISSION FORM

Rochester Institute of Technology

The Design Modeling, and Simulation of Switching Fabrics for an ATM Network Switch

I, Dmitriy Molokov, hereby grant permission to any individual or organization to reproduce this thesis in whole or in part for non-commercial and non-profit purposes only.

Dmitriy V. Molokov

8/10/2000

Date

Abstract

The requirements of today's telecommunication systems to support high bandwidth and added flexibility brought about the expansion of (Asynchronous Transfer Mode) ATM as a new method of high-speed data transmission. Various analytical and simulation methods may be used to estimate the performance of ATM switches. Analytical methods considerably limit the range of parameters to be evaluated due to extensive formulae used and time consuming iterations. They are not as effective for large networks because of excessive computations that do not scale linearly with network size. On the other hand, simulation-based methods allow determining a bigger range of performance parameters in a shorter amount of time even for large networks. A simulation model, however, is more elaborate in terms of implementation. Instead of using formulae to obtain results, it has to operate software or hardware modules requiring a certain amount of effort to create. In this work simulation is accomplished by utilizing the ATM library -- an object oriented software tool, which uses "software chips" for building ATM switches. The distinguishing feature of this approach is cut-through routing realized on the bit level abstraction treating ATM protocol data units, called cells, as groups of 424 bits. The arrival events of cells to the system are not instantaneous contrary to commonly used methods of simulation that consider cells as instant messages. The simulation was run for basic multistage interconnection network types with varying source arrival rate and buffer sizes producing a set of graphs of cell delays, throughput, cell loss probability, and queue sizes. The techniques of rearranging and sorting were considered in the simulation. The results indicate that better performance is always achieved by bringing additional stages of elements to the switching system.

Contents

List of Figures	iii
List of Tables	vi
Glossary	viii
1 Introduction	1
2 ATM Standard	3
2.1 Evolution of Digital Networks.	3
2.1.1 Implementation of digital networks.	6
2.2 The ATM Protocol.	7
2.2.1 Physical Layer.	8
2.2.2 ATM Layer.	8
2.2.3 ATM Adaptation Layer.	10
3 Switch Fabrics and Techniques Overview	13
3.1 Switching.	15
3.2 Switch architectures.	16
3.3 Rearrangeable networks.	24
3.4 Sorting networks.	26
3.5 Queuing methods.	28
3.6 Buffer management schemes.	29
4 Simulation Approach	32
4.1 Existing performance evaluation methods.	32
4.2 ATM library.	34
4.2.1 Object organization.	35

4.2.2 Class StorageObject.	40
4.2.3 Class SwElement.	42
4.2.4 Class InSocket.	48
4.2.5 Class OutSocket.	49
4.2.6 Class Queue.	49
4.2.7 Class Wire.	51
4.2.8 Class Clock.	52
4.2.9 Class Manager.	53
4.3 Formal description of the library elements.	54
 5 Dynamic Traffic Generation	 58
5.1 Class Tmanager.	59
5.2 Group of TMSet Classes.	60
5.3 Traffic Manager Utility Functions.	61
5.4 Class Tgen.	64
 6 Simulation Results	 66
6.1 Simulation of unbuffered networks.	68
6.2 Simulation of buffered networks	79
6.3 Simulation of networks under varying arrival rates	90
6.4 Simulation of buffered networks under varying buffer sizes	95
6.5 Conclusions	103
 7 Conclusion	 107
7.1 Summary	107
7.2 Future work	108
 Bibliography	 109
 Appendix A. Source code.	 112
Appendix B. Sample output traces.	145
Appendix C. ATM library guide.	147

List of Figures

2.1 Segregated transport in a narrowband network.	3
2.2 Integrated access in a narrowband network.	4
2.3 A Broadband Integrated Services Digital Network.	5
2.4 Broadband integrated transport.	5
2.5 Multiplexing in synchronous transmission.	6
2.6 Multiplexing in asynchronous transmission.	7
2.7 ATM protocol reference model.	7
2.8 ATM Transmission link.	9
2.9 Structure of an ATM cell.	9
2.10 Original service classes of the ATM Adaptation Layer.	11
3.1 General switch architecture.	13
3.2 Examples of connection in a switch-box.	14
3.3 Functional scheme of a switching element.	14
3.4 An ATM switching example.	16
3.5 Setting of virtual paths and virtual channels in an ATM switch.	16
3.6 Topologically equivalent non-isomorphic networks.	17
3.7 Isomorphic networks.	17
3.8 Functionally equivalent isomorphic networks.	18
3.9 Interconnection patterns.	19
3.10 A series of banyan switch architectures.	22
3.11 Buddy and constrained reachability properties.	23
3.12 An example of conflict in a banyan network.	23
3.13 A part of generic crossbar switch.	24
3.14 Benes rearrangeable network.	25
3.15 Combined sorting-routing network.	26
3.16 Sorting stage for a 16-input banyan network.	27
4.1 Throughput estimated analytically and with a simulation model.	33

4.2 ATM library UML diagramme.	36
4.3 A chain of ATM library objects.	37
4.4 Non-symmetrical switch configuration with complex interstage connections. . . .	38
4.5 Baseline network.	39
4.6 Pseudocode of class's StorageObject function <i>connectsTo()</i>	41
4.7 Setting up of connections among objects of class StorageObject.	42
4.8 Psudocode of function <i>phase3()</i> of class SwElement.	46
4.9 Structure of an object of class Queue.	50
4.10 State machine of one traffic node of a switch-box.	54
4.11 Petri Net of a two-input, two-output switch-box.	56

5.1 The traffic generation part UML diagramme.	58
5.2. Cumulative distribution function for Poisson distribution.	62

Set of results of simulation of unbuffered networks with 16 outputs.

6.1 Simulation time.	69
6.2 Number of conflicts.	70
6.3 Throughput.	71
6.4 Minimum cell delay.	72
6.5 Average cell delay.	73
6.6 Maximum cell delay.	74
6.7 Average queue size calculated only on active queues.	75
6.8 Overall average queue size.	76
6.9 Maximum queue size.	77

Set of results of simulation of buffered networks with 16 outputs.

6.10 Simulation time.	80
6.11 Number of conflicts.	81
6.12 Throughput.	82
6.13 Minimum cell delay.	83
6.14 Average cell delay.	84
6.15 Maximum cell delay.	85

6.16 Average queue size calculated only on active queues.	86
6.17 Overall average queue size.	87
6.18 Maximum queue size.	88

Set of results of simulation of large networks for 256 outputs

6.19 Simulation time.	90
6.20 Number of conflicts.	91
6.21 Throughput.	91
6.22 Minimum cell delay.	92
6.23 Average cell delay.	92
6.24 Maximum cell delay.	93
6.25 Average queue size.	93
6.26 Maximum queue size.	94

Set or results of simulation of 16-output networks with varying buffer size.

6.27 Throughput.	95
6.28 Number of conflicts.	96
6.29 Loss probability.	97
6.30 Minimum cell delay.	98
6.31 Average cell delay.	99
6.32 Maximum cell delay.	100
6.33 Average queue size.	101
6.34 Maximum queue size.	102

6.35 Delay and throughput(carried load) of banyan networks

with and without cut-through routing.	104
---	-----

6.36 Delay and throughput(carried load) of banyan networks

with and without effect of bypass queuing.	104
--	-----

6.37. Throughput of omega network. Markov chain approximation versus simulation.	105
--	-----

List of Tables

4.1 Functional options of an SwElement.	43
Set of results of simulation of unbuffered networks with 16 outputs.	
6.1 Simulation time.	68
6.2 Number of conflicts.	70
6.3 Throughput.	71
6.4 Minimum cell delay.	72
6.5 Average cell delay.	73
6.6 Maximum cell delay.	74
6.7 Average queue size calculated only on active queues.	75
6.8 Overall average queue size.	76
6.9 Maximum queue size.	77
Set of results of simulation of buffered networks with 16 outputs.	
6.10 Simulation time.	80
6.11 Number of conflicts.	81
6.12 Throughput.	82
6.13 Minimum cell delay.	83
6.14 Average cell delay.	84
6.15 Maximum cell delay.	85
6.16 Average queue size calculated only on active queues.	86
6.17 Overall average queue size.	87
6.18 Maximum queue size.	88
Set of results of simulation of large networks for 256 outputs	
6.19 Simulation time.	90
6.20 Number of conflicts.	91
6.21 Throughput.	91
6.22 Minimum cell delay.	92

6.23 Average cell delay.	92
6.24 Maximum cell delay.	93
6.25 Average queue size.	93
6.26 Maximum queue size.	94

Set or results of simulation of 16-output networks with varying buffer size.

6.27 Throughput.	95
6.28 Number of conflicts.	96
6.29 Loss probability.	97
6.30 Minimum cell delay.	98
6.31 Average cell delay.	99
6.32 Maximum cell delay.	100
6.33 Average queue size.	101
6.34 Maximum queue size.	102

Glossary

AAL (page 11). ATM Adaptation Layer. An ATM layer which interfaces the ATM layer with non-ATM protocols and transport methods.

Arrival rate (page 2). Rate, with which cells arrive at an input port.

ATM (page 1). Asynchronous Transfer Mode. A high-speed connection-oriented switching technology that uses fixed length cells and can support multiple types of traffic. It is asynchronous in the sense that cells carrying user data need not be periodic.

ATM Library (page 2). Software tool for simulation of ATM switches.

ATM Layer (page 8). An ATM protocol stack layer, which is responsible for transmitting cells.

Banyan Network (page 18). A multistage routing network.

BaseClass (page 35). Class of the ATM library, from which many other classes inherit.

B-ISDN (page 4). Broadband Integrated Services Digital Network. A high speed digital network standard that integrates voice, data, and other services. B-ISDN transmits at speeds above 1.544 Mbps and operates over ATM.

Blocking (page 16). A case when a connection between an IPC and an OPC cannot be made, due to another connection, blocking its path internally.

Broadband (page 3). Any service that provides transmission channels capable of supporting data rates greater than the ISDN primary rate.

Buddy property (page 22). A property of banyan networks.

CBR (page 11). Constant Bit Rate. An ATM service category that provides a constant rate of data transmission. CBR is used for connections that require a guaranteed amount of bandwidth and low latency.

Cell Delay (page 10). A quality of service parameter specifying the end-to-end time delay experienced by cells on a specific connection.

Cell Delay Variation (page 1, 10). A quality of service parameter, specifying the variance of cell delays in a given connection.

Circuit Switching (page 3). A method of data communications in which a dedicated path is established between two locations prior to the start of the communication process. Digital data is sent as a continuous stream of bits at a guaranteed bandwidth.

Clock (page 35). The driving class of the ATM library.

Clock cycle (page 2). The unit of time used in the ATM library.

ClockedObject (page 36). A class of the ATM library.

CLP (page 10). Cell Loss Priority. A 1-bit field in the header of an ATM cell that is used to determine which cells get discarded first when congestion occurs.

Connectionless (page 11). A type of data transfer, in which information can be exchanged between locations without prior coordination.

Connection-oriented (page 6). A type of data transfer, in which a logical connection is established between the communications endpoints.

Constrained reachability property (page 22). A property of banyan networks.

Crossbar switch (page 16, 24). Switch with the switching elements interconnected in mesh.

CS (page 10). Convergence Sublayer. A sublayer of the ATM adaptation layer in the ATM protocol model. This layer performs functions that are specific to a certain service.

Cumulative distribution function (page 62). Function giving the probability of interarrival time.

Delayed push-out (page 30). Buffering strategy combining backpressure and push-out.

Discrete time simulation (page 2). Simulation, in which time is measured by discrete cycles.

FDM (page 3). The division of a transmission facility into two or more channels by dividing the total available frequency band into several smaller bands, each of which is used as a separate channel.

Gbps (page 6). Giga-Bits Per Second. A unit used to measure the data transmission rate of a connection. A connection with a speed of 1 Gbps is transmitting 10^9 bits each second.

GFC (page 10). Generic Flow Control. The GFC is a 4-bit field found only in the header of a UNI cell. It is intended to be used in defining a simple multiplexing scheme.

HEC (page 8). Header Error Check. An 8-bit field in the header of an ATM cell that contains a code value used to detect and possibly correct errors in the 5-byte cell header.

Head-of-line blocking (page 28). Blocking of a group of cells at the front of a queue.

IDN (page 3). Integrated Digital Network. The name given to the public switched telephone network after it was upgraded from analog to digital technology.

IPC (page 13). Input Controller. The part of an ATM switch fabric, which processes the cell headers and determines which OPC each cell should be addressed to.

IN (page 13). Interconnection Network. The part of an ATM switch fabric, which performs an ordered routing from an IPC to an OPC.

InSocket (page 36). An interfacing module in the ATM library.

ISDN (page 4). Integrated Services Digital Network. A digital network standard that defines a set of services, capabilities, and interfaces supporting an integrated network and user interface.

Kbit/s (page 4). Kilo-Bits Per Second. A unit used to measure the data transmission rate of a connection. A connection with a speed of 1 Kbps is transmitting 10^3 bits each second.

LAN (page 12). Local Area Network.. A data and computer communications network confined to short geographic distances.

Manager (page 35). Class of the ATM library accomplishing management functions.

Narrow band (page 3). Narrowband channel -- channel with low bandwidth.

NNI (page 7). Network Node Interface. The interface between two ATM switches or two ATM networks.

OPC (page 13). Output Controller. The part of an ATM switch fabric, which accepts upto K cells in a given cycle, and provides the cell headers with the appropriate VP/VC identifiers based on lookup table information.

OutSocket (page 36). An interfacing module in the ATM library.

Packet Switching (page 3). A method of data communications in which messages are divided into units called packets. Each packet is then routed through the communications network independently.

Poisson (page 60). Poisson distribution. Traffic distribution applied to the tested switch inputs.

Protocol Data Unit.(page 1). A block of data exchanged between two entities via a protocol.

PTI (page 10). Payload Type. A 2-bit field located in the header of an ATM cell that identifies the type of information contained in the data field.

Queue (page 36). A class of the ATM library.

Rearrangeable networks (page 24). Networks with redundant paths between the inputs and outputs.

Rearrangeable nonblocking (page 17). A network in which the set of existing connections can be rearranged to accommodate a new connection.

Routing table (page 40). Table keeping the routing tags for routing from input ports to output ports in class Manager.

SAR (page 10). Segmentation and Reassembly. A sublayer of the ATM adaptation layer in the ATM protocol model. This layer is responsible for both segmenting and reassembling data units and mapping them to and from fixed length cells.

SDS (page 3). Space Division Switching. Physical separation of transmission media for signals to be switched.

SONET (page 3). Synchronous Optical Network. An international standard that defines a protocol for transmitting data at high speeds over a fiber optic network.

Sorting networks (page 26). Networks sorting the incoming traffic prior to routing.

Speedup (page 28). Effect, in which a network can accept cells from more than one input during a single slot time.

StorageObject (page 36). A polyfunctional class of the ATM library.

Strict-sense nonblocking (page 16). A network in which all paths through the fabric are independent, that is, any connection can always be made between an idle IPC and an idle OPC without conflict.

SwElement (page 35). Switching Elements. A small-dimension crossbar switch, used as a building block for a multistage interconnection network.

TDM (page 3). Time Division Multiplexing. A data communications technique that assigns available bandwidth to users using predefined time slots. Connections take turns using the transmission channel. Time division multiplexing is the traditional method of sharing network resources.

TGen (page 59). Class in the ATM library generating ATM cells to be fed into the switching fabric.

Throughput (page 28). Performance parameter equal to the ratio of the number of returned cells to the simulation time.

Time Slot (page 6). The time to process one cell, equal to 425 cycles of the ATM library's switching system.

TManager (page 58). Class of the ATM library generating traffic with provided distributions.

VBR (page 11). Variable Bit Rate. An ATM service category that supports predictable data streams within bounds of average and peak traffic constraints. This service category is subdivided into both VBR Real-Time and VBR Non-Real-Time.

VC (page 15). Virtual Channel: A generic, single transmission path within the ATM standard.

VCI (page 10). Virtual Channel Identifier. The VCI is a 16-bit field found in the header of ATM cells that identifies a particular virtual channel within a virtual path.

VP (page 15). Virtual Path. A bundle of Virtual Channels (VCs).

VPI (page 10). Virtual Path Identifier. The VPI is a field found in the header of ATM cells that is used to group virtual channels into paths for routing purposes.

Wide sense non-blocking (page 17). A network, preventing conflicts through proper policy of allocating the connections.

Wire (page 36). The class representing the connector in the ATM library.

1 Introduction

The purpose of this thesis work is to evaluate the performance of switching fabrics for ATM network switches. To accomplish this, the structure of a general ATM network will be studied thoroughly. An ATM network consists of a group of ATM switches interconnected by a physical signal carrier. The switches create the circuits for the ATM protocol data units, called cells, from the source to the destination by assigning them proper paths through the switching fabric. Circuits may be established permanently or temporarily. The network provides a certain quality of service for a circuit that guarantees the maximum delay for a cell, the cell delay variation, and the maximum cell discard rate. Each router has a special processor to ensure the proper cell routing and the quality of service for each established circuit. The effective performance of the network is achieved by the cooperation of the controllers of each individual switch. An ATM network can be interfaced with other types of networks or with other ATM networks by means of gateways.

To build a switch, fitting certain requirements of a particular network, it is necessary to construct a model of this switch and test the model under conditions as close as possible to those expected in the real network. Simulation seems to be the only effective way to observe the effect of cooperation of the sequence of elements and to predict the behavior of an ATM switch under the selected conditions. Since there are several factors affecting the switching fabric behaviour and many parameters need to be evaluated, such an evaluation cannot be made analytically. On the other hand, simulation can only produce results for a certain set of initial conditions, but more accurately. The simulation tool may be implemented in software or in hardware. The most appropriate way of simulation is software discrete time simulation because hardware emulators are more expensive.

A certain amount of work has already been done on designing and simulating of ATM switches. Different approaches were involved in the simulation of switch components and particularly switching fabrics. Some examples are taken from sources [1], [2], [8], and [9]. Common methodologies used in the simulation are procedural and object-oriented. The simulation methods implemented in [8] and [9] served as the starting point in the design and simulation of different types of switching fabrics. In this work an object-oriented approach implemented in C++ was chosen.

A discrete time simulation software package called "ATM library" was created making it possible to construct models of all the components of an ATM switch and to organize communication among them, thus reproducing the ATM switch operation. The ATM library was carefully elaborated, adapting many useful features of "Arch library" [10]. Currently, the ATM library supports description of such elements of an ATM switch as switching elements, queues, connectors, and two auxiliary modules for interfacing with input sources and output locations. This thesis is not intended to check the functionality of the input and the output modules; therefore, the interfacing modules called sockets are used instead of the input and output modules to read the data from the input files and write the data to the output files. The functions of the input sockets were later expanded to accept traffic from dynamic traffic generators.

The created simulation tool models a synchronous network, where time is measured by clock cycles equal to the time, during which one bit propagates between two stages of a network. Consequently, only uniform baudrate data flow can be supplied to the inputs. One clock cycle may be assumed to be equal to the interval between the generation of two bits in a constant baudrate bit stream, that is baudrate^{-1} . So, the time to pull all the input data through the switching fabric can be determined as the number of clock cycles times the reciprocal of the baudrate, normalized to bits per some unit of time. Cut-through routing of ATM cells represented by streams of bits through the fabric makes a cell arrival event not instantaneous, thus extending one arrival to a certain number of clock cycles. Hence, the traffic arrival rate has an upper bound. The simulation results, in such a case, are only comparable with the results obtained analytically or by usual instantaneous arrival event simulation when very large number of simulation traces are generated.

The sections in this work are organized in the following order. In the second section the principles of the ATM standard are discussed. The third section gives the overview of the basic switching methods used in the ATM traffic management. The fourth and the fifth sections provide the simulation model description: of the basic part and the traffic generation part respectively. The sixth section offers the simulation results of the various network architectures, and the last seventh section summarizes the results of this thesis work.

2 The ATM Standard

2.1 Evolution of Digital Networks

The ideal future network is a single network that offers a wide range of services, can carry any type of data, and is able to share resources among different services. Digital networks more efficiently meet the criteria of a uniform network because they are less susceptible to noise and make data transfer more reliable. The description of the evolution is based on [6]. Evolution of digital networks began with the Integrated Digital Network (IDN) combining time division multiplexing (TDM) in the transmission lines and time division switching (TDS) in the routers. The advantage of the digital network IDN over the analogue network based on frequency division multiplexing (FDM) is that it can switch the user signal without changing its format. With the space division switching (SDS) of analogue channels the signal has to be demultiplexed, converted to the original baseband range and then multiplexed back into the transmission line. IDN uses separate channels for voice, video, and data as shown in Figure 2.1. The circuit switching portion supports only voice services, the data information is packet-switched; and a completely disjoint network is dedicated to video and specialized data services [6]. All the transmission channels in IDN provide narrow band (NB) services.

Figure 2.1. Segregated transport in a narrowband network [6].

The next step in the evolution was Integrated Services Digital Network (ISDN). This type of network provided access to all the information transfer services by means of integrated user-network interface (UNI). ISDN was adopted in 1980 by the International Consultative Committee on Telephone and Telegraphy (CCITT), which was reorganized to the Telecommunication Standardization Sector of the International Communication Union (ITU-T) in 1993. As shown in Figure 2.2, the common user-network interface is supplied for three types of networks: packet switching, circuit switching, and signaling. There is still the dedicated network for video. Although ISDN provides standard access to different services, it cannot offer high bandwidth services over 64 kbit/s.

Figure 2.2. Integrated access in a narrowband network [6].

Narrow band has been a barrier in the transmission of data, especially for real-time circuit switched traffic. After rapid development in optical technologies, channel bandwidth significantly increased. It was possible then to combine the services provided by the collection of networks into one single network of a higher capacity. So, the new type of network called Broadband Integrated Services Digital Network was created, and a study group of BISDN was initiated in the middle of 80's. The access to BISDN services is partially integrated in the sense that independent switching is required for each channel as shown in Figure 2.3. The multifunctional switches are connected to the broadband transmission system with the network-node interfaces (NNI). BISDN is what the Asynchronous Transfer Mode currently supports.

Figure 2.3. A Broadband Integrated Services Digital Network [6].

The step after the Broadband Integrated Services Digital Network is the broadband integrated transport, which is the network providing combined services for diverse bandwidth traffic. It has unique interfaces for all kinds of traffic including narrowband sound and broadband data and video. The structure of this network is shown in Figure 2.4.

Figure 2.4. Broadband integrated transport [6].

In order to achieve this level of service integration, it is necessary that all current long and medium-distance network services assume the ATM standard, thus creating a global digital network [6]. The cost of manufacturing and maintaining a uniform network is lower. Therefore, uniform networks such as broad band integrated transport should be used in the future, thus eliminating the need to produce other types of networks. Currently, BISDN with the ATM transmission rule is widely used in the communication services.

2.1.1 Implementation of Digital Networks

The best physical signal carrier for a BISDN network is optical fiber regulated by Synchronous Optical Network Protocol (SONET). The optical carrier is the unsurpassed transmission medium for long-range distances. Optical fiber has a significantly lower cell loss rate than any of the metallic conductors of electric signal. The transmission rate of optical carrier is up to several GBps. Signal attenuation is considerably lower in the optical fiber than that in an electrical signal carrier. SONET is a synchronous protocol, but it easily accommodates asynchronous transmission, which is more efficient than synchronous as will be shown later. Despite all the advantages of the optical signal transmission, switching of traffic is still to be performed by electrical equipment. Signal transmitted by the Synchronous Optical Network is switched electrically at the routers functioning according to the ATM protocol, in other words ATM switches. The optical signal is converted to electrical signal by the input modules of the switches, then routed through the switch fabric, and finally converted back to the optical signal to be further transmitted.

ATM is a connection-oriented way of communication. The connection setup phase precedes the data transmission, and the connection release phase takes place after the data are transmitted. It is required that packets -- the units of information called cells in the ATM standard proceed through the fabric in the original order during an ATM transmission session, and it is desired to drop as few cells as possible.

Synchronous modes of transmission do not completely utilize the transmission channel when some sources of traffic remain idle. In the synchronous transmission mode with time division multiplexing (TDM) each time slot corresponds to a certain information source as shown in Figure 2.5. If the source does not produce any traffic, the associated time slot is idle. In the asynchronous transfer mode the channel is idle only in those time slots when there is no data transmission from any sources as shown in Figure 2.6.

Figure 2.5 Multiplexing in synchronous transmission [6].

Figure 2.6. Multiplexing in asynchronous transmission [6].

Asynchronous transfer mode proved itself to be the best underlying implementation of digital networks as compared to the synchronous mode; therefore, the ATM protocol is considered in this work and described in detail in the later chapters.

2.2 The ATM Protocol

The reference model in Figure 2.7 describes the ATM protocol. The ATM protocol is collaboration of the three planes: the User Plane, the Control Plane, and the Management Plane. The function of the Management Plane is, first, to take control over the whole system by coordinating the functions of the Control Plane and the User Plane and, second, to coordinate functions of single layers of the User and Control Planes [6]. The subplane of Plane Management performs the first task, and the subplane of Layer Management accomplishes the second task.

Figure 2.7. ATM protocol reference model [6].

The function of the Control Plane is to set up, release, and monitor the virtual connections, and the User Plane is aimed to allow transfer of user information between nodes and control information between user and network.

2.2.1 Physical Layer

The Physical Layer is concerned with transmission of bits through the transmission medium. The Physical Layer is subdivided into Physical Medium and Transmission Convergence sublayers. The specific task of the Physical Medium sublayer is to provide the proper timing and coding of the signal. User-network interfaces as a part of the Physical Medium sublayer operate at the two rates specified by UTI-T. The first rate is 155.520 Mbit/s, and the second rate is 622.080 Mbit/s. The first transmission rate is applicable to both the electrical and optical interfaces and is symmetrical - the same transmission rate in the user-to-network and network-to-user directions. The second rate is applicable only to optical interface and may symmetrical or asymmetrical. In the case of the asymmetrical interface transmission in one direction is at 155.520 Mbit/s and in the other direction, 622.080 Mbit/s [6].

Additional functions of the Physical Medium and Transmission Convergence sublayer are cell decoupling and delineation. Cell decoupling is the insertion and extraction of empty cells when no cells are supplied by the sources, and the cell delineation is the identification of cell boundaries. This sublayer also performs the generation and verification of the Header Error Control Field (HEC) in each cell.

2.2.2 ATM Layer

The ATM layer is concerned about supporting the virtual connections between two or more network nodes or end-users. The ATM layer forwards the cells along the set up virtual channels contained in virtual paths. As it is shown in Figure 2.8, a virtual path is a bundle of virtual channels sharing the same identifier value [6]. A group of virtual paths is combined into one transmission link, and the transmission link represents a real physical signal carrier. Availability of an already established virtual path simplifies the setting up of a new virtual

channel, since there is already an available way for the control signal. Normally, virtual paths are defined in advance for the whole net, and the virtual channels within the same path represent information routes of the different processes running within the same host. One destination host can support a certain number of virtual paths, which is usually no greater than 256 because a virtual path identifier consists of 8 bits as described later.

Figure 2.8. ATM Transmission link [12].

The information transmitted according to ATM protocol is encapsulated in 53-byte cells with a 5-byte header and 48 bytes of user information. The structure of a cell is shown in Figure 2.9. The fields of the cell header are the following.

GFC/VPI		VPI	
VPI		VCI	
VCI			
VCI		PTI	CLP
Header Error Check (HEC)			
Payload			

Figure 2.9. Structure of an ATM cell [12].

GFC/VPI -- Generic Flow Control as a part of Virtual Path Identifier. Generic Flow Control field is defined only in network control cells. It is an extension of the virtual path identifier. The virtual path identifier is used in the virtual path routing of the cells. In the user data cells this field is held for future use.

VPI -- This field represent the virtual path, according to which cells are routed.

VCI -- Virtual Path Identifier. This field indicates the specific channel in a routing path.

PTI -- Payload Type. This field indicates the type of transmitted information.

CLP -- Cell Loss Priority bit. This bit is set when the priority of traffic is lower.

HEC -- Header Error Control. Single-error correction code based on 8-bit polynomial, checking the validity of the remaining 32 bits.

Payload -- user data.

The ATM Layer also accomplishes such functions as transmission and switching, congestion control and buffer management, cell header generation and removal at the source and the destination, and cell address translation. The ATM layer also ensures the sequential delivery and maintains the traffic parameters. The traffic parameters include the cell rate, cell delay, cell delay variation, and the cell error rate. The traffic parameters are described in more detail in [14].

2.2.3 ATM Adaptation Layer

The purpose of the ATM Adaptation Layer is to break the messages from the application layers into ATM cells. The ATM Adaptation Layer is subdivided into the Segmentation and Reassembly sublayer (SAR) and the Convergence sublayer (CS). The task of the SAR sublayer is to break the variable size user information into fixed-size ATM cells at the transmitter and reassemble the information at the receiver. The Convergence sublayer is needed to map the specific user requirements onto the ATM transport network. The Convergence sublayer consists of the common and the application specific parts. The functions of the subparts are message framing and error detection [5]. There are four classes of traffic that the ATM Adaptation Layer deals with named A, B, C, and D. They are provided in Figure 2.10.

	Class A	Class B	Class C	Class D
Time Synch	Required		Not Required	
Bit Rate	Constant	Variable		
Connection Mode	Connection oriented			Connect ionless
AAL	AAL 1	AAL 2	AAL 3/4/5	AAL 3/4/5
Examples	Circuit emulation	Compressed Video	Frame Relay	SMDS

Figure 2.10. Original service classes of the ATM Adaptation Layer [12].

The service classes are determined by the requirements of each particular type of information transmitted. One kind of traffic may require constant bit rate (CBR), some other type of traffic may use streams of information with varying bit rates. A session may be connection-oriented or connectionless. There are four divisions of the AAL. They are AAL1, AAL2, AAL3/4, and AAL5, each of which corresponds to the particular type of transmitted information. The intended use of the divisions of the AAL is described in more detail below.

AAL1. This division of the AAL is intended to work with the constant bit rate and serves the traffic class A including video, sound, and circuit emulation. This is a connection oriented type of service.

AAL2. This division of AAL services class B traffic, which is compressed realtime video such as video conferencing or interactive multimedia. The bit rate is variable (VBR). Type of session is connection oriented.

AAL3/4. This protocol provides services for class C traffic that is also connection oriented with variable bit rate. This type of protocol services multimedia, e-mail, and other types of data transfer.

AAL5. This AAL part supports connectionless services. It is used for communication in LAN interconnecting applications with variable bit rate. Class D traffic is serviced by this protocol.

The network may or may not be able to provide the services that a particular connection requires. When a connection is established the user and the network agree on the type of services provided.

This section gave an overview of the ATM standard. The topics considered were the evolution of digital networks, requirements imposed on the transmission medium and some details of the ATM protocol. The next section describes the switching fabric structure and the principles of switching in detail and gives the theoretical base for the implementation of a real simulator.

3 Switching Fabrics and Techniques Overview

Modern communication systems are built from transmission links interconnected with routers. Routers rearrange the connections among certain links according to a packet switching or a circuit switching regulation. Particularly, ATM standard accepts switching of cells preserving their order within the router. The process of switching is rather complex and became the subject of switching theory that has been developing for about forty years. The first step of the switching theory was to describe three-stage interconnection networks, either non-blocking or rearrangeable, which were only useful for the telephone service. This classical theory is no longer effective to describe switching environments when hundreds of millions of packets per second must be switched [6]. A new theory highlighting various networking architectures and methods has emerged. The general switch architecture adopted by the new theory is shown in Figure 3.1. An ATM switch consists of a set of input port (IPC) and output port (OPC) controllers connected with an interconnection network (IN).

Figure 3.1 General ATM switch architecture.

The functions of the controllers include rate matching between the channels fixed to the inputs and outputs of the switch and the fabric, attaching or detaching the routing tags, aligning cells for switching or transmission, modifying the VPI/VCI fields described in section 2, and queuing. Port controllers sometimes may be the input and output modules interfacing the ATM protocol with SONET. The function of the interconnection networks

is to forward each packet from the input port it entered to the destination output port according to the routing information. In order to establish a path within a switch, a sequence of commutations should be performed in the switching elements forming the interconnection network. A switching element is a box having a certain number of inputs and outputs. Any input may be connected to any output when setting up a path. Unless high capacity transmission links are used, an output can have only one input connected to it. Figure 3.2 gives the possible connection patterns among inputs and outputs within a 2x2 switching element. The examples are taken from [6]. The functional scheme of a switching

Figure 3.2. Examples of connection in a switch-box: (a) – straight, (b) – cross, (c) – upper broadcast, (d) – lower broadcast.

Figure 3.3. Functional scheme of a switching element.

element is shown in Figure 3.3. To provide dynamic connecting of inputs to the corresponding outputs, an arbitration mechanism must be involved because the needed output may be busy. The decision logic part accomplishes the arbitration procedure. Arbitration scheme may be more complex in switching elements with number of inputs more than two. A connection may be delayed, as shown in Fig. 3.3, in the case if another input engages the output line.

The intention of this work is to study the performance of the interconnection networks; therefore their architectures are described in more detail in the following chapters.

3.1 Switching

In order to forward an ATM cell from an input port to the corresponding output port the network management has to make a decision based on the information contained in the ATM cell header. The routing may be centralized or local -- on the self-routing basis. The switching decision may involve reassigning the VPI/VCI pair for each cell in the case of virtual channel (VC) switching or reassigning only VPI in the case of virtual path (VP) switching. The process of switching in an ATM network is shown in Figure 3.4. Here each cell entering any of the switches is reassigned a new VPI/VCI label marked with a capital letter and forwarded to the appropriate output link indicated by a small letter. In the case of VP switching each cell follows the predetermined path. In the case of VC switching a cell may change the virtual path within the fabric. The setting of paths and channels in a switch for the case of virtual path switching is shown in Figure 3.5. One transmission link is attached to each input port of a switch, and the transmission paths of each input port are distributed among the output ports.

Dashed lines depict virtual channels. The VPI and VCI values in the cell header attribute an ATM cell to a particular virtual path and virtual channel. Since VPI has 8 bits, the choice of destination for cells entering each input port may be made of 256 output ports. Normally, VP switching is used rather than VC switching, because the virtual channel identifiers are basically needed to identify the process at the destination host, to which the cell is addressed.

Figure 3.4. An ATM switching example [6].

Figure 3.5. Setting of virtual paths and virtual channels in an ATM switch.

One switching element may be treated as a 2x2 crossbar switch because each input port can be directly connected to each output port without internal blocking. Figure 3.5 would illustrate to principle of operation of such crossbar switch.

3.2 Switch architectures

There are different types of switch architectures interconnecting inputs and outputs of a network. The architectures may be blocking or non-blocking. Blocking networks may deny requests to set up new connections due to conflicts with already established connections. Non-blocking architectures are further divided into strict sense non-blocking -

- always allowing establishing a new connection, wide sense non-blocking -- preventing conflicts through proper policy of allocating the connections, and rearrangeable non-blocking -- giving the way for a new connection by rearranging the already existing paths.

Figure 3.6. Topologically equivalent non-isomorphic networks.

Three other important properties of the networks are isomorphism, topological equivalence, and functional equivalence. Topological equivalence is the similarity of underlying graphs. Isomorphism is the topological equivalence with the correspondence of all the labels. Functional equivalence is the correspondence of the permutations performed by two networks. Functional equivalence does not imply isomorphism because the implementation of two "black boxes" functioning equally may be different. Isomorphism leads to functional equivalence when the permutations are the same. Figure 3.6 shows an example of topologically equivalent but not isomorphic networks, and Figure 3.7 gives an example of three isomorphic networks.

Figure 3.7. Isomorphic networks.

Networks A and B in Figure 3.6 are not isomorphic despite that they have the same graph because mapping of label preserving isomorphism between them cannot be found. The graph of these two networks is shown in the same figure. The circles in the graph symbolize the switch boxes, and the lines show which boxes are connected. Each switch box has outputs "0" and "1". In network A output "0" of the upper box H is dangling and output "1" is connected to the box in the next stage. In network B output "0" of the corresponding box either X or Y is connected, but output "1" is unattached; therefore, the networks are not isomorphic. In Figure 3.7 networks A' and A'' are isomorphic to B because all the corresponding outputs of the matching boxes are connected equivalently between the stages. In isomorphic networks the paths between any two corresponding terminals, as specified by the labels, go through the corresponding boxes and pass through the same sequence of output connections in the boxes labeled "0" and "1". However, if the corresponding inputs and outputs of the isomorphic networks are not grouped in the same order, the networks are not functionally equivalent. So, the networks in Figure 3.7 are not functionally equivalent. To achieve functional equivalency, the matching inputs and outputs of the isomorphic networks must be regrouped equivalently as shown in Figure 3.8.

Figure 3.8 Functionally equivalent isomorphic networks.

Networks A and B shown in this picture are functionally equivalent because the matching terminals on both sides of the nets are ordered equally.

A common type of interconnection network forming the switch architecture is the banyan network. The Banyan network has a mixture of topological variations that are

formed by interconnecting adjacent stages of switching elements according to the interconnection patterns. Some interconnection patterns described in [6] are shown in Figure 3.9. An interconnection pattern is a correspondence of the connection made on the input side of a connector to that made on the output side of it. The interconnection patterns are defined by so-called shuffle functions or bit permutation equations that rearrange the bits making up a number. The functions allow finding the output connection according to the input connection using the pin number represented in the binary form. In the patterns of Figure 3.9, the pins are numbered 0 through 15; to represent this count, four-bit numbers are needed. Some shuffle functions may also be described analytically. To demonstrate how shuffling works, below given the definitions of basic shuffle functions [6]:

Figure 3.9. Interconnection patterns.

- $\sigma_h(a_{n-1} \dots a_0) = a_{n-1} \dots a_{h+1} a_{h-1} \dots a_0 a_h \dots a_{h-1} \dots a_0$ ($0 \leq h \leq n-1$) -- h-shuffle;
- $\sigma_h^{-1}(a_{n-1} \dots a_0) = a_{n-1} \dots a_{h+1} a_0 a_h \dots a_1$ ($0 \leq h \leq n-1$) -- h-unshuffle;
- $\beta_h(a_{n-1} \dots a_0) = a_{n-1} \dots a_{h+1} a_0 a_{h-1} \dots a_1 a_h$ ($0 \leq h \leq n-1$) -- butterfly;
- $j(a_{n-1} \dots a_0) = a_{n-1} \dots a_0$ -- identity;
- $\delta(a_{n-1} \dots a_0) = a_1 a_2 \dots a_{n-1} a_0$; -- delta;
- $\rho(a_{n-1} \dots a_0) = a_0 a_1 \dots a_{n-2} a_{n-1}$ -- ro.

H-shuffle and h-unshuffle permutations with $h = n - 1$, become perfect shuffle and perfect unshuffle. The perfect shuffle and unshuffle permutations can be expressed analytically as:

- $\sigma(i) = (2i + \text{floor}(2i/N)) \bmod N$ – perfect shuffle;
- $\sigma^{-1}(i) = \text{floor}(i/2) + (i \bmod 2)N/2$ – perfect unshuffle.

A series of banyan networks is presented in Figure 3.10. The SW-banyan network (a) is interconnected with butterfly permutation β_{n-1} , and the baseline network (d) is built with perfect unshuffle σ_{n-1}^{-1} ; n is the number of current stage changing from 1 to $M = \log_2 N$, where N is the number of outputs. Stage 1 interfaces the inlets and stage M interfaces the outlets of a network. These two networks are assembled recursively in $M-1$ steps. The first stage of a Σ_N SW-banyan network is connected to two other SW-banyan networks $\Sigma_{N/2}$ of half size. Analogously, the first stage of a Φ_N baseline network is connected to two other half-sized baseline networks $\Phi_{N/2}$. N goes down from 16 to 2. The omega network shown in Figure 3.7 (b) is built with the perfect shuffle σ between each pair of stages. And finally, 4_cube is constructed with butterfly β_{M-n+1} between stages 2 through 4 and with perfect shuffle σ between the first stage and the inlets.

(a) SW-banyan;

(b) omega;

(c) 4 cube;

(d) baseline;

Figure 3.10. A series of banyan switch architectures.

Two important properties of banyan networks are the buddy property and the constrained reachability property. The buddy property is formulated as follows. If switching element j_i (SE j_i) at stage i is connected to SEs l_{i+1} and m_{i+1} , then these two SEs are connected also the same SE k_i in stage i [6]. Or simply, switching elements in adjacent stages are connected in couples to each other. The other constrained reachability property is defined in this way. The 2^k SEs reached at stage $i + k$ by an SE at stage i are also reached by exactly $2^k - 1$ other SEs at stage i [6]. These two properties are demonstrated in Figure 3.11. Since the constrained reachability property applies for the architectures shown in Figure 3.10, all of them are isomorphic to each other. One network can be obtained from another one by moving the switch boxes around. However, to achieve functional equivalency, permutations must be the same in two given networks.

Figure 3.11. Buddy and constraint reachability properties.

All the considered here architectures of banyan type are subject to internal conflicts that substantially reduce their performance. An example of conflict is shown in Figure 3.12.

Figure 3.12 An example of conflict in a banyan network.

Two cells forwarded to different output ports 1010 and 1000 are in conflict in the first stage. If cells entered the fabric from a different pair of inputs, this internal conflict would be eliminated.

There are lots of alternatives to banyan networks that reduce the contention in the fabric. One group of alternatives is completely

Figure 3.13. A part of generic crossbar switch.

interconnected networks, such as knockout or crossbar tree with each input connected to each output. Another group is fully connected networks where every element is connected to every element in the contiguous stage. The network with the largest number of switch boxes is the crossbar switch. It eliminates all internal conflicts. The layout of an indefinite size crossbar switch is shown in Figure 3.13. Other techniques of conflict reduction are network replication, link dilation, and extended general shuffle. In this work the methods of rearranging and sorting reducing conflicts as well are investigated. The description of those methods is provided in the following two chapters.

3.3 Rearrangeable networks

Usual multistage interconnection networks provide only a single path between each input port and each output port. In such networks, apart from blocking at the final stage, an

internal conflict may happen when two cells directed to adjacent output ports enter the fabric. There is a special sort of networks called rearrangeable networks, where an incoming cell may follow not necessarily a predetermined way towards the output port, but some alternative way. The availability of alternate paths makes it possible to eliminate the internal conflicts in the fabric to a certain degree. Another definition of a rearrangeable network is the network allowing rearranging already set up paths within the fabric when a new cell arriving at input is in conflict with some other cells already on their way through established paths. The first approach seems to be more reasonable because rearranging already set up paths takes additional efforts. However, just occupying an available path may be not as effective as rearranging. However, algorithm of the first method is simpler and may be easily designed and implemented.

Benes rearrangeable network as an instance of a rearrangeable network is certainly supposed to improve the performance of a banyan network at the cost of additional m stages. Performance of the Benes rearrangeable network shown in Figure 3.14 is explored in this work. The network was constructed by the horizontal extension technique discussed in [6]. This technique is referred to as extended banyan network (EBN). According to the technique, m additional stages can be incorporated into the available banyan network architecture of n stages.

Figure 3.14. Benes rearrangeable network.

Connection pattern of the new stages is obtained as the mirror image of the permutations in the last m stages of the original banyan network. Number m cannot exceed $n-1$. Presence of m additional stages increases the number of paths from any input to any output from 1 to 2^m .

Rearrangeable networks eliminate internal conflicts by providing alternative paths for the cells. But even in this case the number of paths may not be sufficient enough to eliminate all the internal conflicts. Another approach of conflict elimination is sorting. The sorting techniques discussed in the next chapter may be even more effective and sometimes allow complete eliminating of internal conflicts, although at even higher cost.

3.4 Sorting networks

The problem in banyan networks is that internal conflicts appear when the traffic on the inputs is not properly arranged. For a particular traffic correlation, a unique type of network may be needed to avoid conflicts. Sorting rearranges traffic on the inputs so that a fixed type of a banyan network can handle it. Sorting stage S_8 is placed before a switching stage Γ_8 in a sorting-routing network as shown in Figure 3.15. As it can be seen in the picture, there are no conflicts in the switching stage.

Figure 3.15. Combined sorting-routing network. The "gray element" forwards the cell with the minimum address to the output with ID of 0, and the "black element" sends the cell with the minimum output address to the output with the ID of 1.

An elementary sorter arranges two ordered sequences $a_0, a_1, \dots, a_{n/2-1}$ and $b_0, b_1, \dots, b_{n/2-1}$ into one ordered sequence $c_0, c_1, \dots, c_{n-1}$. A cascade of elementary sorters forming a sorting stage allows arranging a group of unsorted elements into an ordered sequence. In the networks, packets arranged with a sorter are presented in an ordered sequence at switch input and therefore cause fewer conflicts

Figure 3.16. Sorting stage for a 16-input banyan network [8].

There are two basic sorting algorithms: odd-even merge sorting and bitonic merge sorting. In the odd-even merge sorting rule each sorted element can be compared different number of times with the other elements. In the bitonic merge sorting algorithm each element is compared with the other elements the same number of times. The number of sorting elements is the same in each stage in a bitonic merge sorter. In an odd-even merge sorter the number of sorting elements can differ from stage to stage. There are fewer sorting elements in an odd-even merge sorter than in a bitonic merge sorter.

Figure 3.16 gives a binotic merge sorting stage for sixteen elements with an example of sorting 4 elements. Such stage can be interfaced with a banyan network to form a sorting-routing switch. The simulation results are provided for the sorting and switching network in section 6.

3.5 Queuing methods

Queuing is an inevitable part of any switch design approach. Queues are needed to accumulate the cells on the input side when internal conflicts prevent their entering into the fabric and on the output side when the arrival rate of cells to the outputs exceeds the service rate of output modules. The common queuing techniques defined by [14] are input queuing, output queuing, internal queuing, recirculating buffers, and buffer sharing.

Input queuing is the accumulation of cells on the input side when for some reasons they are not allowed to enter the fabric of the switch. Input queuing takes place when the cells that are in conflict are stalled in the fabric by the backpressure mechanism. Input queuing suffers from a problem called head of line blocking. It prevents entering of the other cells into the fabric when the first cell in the path has been blocked. Internal queuing or output queuing without internal conflicts can substitute input queuing.

Output queuing is the buffering of cells at the output stage of a switch. Output queuing eliminates the head of line blocking, but its implementation is rather complex. In order for an output buffer to become effective, it should receive multiple cells per a time slot, otherwise no cells will be accumulated. To ensure this, the switch fabric can operate at some factor times the port speed or there should be multiple buffers at the output ports, for example, a buffer for each class of traffic.

In ordinary networks at most one cell is delivered to an output port in a time slot; therefore, output buffering is only practical when the fabric works with a speedup. To reduce the undesirable effect of head of line blocking, there is another option -- internal buffering. In this case buffers are placed inside the switching elements. If a conflict occurs close to the output stage, new cells can enter the fabric, because absorbing the cells in conflict by the buffers clears the space. Internal buffering also reduces the number of internal conflicts proportionally to the number of cells having entered the fabric. The overall throughput usually increases even though the total number of conflict sometimes also increases. The problem with internal queuing is random delays of cells within the fabric, which causes undesirable cell delay variation.

Recirculating buffers route the cells forwarded to the same output port back to the input stage, thus reducing the contention at the output ports. This method has two

restrictions. First, the network has to be large enough to handle the recirculating cells, and the second is that an advanced mechanism maintaining the cell order must be employed.

Finally, buffer sharing within a stage of queues allows using the same portion of memory for storing the cells belonging to different logical buffers. In a shared queue, when a large burst arrives at one logical buffer, the memory space belonging to other logical buffers in the queue can be granted to the busiest one. The cell loss probability considerably reduces with this method, but complexity increases.

3.6 Buffer management schemes

The switch management to handle the traffic of different Quality of Service (QoS) classes must perform special buffer monitoring functions. The buffer management functions include separate output buffering for different quality of service traffic, discarding and scheduling policies, congestion indication, and accounting.

The traffic classes are determined by the allowable cell delay variation and cell loss priorities. The Cell Loss Priority (CLP) bit is set in the ATM cell header when the priority of the cell is relatively low. Cells with set CLP bit are discarded prior to those with CLP bit cleared when buffers overflow. The CLP bit is either set by the user, or by the Switch Management when the user exceeds the traffic agreed upon [14]. Traffic within a certain virtual path that corresponds to one circuit may have several degrees of delay priority in each virtual channel. Within the same virtual channel, however, the delay priority does not vary. Only cell loss priority can be different within a virtual channel. The CLP bit determines the loss priority of a cell, but there is no special field in the cell header to show the delay priority. Usually the information on delay priorities is stored in the VPI/VCI translation table within the switch. Sometimes this information is embedded into the routing tag of each cell.

Separate buffering is one of the buffer management functions. It allows handling different delay priority streams of traffic. Since traffic following the same virtual channel has the same delay priority, maintaining several FIFO buffers for different traffic classes at each output port complies with the ATM standard of no reordering. Multiple buffers are maintained at the output ports because it was discovered that output queuing is the best queuing approach as far as the head of line blocking is concerned.

There are two common cell-discarding schemes. One of them is push-out and the other is partial buffer sharing. In the push-out scheme a cell with CLP field equal to 1 can no longer enter a full buffer, whereas a cell with CLP field set to 0 can replace a cell with CLP of 1 in a full buffer. This discarding policy is the most acceptable in terms of efficiency and complexity. The other scheme called partial buffer sharing makes use of two thresholds in a buffer, one of them is for cells with CLP of 0, the other one is for cells with CLP of 1. The cells with CLP of 1 are discarded before the cells with CLP of 0. Partial buffer sharing is more easily implemented, but it is less efficient because it starts discarding CLP 1 cells even if there is no potential buffer overflow. In a cell-discarding scheme the cells are buffered and, if the buffer overflows, they are discarded. The alternative to the cell discarding is backpressure. The backpressure mechanism does not allow entering cells to a full buffer, it rather sends signals to the backward stages to stall the traffic. There is one advanced way of buffer managing dealing with cell flow regulation called delayed push-out. This method is described in detail in [18]. The method employs both the buffering component and the backpressure component. The buffer size is predetermined as well as for the pure buffering schemes. But the cells about to arrive at a full buffer are not discarded but rather stalled at the previous stage. According to this strategy, memory of different stage buffers is shared. This helps to significantly decrease the cell loss probability, but sometimes head of line blocking problem emerges.

The cells scheduling policy of the buffer management regulates the exiting of cells from the buffer rather than their entering like the cell discarding policy. This work is not focused on the cell scheduling policies; therefore, only a few of them are mentioned. Common cell scheduling policies are the static priorities and the deadline scheduling. In the static priorities the lower priority class traffic is output only when there is no higher priority traffic in a buffer. In the deadline scheduling each cell has a target departure time from the queue based on its quality of service requirements.

The buffer management's congestion indication mechanism performs the signaling functions in an ATM switch. Signaling prompts immediate adjusting of the discarding and scheduling policies in the switch in the case of congestion. The congestion indication mechanism collects the queue statistics by examining such fields in the internal routing tags as timestamps and housekeeping and alerts the switch management if congestion is detected

within the fabric. Queue statistics shows whether congestion is increasing or receding, and whether it is focused or widespread, this helps to make right policy change decision [14].

The congestion indication mechanism cooperates with the accounting mechanism by presenting it the information about the performance, congestion, discarded cells, and buffer utilization. The accounting mechanism records the above mentioned data and utilizes them to make improvements in the switch buffering policy.

This section introduced the theoretical principles used in the construction of ATM switches. The various routing techniques were considered for optimized routing of ATM cells as well as buffer management schemes and cell scheduling strategies. The next section describes how a real simulation model was implemented utilizing the valuable theoretical information presented in this section.

4 Simulation Approach

Switching is not part of the ATM standard; therefore, many different techniques are employed in building ATM switches. In order to evaluate the performance of each particular method, either an analytical model or a simulation model must be constructed. Simulation model seems to be a better way to estimate the performance because it more specifically describes the events that happen in the real system, whereas in an analytical model some factors may be ignored and the whole model may only serve as an approximate scheme of the real system. At the same time, simulation cannot guarantee the correctness of the system in all the cases because it is impossible to run an exhaustive test for a large network. Therefore, the formal verification is currently of much interest. Formal verification of such a complex construction as an ATM switch, however, is not an easy task because the existing techniques either require a profound understanding of the underlying mathematical model or are inept for handling large systems.

4.1 Existing performance evaluation methods

An example of analytical network modeling is described in [15], where the throughput of a network is evaluated by the probability mass functions of channel loads. The analytical method is defined as follows. Each channel has a load probability. The probability that output data are present on some output is calculated repetitively by recurrent formulae. The time to pass all the traffic through the fabric is determined by considering the probabilities of a conflict occurring at each stage of the network. Time spent on contention during a conflict is added to the total time; and the throughput is inversely proportional to the overall time. This work is an attempt to develop an all-purpose analytical model adaptable to any kind of network. Nevertheless, the model is still lacking the ability to simulate buffered networks. As simulation results show, buffered networks achieve higher throughput than those without buffers do. The analytical model is less scalable than simulation models because every iteration requires a large number of calculations that do not scale linearly with the net size. However, the analytical method converges fast and therefore is more effective for networks of smaller size.

The work on analytical modeling [15] gives the comparison of the results obtained by the simulation and the ones predicted analytically. In Figure 4.1 the throughput graph is drawn as an example of such comparison. As it is seen in the picture, the analytical evaluation curve of an unbuffered network is identical to the one produced by the simulation. In the case of buffered networks with buffer sizes 2 and 4, the simulation gives other curves with higher throughput. The simulation model used in this comparison is based on the generation of messages according to the probability mass function of channel loads. Analytical methods may be effective and cheap for small networks, but they still do not allow evaluating many different parameters at the same time because the number of calculations considerably increases with each new parameter introduced. Such parameters may include cell delay, number of conflicts and so on. For the simultaneous evaluation of a large set of parameters a simulation model is preferable.

Figure 4.1 Throughput estimated analytically and with a simulation model [15].

The common way of simulating networks, as many sources suggest, is to make use of a synchronous network, in which at each clock cycle messages move from stage i to stage

$i+1$. This method was proposed in [2]. Networks are usually constructed of "software chips" and the organization of the simulation model is object oriented. Possible tools of implementation of such systems are C++ and VHDL languages. In the simulation models known so far, messages are treated as the elementary units of information that arrive instantaneously in the system. This practice mimics store-and-forward routing. In the simulation model of this thesis work, bits represent units of information, and the model makes use of wormhole or cut-through routing. The messages are not instantaneous and it takes 425 cycles for the elementary unit of information, i.e. one cell to arrive in the system. Definition of a clock cycle is further provided in the functional description of the classes. As a result, the arrival rate is bounded by some value. In the simulation, an object-oriented approach was chosen. Objects represent all the basic components of ATM switches; and the objects are described in C++ language. The simulation model is synchronous as well. This methodology is based on the ATM library, which is to be discussed in the next chapter.

An attempt to formally verify the Fairisle ATM switch is described in source [3]. This formal verification method is based on the MDG's -- multiway decision graphs. The multiway decision graphs allow representing large logical expressions in a convenient compact form. The MDG's method was successfully employed in the formal verification of the ATM switch on the behavioural, abstract RTL (register transfer level) and the gate level. In this work a step was made also to illustrate the proper functioning of the system, it is described at the end of this section.

All the considered switch fabric evaluation methods have their merits and drawbacks as far as efficiency and precision are concerned. The main reason why an object-oriented simulation model was chosen in this work is that it is convenient for the simulation of any particular switch architecture. It allows constructing switches of any possible configurations. It is also possible to evaluate a large set of parameters in one simulation step.

4.2 ATM library

This new discrete time simulation tool used for the simulation of ATM switch fabrics uses synchronous network. The approach was borrowed from the "ARCH library" [10] -- software library used for the simulation of computer architectures. The new software package was named "ATM library". ATM library allows evaluating the time required to pull

a certain amount of traffic through an ATM switch. The time is measured by clock cycles that the simulation requires. One clock cycle is equal to the time, during which one bit propagates between any two stages of the switch. It takes 425 cycles for one ATM cell to arrive in the system; therefore, this time is considered one time slot. It is also possible to obtain the average and the maximum queue sizes in all the objects. Additionally, the absolute maximum queue size and the overall average queue size considering all the queues and only active queues are determined for all the objects of the same type: Queue's and SwElement's separately. These two classes are further defined in this section as the classes representing a queue and a switching element. Special functions make it possible to calculate the number of conflicts in all the switch boxes. Total numbers of lost cells in all objects and the lost probability are calculated as well. For shared queues, the size of each logical queue is estimated as well as the overall queue size in all the logical queues forming the shared queue. For each output port, the minimum, average, and maximum cell delay is determined. For all output ports, the absolute maximum and minimum delays and the overall average delays are calculated. After the simulation is complete, the total number of lost cells, the total number of conflicts and the overall loss probability are recorded in the output file. A programme compiled with the ATM library can use as input data either artificial traces or traffic produced by traffic generators, which will be discussed in section 5. The ATM library utilizes many valuable features of the ARCH library. UML diagramme of the ATM library without the traffic generation part is given in Figure 4.2. The traffic generation part will be discussed in section 5.

4.2.1 Object organization

As it was mentioned earlier, the ATM library was derived from the Arch library [10] and has many common elements. The difference between them is that the object organization in the ATM library is more appropriate for the simulation of switching elements than computer hardware, and there are additional classes specially created for an ATM switch simulation. The ATM library has a class called BaseClass from which many other classes inherit. BaseClass keeps the count of all created objects and contains some common functions of the polyfunctional classes. When the objects of the simulation are destroyed, BaseClass invokes the *death()* functions of Clock and Manager to display the final

results of the simulation. Objects of the other classes that do not inherit from BaseClass are destroyed with the help of their destructors invoked by the host objects. Clock and Manager are the two static classes carrying out some controlling functions; no objects of these classes need to be instantiated. All of the classes that inherit from class ClockedObject have functions *void phase1()*, *void phase2()*, and *void phase3()* invoked by the Clock, when the main programme executes function *tick()*. Classes InSocket, OutSocket, SwElement, and Queue represent the building blocks of an ATM switch. The four mentioned above classes have many common functions; therefore, class StorageObject is used to contain all the functions shared by them. StorageObject itself is derived from class ClockedObject; consequently, all of its subclasses are so-called active classes, because the Clock drives them, and they actively participate in the

Figure4.2. ATM library UML diagramme.

propulsion of data bits through the interconnection network. The other major class used in the simulation is Wire. Objects of class Wire are used to interconnect the basic elements of a network switch.

Figure 4.3. A chain of ATM library objects.

Figure 4.3 gives an example of a simple chain built of the ATM library objects. This chain was constructed "manually" and source file "element.C" executing this chain is provided in appendix A. The function of the chain is to take ATM cells from the source file and forward them bit by bit to the destination file. 53 integer numbers, whose values stand for the contents of the 53 bytes of an ATM cell, represent ATM cells. One destination file with eight ATM cells is included in appendix B. The output file is named "output1.cel", which means that the object of class OutSocket numbered 1 writes output data to this file. The cells in the file contain virtual path identifiers. The standard virtual path identifiers (VPI) occupy bits 4 through 11 of a cell. In this model the destination file is determined as the remainder of the division of VPI by the number of destinations ($VPI \bmod num_destinations$). For example, if a cell has the third hexadecimal character '1', it will definitely go to output file output1.cel, provided that the number of destinations is no greater than 16. For convenience, each cell has the embedded source file number and the sequence number; those numbers are not standard, of course.

There are two ways to build and to simulate a switch using the ATM library. One of them is to declare all the objects and describe all the interconnections manually, and then write a while-loop that will reiterate the simulation cycles as in the example of programme "element.C". For the manual construction the following prototype files must be included: <InSocket.H>, <OutSocket.H>, <SwElement.H>, <Clock.H>, and <Manager.H>. <Queue.H> is optional and is needed only if any queues are used in the switch design. The other way is to use programme *nm* or *nm1* that are intended to facilitate the construction of excessively large networks, whose manual construction is impractical. These two programmes have one common driver "run.C" with the function *main()*. Programme *run*

uses the switch constructor contained in the file "Switch.C" and programme *run1* uses the constructor in "Switch1.C". For the automatic construction and execution a setup file needs to be produced and put into the directory "INIT" for *run* and into the directory "INIT1" for *run1*. The two directories are located at the same level as the object files *run* and *run1*. A setup file consists of a group of numbers, which are scanned by the constructor of class Switch that assembles the ATM switch for the simulation. Class Switch has two public functions: the constructor and the function *work()* used by the programs *run* or *run1* to execute the switch. The constructor in the "Switch.C" is more flexible than that in the "Switch1.C" and allows constructing any network configuration, not necessarily symmetrical and with any random interconnection of elements possibly linking not only adjacent stages as shown in the example in Figure 4.4. Greater flexibility is achieved by a larger size of the setup file.

Figure 4.4. Non-symmetrical switch configuration with complex interstage connections.

Constructor in "Switch1.C" is less flexible and can only be applied to symmetrical networks with adjacent stages interconnected and crossbar switches. The second constructor is preferred when simulating very large networks since it allows interconnecting adjacent stages very efficiently with the interconnection pattern provided. Shuffling functions to form interconnection patterns are implemented in "Shuffle.C" linked to the constructor of the *run1* programme. Interconnection patterns are discussed in section 3.

Shown in Figure 4.5 is an example of how the baseline switch with 4 rows and 3 stages is constructed using the ATM library elements with the help of the "run.C" driver. In both Figures 4.3 and 4.5 all the elements have their corresponding names starting with some letters and ending with one or two numbers. Those symbols appear when tracing the running programme and in the file "OUTPUTS/postscript.txt" containing the results of a simulation. Debugging of such a huge library is a complex task, and the symbols associated with each element help to observe in the tracing mode what value at what clock cycle propagates from one element to another. Here the switching elements are marked with letters "SE", strings "IS" or "SI" symbolize input sockets and letters "OS" or "SO" symbolize output sockets. Letter "Q" represents queues and wires are indicated by "WR". Labeling of SwElements, Queues and Wires can be arbitrary; these numbers are needed only for the user during a debugging and to observe the final results. Indexing of pointers to these objects is a separate issue from labeling and is very important because correct indexing helps to arrange the objects properly in the network. It is desirable for the purpose of convenience that the labels correspond to indices.

Figure 4.5. Baseline network.

If there are more than one InSocket or more than one OutSocket, labels for both of these types of objects must correspond to the pointer indices the making of and should go in order starting from 0, because those numbers are used by the Manager for the routing table. The source files of the driver programme "run.C", the descriptors of class Switch: "Switch.C", "Switch1.C", and "Switch.H", and the shuffling function implementation "Shuffle.C" and "Shuffle.H" are given in appendix A. To create the ATM switch shown in Figure 4.5, constructor of class Switch used the setup file called `q_baseline3x4.des`, which is also provided in appendix C in the "Switch.C" and "Switch1.C" versions together with the manual explaining how to create setup files. In appendix B, a sample output of the programme execution is supplied as well. To illustrate the principles of the simulation better, in the following chapters each of the basic classes of this library is described in more detail.

4.2.2 Class StorageObject

Class StorageObject contains the implementations of the functions common for the series of other objects. One of the important functions that the class StorageObject provides is function *connectsTo()*. Pseudocode of this function is shown in Figure 4.6. This function takes a reference of the Flow, which it is about to connect to the StorageObject and the connection ID of this connection as parameters. The chart explaining the process of connecting of any two objects of class StorageObject is in Figure 4.7. Each connection in an object of any subclass of class StorageObject has a connection ID. The Manager uses this connection ID in order to create the routing table and the objects use it to dynamically reconnect the outputs when forwarding cells to their destinations. The simulation equipment does not support run-time type identification; therefore, in order to differentiate between the input and the output connections, connection ID's for the input connections should be chosen equal or greater than 1000, and connection ID's for the output connections must be less than 1000. As it is shown in the Figure, an object of class Wire has one member called input of class InFlow and another member called output of class OutFlow. Both InFlow and OutFlow are derived from class Flow. When executing the function *connectsTo()*, any object of class StorageObject first asks the Flow, to which it is going to connect, the information about the object on the other side of the Wire using

function *set_communication()*. If, on the other side, no object is connected yet, the first object puts the pointer to itself and its connection ID inside the box contained by the Wire. Later another object will connect to the other side of the wire; it will take information about the adjacent object and transfer the information about itself to the neighbour with the help of function *take_connection()* as shown in the code in Figure 4.6. Thus the two objects of any of the subclasses of *StorageObject* connected by a Wire learn about each other in order to be able to pass information throughout the whole network. Objects of subclasses of *StorageObject* allocate one *FlowNode* for each connection and one *CDNode* for each input connection. An object of class *FlowNode* holds the information about the object on the other side of the Wire. An object of class *CDNode* contains variables describing the flow of

```
void StorageObject::connectsTo( Flow& f, int con_id )
{
    int adjacent_con_id = con_id;
    StorageObject* neighbour = NULL;

    neighbour = f.set_communication( this, &adjacent_con_id );
                                // Connect to one side of the Flow

    if( neighbour )
        neighbour->take_connection( adjacent_con_id, this, con_id );
                                // If on the other side somebody is
                                // connected, pass pointer to the
                                // neighbour

    flows.add( &f, con_id, neighbour, adjacent_con_id );
                                // Register this connection

    if( con_id < 1000 )
    {
        InFlow& inflow = (InFlow&)f;
        inflow.permanent_source = this;
        // Allocate additional objects
        // according whether this input
        // or output connection
    }
    else
        cd.add( con_id, defaultValue );
}
```

Figure 4.6. Pseudocode of class's *StorageObject* function *connectsTo()*.

data through this object. Class *StorageObject* has one more public function *latch()*, which is invoked by the main programme each clock cycle so that to fetch a value from the preceding object. Object of class *SwElement*, which is one of the subclasses of *StorageObject*, shown

in Figure 4.7 has three input connections and two output connections. Cooperation of the class SwElement members is described in detail in the next section.

Figure 4.7. Setting up of connections among objects of class StorageObject.

4.2.3 Class SwElement

Class SwElement describes the switchboxes and the sorting boxes as the building blocks of an ATM switch. The function of a switching element is to examine the destination address of an incoming cell and to establish a temporary connection between the input and the output for the time, during which the cell is being transferred, in order to forward this cell to the corresponding output. The function of a sorter is to forward two incoming cells to the corresponding outputs based on which of the cells has the higher destination address and which one has the lower destination address.

Described in the previous section switching elements shown in Figure 3.2 have two inputs and two outputs each. An object of class SwElement can have any arbitrary number of input and output connections. In this version of the library, the input side is limited by 999 connections because of the design problems. But the number of possible outputs can be increased if needed. Practically, there doesn't exist any switching element with the number of connections greater than 10 or even less, because of the interconnection problem in the VLSI implementation of the switch. However, in a shared queue the number of inputs and outputs may be up to 256.

An object of class SwElement can have six functional options shown in Table 4.1. The basic function that this class accomplishes is switching without buffering (option 0). In

this case a cell head when encountering a busy path offers backpressure that is sends a signal to the elements back in the chain for the bits to stop propagating through the fabric and another signal to resume motion when the path is free. In the later version of the library another feature was added to SwElement to put the arriving cells in queue when there is a conflict in this switching element (option 1). If the queue overflows, the incoming cells are

Option	Functional description
0	Unbuffered switch-box
1	Buffered switch-box
2, 4	Unbuffered sorter
3, 5	Buffered sorter (reserved for future use when improved)

Table 4.1. Functional options of an SwElement.

dropped. Size of the queue is set by the parameter and must be an integer greater than 0. There are four additional options to accomplish sorting. Examples of sorting elements are shown in Figures 3.15 and 3.16. With options 3 and 5 an SwElement becomes a sorter with usual queues, the sizes for which are set by the input parameter. While only a few bits that belong to the tag of a cell are used in the process of sorting to determine the output address. In option 3 the minimum corresponds to connection ID of 0, and the maximum -- to connection ID of 1 ("gray element"). In option 5 minimum and maximum are reversed ("black element"). Options 2 and 4 also make a sorter from SwElement. Option 2 corresponds to the "gray element", option 4 corresponds to the "black element". With these two options the queues can only hold the bits of the routing tag, and if the queue overflows, the sorter offers backpressure by sending the stall signal to the backward elements. This technique is an attempt to use the delayed push-out method discussed in section 3. Options 2 and 4 create so-called unbuffered sorter. The pair of options of 2 and 4 is more preferable for the simulation than the pair of 3 and 5. There is no guarantee than a buffered sorter will not rearrange cells; and as the simulation results show some unexpected thing happen with the buffered sorter. Therefore, options 3 and 5 are reserved for the future use when the simulation model will be improved to ensure that the cells are not reordered according to the ATM standard. Class StorageObject has the protected member called *sorting_option* to

identify the specific functionality in the objects of the inheriting classes. Objects with *sorting_option* greater than 0 do not transfer the busy signal to the backward element in the chain if they received it from the another object forward in the chain. For example, in class Queue this option is permanently set to 1, in other classes except SwElement this option is 0, and in SwElement the user sets this option as the parameter. Class SwElement also has the private member *int SRT* used to indicate whether it is a switchbox or a sorter, because the functions of these two alternatives differ significantly. *SRT* is set to 1 if *sorting_option* is greater than 1, otherwise it is 0.

If the simulation model is set up for the self-routing, objects of class SwElement must establish the temporary paths for the flow of data. Since the value of a bit can be either 1 or 0, outputs of a switching element must have the connection ID's of 1 and 0, and there can be no more than two outputs in a switch element theoretically. Nevertheless, this model of simulation assumes that a bit can have any integer value in order to be able to forward the cells to more than two outputs in a switch-box. A special variable called *defaultValue* in a CDNode (see Figure 4.7) is used to separate the streams of valid data bits making a switching element know where an ATM cell ends. *DefaultValue* is assigned the value greater than any other valid value of an ATM cell. *DefaultValue* is equal to *mask*, which is defined in "BaseClass.C".

To make the simulation more realistic instead of mimicking the manager's functions by manipulating with the bit values, some more functions may be added to the class Manager. To expand the potential of the library, the Manager can possibly get another capability of cell delineation performed by the Convergence Sublayer of the ATM Adaptation Layer.

This version of the library does not support broadcasting and merging. To make broadcasting and merging possible some functions of the Manager and SwElement may be enhanced.

Objects of class SwElement establish the routing paths by asking the FlowNode describing the forward link if it is vacant. If the link is free, the CDNode occupies that link and the link will be busy until the one bit trailer of an ATM cell equal to *defaultValue* releases it. A CDNode can engage only one output link and an output link can be engaged by only one CDNode, therefore neither broadcasting nor merging are possible in this version of the library.

In the case of a switchbox, if two ATM cells arrive simultaneously at the same element and occupy two CDNodes, the CNode, which was created first by the function *connectTo()*, will first engage the output link. The cell in the other CNode will be stalled until the first one releases the path. When a cell is stalled, the CNode holding it executes function *signal_stall()* each clock cycle in the backpressure approach (option 0). Function *signal_stall()* blocks and unblocks the ATM cell's path originating either in some of the objects of class InSocket or class Queue. When queuing option is installed in an SwElement (option 1), if a cell is stalled, it is just accumulated in the buffer contained in the switchbox. So, in the case of unbuffered network routing is accomplished in the wormhole fashion, and in the case of buffered network cut-through routing is employed. Arbitration between two cells is accomplished by the rule of a zip-fastener. If there are more than two inputs in a switching element, the first two CDNodes will have higher priority over all the rest of the CDNodes. So, the arbitration principle has not yet been developed for a multiinput switch. If there is any necessity of the multiinput switchbox use, the arbitration may be performed, for example, on the round-robin basis. To achieve such an arbitration principle, pointers to FlowNode's containing the output connections may be grouped in an array instead of a linked list because the pointers in the linked list are always accessed in the same order. The array, on the other hand, may be reorganized after each departure of a cell, thus reshuffling the priorities of the inputs. After a cell is allowed to go through a switchbox, its first bit is detached. The whole routing tag is detached from a cell when it makes the complete tour via the fabric.

In the case of a sorter the following algorithm works for determining the current cell direction. The routing tags of two cells gradually entering the sorter are compared each cycle. The routing tags enter the sorter with the most significant bits first; therefore it may be enough to compare only the first pair of bits. If they are equal, the comparison may advance to the next pair until the difference is found or the routing tags of the both cells have entered the sorter completely. Once the cell destinations are found different, the cells are forwarded to the corresponding outputs. If the routing tags completely enter the element, and the destinations are the same, the cells are both forwarded the input marked "min", thus preventing a conflict in the future. One cell in this case will be stalled until the other one releases the path. If only one cell arrived from one input of a sorter, it is blocked in the element until its tag has completely entered and only after that it is sent to the output

```

void SwElement::phase3()
{
    CDNode* tm_node;    QSet* q;
    for( tm_node = cd.first(); tm_node; tm_node = cd.next() ) // traverse the list of CDNode's
    {
        upgrade_status_1( tm_node );
        if(( !tm_node->admission ) && ( peek_value( tm_node ) < defaultValue ))
        {
            // Beginning of the main body
            if( !SRT ) // if it is a switch-box
            {
                tm_node->stall = OutSwitchTo( tm_node->connection_id, peek_value( tm_node ) );
                if( !sorting_option )
                    signal_stall( tm_node->preceding, tm_node->bkwd_con_id, tm_node->stall );
            }
            else // if it is a sorter
            {
                int dest = get_sort_set( tm_node->connection_id )->cel_destination;
                tm_node->stall = OutSwitchTo( tm_node->connection_id,
                                                ( sorting_option < 4 )? dest : 1 - dest );
            }
            upgrade_status_2( tm_node );
        } // End of the main body
        if( sorting_option ) // Update the variables describing queues
        {
            // and check the terminating condition in each object
            q = get_queue( tm_node->connection_id );
            if( q->occupation > q->max_q_length )
                q->max_q_length = q->occupation;
            if( q->occupation > 0 )
            {
                tm_node->p_status = continuing;
                q->q_length += q->occupation;
            }
            else
                tm_node->p_status = done;
        }
        if( sorting_option == 2 || sorting_option == 4 )
        {
            // Sorter backpressure component.
            if( q->occupation >= Q_SIZE )
                tm_node->backward_stall = 1;
            else
                tm_node->backward_stall = 0;
            signal_stall( tm_node->preceding, tm_node->bkwd_con_id, tm_node->backward_stall );
        } // End of sorter backpressure component
    }
    update_status(); // Update the termination condition
}

```

Figure 4.8. Pseudocode of function *phase3()* of class SwElement.

marked "min". This is because another cell may arrive a few cycles later and the decision about the directions may be premature. Destinations of any two cells are compared regardless of what are their current positions in the sorter. For example, one cell was already

forwarded to its destination because the other one didn't arrive before the routing tag of the first one was completely examined. Destination of the second cell will be inspected bit by bit each cycle until it is clear whether to forward this cell to the vacant output or to stall it and forward to the same output as the first one. This algorithm is implemented in function *check_sort_set()* of class SwElement.

Figure 4.8 gives the pseudocode of function *phase3()* of class SwElement describing how the switching and sorting functions of a SwElement are accomplished. Function *phase3()* is invoked after the bits have already been admitted by the inputs and the contents of CDNode's have been updated. This function traverses the list of CDNode's and makes decisions where to forward the cells and what signals to send. For each CDNode, first function *upgrade_status_1()* is called in order to determine whether it is a boundary of a cell, and if so, set variable *admission* to 0 to make it possible to enter the main body of the function as soon as a new cell arrives. If a cell is currently being pulled through a CDNode, the main body of the function will not be executed for that CDNode because no switching or signaling actions are needed. In the main body, function *OutSwitchTo()* is executed to establish the temporary path, and the output port for this function is determined according whether it is a switchbox or a sorter. After the attempt was made to connect to the output port, function *upgrade_status_2()* is executed to determine if the dynamic connecting was successful or it must be repeated because the needed output port was busy and a conflict has occurred. After the main body of the function *phase3()* goes the backpressure component implementation for the limited queues (options 2 and 4). If the queue exceeds a predetermined size, a busy signal is sent to the backward element. And finally, the statistical part starts. It is needed to update the maximum queue size and to upgrade the total of all the queue lengths achieved during each clock cycle in order to calculate the average queue size later. Each clock cycle function *update_status()* is invoked to find out whether this object is ready to finish the simulation process and for the Manager to decide whether to stop the simulation.

When the simulation is supposed to be stopped because all the traffic on the input side has been delivered to its destination, destructors in all the objects are invoked. Class SwElement performs the statistical calculations in its destructor. In the destructor, average queue lengths and arrival rates are calculated for each logical buffer as well as their overall figures for all the buffers. Values representing the numbers of cells entered, numbers of

conflicts, arrival rates, lost cells, and average and maximum queue lengths are placed into the corresponding arrays holding as many elements as the number of logical buffers or CDNode's. Then the Manager's function *calculate_statistics()* is called.

4.2.4 Class InSocket

Class InSocket is one of the subclasses of class StorageObject. Intention of an object of this class is to receive ATM cells from an input source that can be "plugged" into it. Currently, object of class InSocket can have two input options. One of them is to read the ATM cell traces from an input file in the directory "INPUTS" which is situated in the same directory as the executable files *run* and *run1*. The number of the file corresponds to the number of the object of class InSocket or, in other words, to its position in the column as shown in Figure 4.5. The member *input_option* is set to 0 in that case. Another option is to use the automatic traffic generator created on the basis of the source code of [8], with *input_option* is set to 1. Class InSocket may be interfaced to some other source of information. Alternate source of information may be an object representing an input module. Source code for an input module simulator is provided in [9]. The source code may be organized in such a way that to form the class of Input Module. An object of class InSocket is able to check the incoming data for validity and to discard those groups of characters that cannot be treated as valid ATM cells.

When a valid ATM cell arrives, an InSocket calls the Manager to attach a tag to this cell in order to be routed through the network. It also signals to the destination OutSocket to start timer for this cell in order to determine the time in clock cycles that will be needed to transfer this cell through the fabric. When there is no more input data, the InSocket sets its status to "done" in order for the Manager to read this status and make a decision whether or not to stop the simulation. If a cell is connected to a traffic generator, in order to turn off the InSocket, the traffic generator sets the value in its buffer to -1, which corresponds to the end of file in a trace file in the case when the inputs are taken from traces. An object of class InSocket can have only one output connection.

4.2.5 Class OutSocket

Class OutSocket is designed to send ATM cells further into the net when the switching fabric has properly routed them. Currently, objects of this class write the output data into the output files with the corresponding numbers inside the directory "OUTPUTS" in the same directory as the executable files *nm* and *nm1*. OutSocket can possibly be interfaced with an output module. An object of class OutSocket can have only one input connection. OutSockets participate in the creation of the routing table. The Manager builds up the routing table using the recursive function that traverses the fabric going from one object to another one, and the recursion stops when an object of Class OutSocket is reached. This routing table construction method is only suitable for the simulation model. In a real hardware switch such routing table is permanently built in.

Class OutSocket contains the functions needed to evaluate the delay for a cell to get from the input side to the output side. When a cell arrives at the input port, the destination OutSocket is asked to make a note of the current time by putting this record in the queue. There is an array of queues with the size equal to the number of input ports to keep the records of the time when a particular cell entered the fabric. Each cell has a non-standard embedded source number, which is the last byte, to be used as the index into the array of queues at the destination OutSocket. There is another sequence number that is the next to the last byte in a cell, to choose the appropriate record in the queue in case some cells have been lost. When the cell reaches the output port, the current time is measured and the record for the cell with the appropriate source and sequence numbers, taken from the queue, is subtracted from the current time to evaluate the cell delay. In this way the minimum, average and maximum delay is determined for all the cells having reached this destination. In its destructor class OutSocket calls the Manager in order for the Manager to record those values in *postscript.txt*.

4.2.6 Class Queue

Class Queue is a special modification of class SwElement. Objects of this class also allocate structures describing the flow of information when executing function *connectTo()*, but the number of inputs must always be equal to the number of outputs and each input

must have the connection ID equal to the corresponding output connection ID plus 1000. Unlike the objects of class SwElement, objects of class Queue cannot reestablish connections dynamically; connections are set up permanently instead. Figure 4.9 explains the structure of a Queue.

At the first clock cycle each Queue connects each input to the corresponding output. If there is more than one pair of connections, the Queue becomes shared. Each clock cycle, an object of class Queue reads one value from each input source. If for some reason the object following the Queue does not read a value from the Queue, the Queue starts accumulating the input values allocating a QNode for each of them. A QNode contains the integer member representing the value of the bit that reached the stage, in which the queue is located, and two pointers to the following and to the preceding

Figure 4.9. Structure of an object of class Queue.

QNode in the queue. The number of QNodes in a single queue cannot exceed the defined size of the queue. For a shared queue the total number of all QNodes cannot exceed the possible maximal size of the queue. When the size of the queue of QNodes becomes greater than some threshold value, the last ATM cell to enter the object of class Queue is discarded. Consequently, the smaller the maximal size of the queue, the greater the probability of a cell discarding in this queue. All the cells are treated with the same priority in this library. The functionality of class Queue can be possibly improved with another implementation of the queue. Currently, it is implemented as a doubly linked list, but it can have the form of a

circular buffer. In that case it would be possible to realize the partial buffer sharing, push-out or even delayed push-out strategies discussed in section 3.

The last stage of class Queue objects makes use of one more parameter called *suppressor*, which is a variable regulating the rate of outflow of bits from the Queue. This is needed to imitate the speeded up operation of the fabric. By default, the member *suppressor* is initialized to 0, in which case a Queue object returns values each clock cycle. If the variable *suppressor* is set to some positive value, the Queue returns values with intervals of several clock cycles, thus simulating the slow service rate of the output controller that sends the traffic to a slower transmission line than the switching fabric. The average size of the queue in such a case increases.

Class Queue performs almost the same statistical calculations in its destructor as the class SwElement. A Queue also evaluates the average queue lengths and arrival rates for each logical buffer and the overall figures for all the buffers together if it is a shared queue. As in SwElement values representing the numbers of cells entered, numbers of conflicts, arrival rates, lost cells, and average and maximum queue lengths are placed into the corresponding arrays of values of the same type, and the size of each array equals to the number of logical buffers. Manager records the data left after the queue is destructed as well.

4.2.7 Class Wire

Class Wire is designed to establish permanent connections among the objects of StorageObject's subclasses by passing the information about the neighbour to each other. The structure of this class is shown in Figure 4.7. It has the members *input* and *output* of class Flow that serve as the parameters for StorageObject's function *connectsTo()*. The function of this class used to assist in the fetching of a value from a StorageObject is called *pull()*. Each clock cycle the main programme invokes this public function for each object of class Wire.

4.2.8 Class Clock

There is only one instance of the object of Class Clock. This is a static class. Clock is needed to invoke the driving functions of *phase1()*, *phase2()* and *phase3()* in all of the objects inheriting from the ClockedObject. Clock also records the time and asks the Manager to check the statuses of all the objects each clock cycle to determine the termination condition for the simulation.

All the basic classes forming the ATM switch structure inherit from class ClockedObject. The three ClockedObject's pure virtual functions have their individual implementations in all the inheriting classes. When function *phase1()* is executed for all the objects, each object fetches the new value from another object that is back in the chain, and function *phase2()* substitutes the contents in each object by the new value. In this way, information flows from one object to another one in a pipeline fashion. Function *phase3()* is needed to transmit the control signals among the objects when the current values are already updated to synchronize the flow of information. These three functions are called in three separate series in each clock cycle. The Clock maintains the list of pointers to all the objects pertaining to classes inheriting from ClockedObject. In the majority of the cases the order, in which the Clock accesses those pointers to invoke *phaseN()* functions is irrelevant. But, if the backpressure component of the limited queues is involved, as discussed in the description of class SwElement, it is important that the signals are first originated in the objects of the last stages. For example, in the baseline network in Figure 4.5 signal should propagate from starting from stage 2 down to stage 0. In order to accomplish this, objects of the following stages must execute function *phase3()* prior to the objects in the preceding stages. Therefore, all the objects must be declared in order, and if necessary, the order of the pointer list may be reversed. Objects of similar classes must be declared in clusters besides that. To implement the delayed push-out strategy, an advanced synchronization technique may be needed. In such a case, it may be necessary reschedule the order, in which the objects are accessed by the Clock, and the pointers will need to be stored in an array rather than in a list.

Number of cycles in the Clock is recorded by two concatenated variables of type double. Maximal possible value of a double variable when it is still able to be incremented by 1 is 10^{15} . Since two variables are used together, the clock cycle count may go up to 10^{30} .

4.2.9 Class Manager

Class Manager is also a static class in the simulator. There is only one object of this class that does not need to be instantiated. Class Manager creates the routing table for the ATM switch at the first clock cycle by exhaustive traversing of the graph representing the switch network. Manager allocates the list to hold the tags for any pair of the source and the destination. In the case of a crossbar switch the routing table becomes enormously large so that the computer cannot allocate such an amount of memory to hold the routing table. By a rough estimation, the order of the table size is $N!^2$, where N is the number of outputs. To eliminate this problem, there is a special option for the crossbar switch, which is set up either by putting the string "-c" as a parameter in the argument line for programme "run.C" or by "manually" equating the *crossbar_option* to 1 in class Manager. When the *crossbar_option* is set to 1, the Manager uses a straight approach in the creating of the table. It assumes that a cell should go from an InSocket to an OutSocket making only one right angle turn, and there is no alternative path. In the crossbar switch, with the size 8 x 8, it is still possible to build the routing table by exhaustive graph traversal.

As a new ATM cell arrives in an InSocket, the Manager, according to the parameter *no_rearrangeability* with default value of 1, either looks for the optimal path for the cell or takes the first available tag in the array with the index of 0. The optimal path is computed by comparing the number of possible internal conflicts in the switch. A flow descriptor FlowNode has the variable *engagement* as a private member, which is used to keep track of how many cells are about to occupy this FlowNode, and how many conflicts can possibly occur. As one cell releases a FlowNode, it decrements the variable *engagement*. The Manager attaches the routing tag for an ATM cell and signals to increase the engagement for the path this cell will occupy within the same clock cycle as the ATM cell arrived.

In the simulation results, it is shown that for some traffic pattern the Benes ATM switch with seven stages pulls the data through within a smaller number of clock cycles than some other four-staged switches. The reason for that is the availability of alternate paths for the traffic in the seven-staged switch that eliminates a great number of conflicts. Such a switch is considered rearrangeable. Parameter *no_rearrangeability* may be reset "manually" or by the string "-r" in the argument list of programmes *run* or *run1*.

It is Manager's duty to check the status of each object in the programme, and if every object has the status set to "done", the Manager terminates the simulation.

After the simulation is stopped, all the objects start calling Manager asking to record the results of the simulation in the postscript.txt file. When an object is destroyed, a record appears in the output file. It is necessary that all the objects be declared in clusters of equal type because they are destroyed in the same order and the records are grouped according to the type of objects. The Manager also calculates the total and the overall parameters for all the objects of its class as it was mentioned earlier.

4.3 Formal description of the library elements

In this thesis work, a state machine and a Petri Net were chosen to illustrate the behavioural model of one switching element. The state machine, shown in Figure 4.10, describes the behaviour of one traffic node of a switch box, that is one input. It has three states: waiting, when no valid value arrives; switching, when the input is contending for an output; and pulling the traffic through. A valid value is a value representing a bit – 0 or 1. The alternative is the separator value denoting the end of cell. The first valid value to arrive represents the destination where to forward the cell. It puts the switch in a switching state until the desired destination is vacant. After that the traffic node goes to the state of continuous traffic passing, which is terminated by another cell separator.

Figure 4.10. State machine of one traffic node of a switch-box.

The Petri Net, shown in Figure 4.11, describes the behaviour of a whole switching element, consisting of two traffic nodes that contend for the two outputs, in other words, a 2x2 switch box. The Petri Net consists of two identical parts. Each part corresponds to one output of the switch-box. In the net all the conditions are marked with signs and all the transactions are numbered. The net has the following conditions, identical for the two outputs:

- "Ready" -- input is ready to receive a value;
- "0" -- input has received a 0 bit;
- "1" -- input has received a 1 bit;
- "S" -- input has received a value representing the end of cell;
- "V" -- this output is vacant, none of the inputs transmits cells to this output;
- "Thru" -- this output is engaged by an input and is passing bits through;
- "VE0" -- this output is vacant or engaged by input 0;
- "VE1" -- this output is vacant or engaged by input 1;
- "NVE0" -- not VE0;
- "NVE1" -- not VE1.

There are also the following transactions in the net:

- t1, t21 -- arrival of a 0 bit;
- t2, t19 -- arrival of a 1 bit;
- t3, t20 -- arrival of a bit representing a separator value;
- t4, t21 -- returning to ready state after a separator has been received;
- t5, t18 -- passing of a 0 bit to the next stage in the busy mode
- t6, t17 -- releasing the output when a separator arrived;
- t7, t16 -- passing of a 1 bit to the next stage in the busy mode;
- t8, t15 -- restoring the VE0 condition after a the output is released;
- t9, t14 -- restoring the VE1 condition after a the output is released;
- t10, t13 -- switching to the state when the output is engaged by input 0;
- t11, t12 -- switching to the state when the output is engaged by input 1.

Figure 4.11. Petri Net of a two-input, two-output switch-box.

Each of the identical parts of the Petri Net demonstrates the same functionality as that represented by the state machine in Figure 4.10. It describes in detail how the process of switching and passing traffic through the element proceeds. The main point here is to show that the switching cannot happen if the destination port is busy. In such a case the input has to wait until the output is released, which is shown by the conditions of transactions t_{10} , t_{11} , t_{12} , and t_{13} . This Petri Net was proved to be live by the method of unrolling the net into a tree.

This section described the core of the object-oriented network simulation model, which represents the switching part of the design. The other logical part of this model is the dynamic traffic generation facility that allows generating traffic with a specified distribution. This feature is presented in the next section.

5 Dynamic Traffic Generation

The ATM library allows simulating switch fabrics under the traffic, automatically generated by the traffic generation part based on the source code of [8]. The graduate thesis [8] was intended to simulate the queue behaviour with various traffic patterns applied to the source. In that work, switching fabric is not considered. The switching fabric is assumed to be perfect and not producing any conflicts. In this thesis, attention is focused on the switching fabric, but the traffic generation part is still suitable for the inputs to be produced. Originally, objects of class InSocket taking inputs from the trace files fed the switching fabric. Now InSockets can take input data from the plugged in traffic generators controlled by their managing object of class TManager. The UML diagramme for the traffic generation part is in Figure 5.1.

Figure 5.1 The traffic generation part UML diagramme.

In the UML diagramme it is shown that the classes TManager and TGen communicate with the classes from the switching part described before. To be specific, objects of class TManager receive the control signals from Clock, and objects of class TGen produce ATM cells to be consumed by InSocket's. In the following chapters the classes forming the traffic generation part are described in more detail.

5.1 Class TManager

Class TManager coordinates the process of traffic generation in all the input ports. If the system is to be simulated with the dynamic traffic generation, at least one object of this class needs to be instantiated. The object of class TManager keeps the list of all the traffic generators "plugged" into the input ports. The traffic generating objects are of class TGen. As an object of class TGen is declared, it is interfaced with an object of class InSocket, to which it is supposed to send traffic, and is also registered at the TManager. TManager takes the argument parameters when constructed. This list of character strings contains the information about the type of traffic distribution to be generated, the traffic parameters, the number of traces, and some other simulation factors. It is possible to declare several objects of class TManager for independent groups of inputs. Each object may take a different list of parameters. It is also possible to feed one group of inputs from traffic generators and another group from artificial traces. The object of class TManager allocates an object of class State when declared. Class State has a group of additional private members that describe the input port properties and facilitate in the organization of traffic distribution patterns. Prototypes and implementations of class State and other supporting classes are in files "TMSet.H" and "TMSet.C" respectively.

Class TManager inherits from class ClockedObject; therefore, it is also an active class. One cycle of class TManager is equal to 425 cycles of class Clock, since it takes 425 cycles to accept one ATM cell from input. Each clock cycle of the switching system the traffic generation system advances its clock by $1/425$ of a cycle. To imitate speedup of the switching system, the clock of the traffic generation system may be slowed down. The clock of the traffic generation system is represented by one variable of type double; therefore, it can go up to 10^{15} cycles.

The process of cell arrival at different inputs may be independent for each port as in Poisson, Pareto and Bursty traffic distributions. The arrival times at different ports may also depend upon each other, as in Zeta distribution. In Zeta distribution cells are fed to the input ports on the round-robin basis, and a single object of class TManager is needed to coordinate those arrivals. At the arrival time, separated from the previous arrival by an interarrival period, the object of class TManager signals a certain traffic generator to produce a cells with the provided output address. When the number of traces for a particular port has exceeded the limit, TManager shuts the corresponding traffic generator down. The necessary, but not sufficient condition for the simulation termination is shutting all the traffic generators down.

The main issue in TManager implementation is how to compute the interarrival times for the traffic generators. As it was already mentioned, this library allows generating Poisson, Pareto, Bursty, and Zeta distributions. Interarrival times are recalculated after each arrival event with the functions implemented in the file "TMFunctions.C". For Poisson and Pareto distribution new interarrival times are calculated using simple formulae with all coefficients known in advance, only a random number is needed to produce an interarrival time with such formula. In more complex Bursty and Zeta distributions, some additional tables are involved in the determining of an interarrival time. In this work, the switch fabric is only simulated under Poisson distribution traffic and the generation method for this distribution is described in more detail. The implementations of other types of distribution are also available; they were realized by restructuring of the source code of [8] to be used with the ATM library. There is also a possibility to read the interarrival times from a group of input files, thus simulating any arbitrary distribution.

5.2 Group of TMSet Classes

Files "TMSet.H" and "TMSet.C" provide the description of supplementary classes used by class TManager. Classes State, InputPorts, Correlation, Pz, and Bursty are implemented here. Class State is the main class in this series. One object of class State is allocated in a TManager. It contains the clock information, termination condition variable, and the variables representing next event in the system and speedup of the switching fabric - the factor by which traffic generation system clock is slowed down. An object of class

State allocates another object of class `InputPorts`, which describes the properties of the traffic fed to the input ports. Class `InputPorts` contains the variable representing the traffic parameters, the number of input ports, and the number of simulation traces. Names of variables are self-explanatory. The traffic parameter variables are `lambda` -- arrival rate for all distributions, and `hurst`, `epsilon`, and `theta` -- for Pareto distribution. Class `InputPorts` keeps the pointers to arrays of variables representing interarrival times and availability of traces for each input port. When declared, an object of class `InputPorts` allocates three additional objects of classes `Correlation`, `Pz`, and `Bursty`. Class `Correlation` keeps the current output address that each input port forwards cells to in the case of correlated traffic. `Pz` is the special class for the Zeta distribution, it keeps all the tables and variables for this type of distribution. And finally, `Bursty` is the class keeping parameters for Bursty distribution. There is the variable `traceType` in class `InputPorts` representing the traffic distribution produced. Detailed description of all the distribution types is provided in [8].

5.3 Traffic Manager Utility Functions

Class `TManager` uses a group of functions for the calculation of cell interarrival times. The intention of those functions is to determine the interarrival times for Poisson and Pareto, the burst lengths for Bursty and Zeta distributions, and the output address for each cell based on random value or correlation. All the functions are borrowed from [8] -- queue simulator under different traffic patterns. The three important functions used in this work are *double* `getRandom()`, *long* `getOutput.Address()`, and *double* `getPoissonTime()`. Function `getRandom()` produces a random number with the generator *drand48()*. One restriction on the random number generating function is that it is not allowed to produce 0 as an output. This is because the natural logarithm of a random number is calculated in function `getPoissonTime()`. The chance of a 0 result is eliminated in the function `getRandom()`. Function `getOutput.Address()` determines the output address for those cells that are generated first time or in an input port without correlation. Special attention is given to the function `getPoissonTime()`. This function calculates the interarrival time for Poisson distribution and returns a double value representing the time.

The interarrival time for Poisson distribution is calculated as follows. The original formula determining the probability that the next arrival time is lower or equal to some threshold T is:

$$p(t_a \leq T) = 1 - e^{-\lambda T}, \quad (5.1)$$

which is called cumulative distribution function with λ representing the mean arrival rate. The graph of this function for different arrival rates is in Figure 5.2.

Figure 5.2. Cumulative distribution function for Poisson distribution [23].

Source [22] suggests the method of generating a continuous random variate X that has the continuous and strictly increasing function $p(t)$, provided that $0 < p(t) < 1$. According to that method, the two steps should to be taken:

1. Generate U distributed in the interval $[0, 1]$.
2. Return $X = p^{-1}(U)$.

The inverse of the Poisson cumulative distribution function would be:

$$t = -\frac{1}{\lambda} \ln(1 - p(t)). \quad (5.2)$$

For the purpose of simulation it is necessary to determine the interarrival time, that is:

$$\Delta t = t_i - t_{i-1} = -\frac{1}{\lambda} (\ln(1 - p_i(t)) - \ln(1 - p_{i-1}(t))) = -\frac{1}{\lambda} \ln \frac{1 - p_i(t)}{1 - p_{i-1}(t)}. \quad (5.3)$$

Since $p_i(t)$ is always greater than $p_{i-1}(t)$, the result of expression $\frac{1 - p_i(t)}{1 - p_{i-1}(t)}$ is in the interval (0;1), and this expression may be assigned to variable U that is to be generated with the uniform probability on interval (0;1]. The boundary 0 is omitted because the interarrival time will become ∞ in this case. The boundary 1 is still included despite the resulting 0 interarrival time; such an event is unlikely however. The final formula for the Poisson interarrival time is:

$$\Delta t = -\frac{1}{\lambda} \ln U. \quad (5.4)$$

The mean value produced by this expression is equal to $\frac{1}{\lambda}$, which corresponds to the mean interarrival time defined as the reciprocal of mean arrival rate. An attempt to analytically verify the expression for the interarrival time is provided below. In this expression it is

$$\overline{\Delta t} = \frac{-\frac{1}{\lambda} \int_0^1 \ln U dU}{1 - 0} = -\frac{1}{\lambda} (U \ln U - U) \Big|_0^1 = -\frac{1}{\lambda} (-1 - 0) = \frac{1}{\lambda}. \quad (5.5)$$

assumed that $U \ln U$ is equal to 0 when U is 0 because U approaches 0 faster than $\ln U$ approaches $-\infty$. Another way to illustrate this would be to express U in terms of Δt and integrate by $d\Delta t$, as follows:

$$U = e^{-\lambda \Delta t} \quad (5.6)$$

$$\frac{-\frac{1}{\lambda} \int_0^1 \ln U dU}{1 - 0} = \int_0^\infty e^{-\lambda \Delta t} d\Delta t = -\frac{1}{\lambda} (\lim_{\Delta t \rightarrow \infty} e^{-\lambda \Delta t} - e^0) = \frac{1}{\lambda} \quad (5.7)$$

In a real simulation system the mean interarrival time will not always be equal to $\frac{1}{\lambda}$. The mean interarrival time approaches $\frac{1}{\lambda}$ when a large number of simulation traces is generated.

Testing of function *getPoissonTime()* showed that the mean interarrival time differs from $\frac{1}{\lambda}$ by only 0.00001 with the number of simulation traces equal to 50000. There is an upper bound on the arrival rate because the event of arrival to the system is not instantaneous. The upper bound is equal to 1, assuming that one time slot is equal to 425 cycles of the switching system -- the time required of the fabric to accept one ATM cell. If the arrival rate is set greater than 1, the system will saturate, and its behaviour will not be different from that with arrival rate equal to 1. However, this may not be true with a small number of simulation traces.

5.4 Class TGen

Class TGen describes a traffic generator that is "plugged" into an object of class InSocket. Objects of class TGen generate ATM cells consisting of 53 bytes represented by integers ranging from 0 to 255. An object of this class receives a signal from the TManager to produce an ATM cell with a particular output address that is passed as a parameter in the function *generateCell()*. The output address of an ATM cell is embedded into the VPI -- virtual path identifier. Each ATM cell also contains additional embedded source number and sequence number needed to determine the cell delay by the destination object of class OutSocket.

When there is no signal to generate a cell, a TGen sets its buffer to -2, which means that the InSocket interfaced with it will stay in "waiting" mode. When there has been a signal from the TManager to shut down by executing function *shut()*, a TGen sets its buffer to -1. This value, when accepted by an InSocket, puts the InSocket into "done" mode to make it possible to stop simulation when the other objects are also turned to "done" mode. An object of class TGen has a request queue. The request queue is needed because it takes 425 cycles of the switching system to accept one ATM cell, and another request may be

made by the TManager to generate a cell while the TGen is busy servicing the previous cell. TGen services the new request as soon as the previous one has been processed.

This section concludes the description of the ATM library functions allowing effective evaluation of various network parameters. In the next section the results of extensive simulations are provided for the 16-output and 256-output ATM switches.

6 Simulation Results

For the purpose of simulation, seven types of ATM switches have been constructed for sixteen outputs, which form the first group. They are `q_4_cube4x8`, `q_omega4x8`, `q_baseline4x8`, `q_banyan4x8`, `q_sorter_omega4x8`, `q_benes7x8`, and `q_crossbar16x16`. The layouts of those switches without queues are shown in Figures 3.10 and 3.13; they are taken from [6]. Switch architecture `q_sorter_omega4x8` was constructed by linking the sorting stage shown in Figure 3.16 and omega switching stage shown in Figure 3.10 (b). The Benes switch was tested in the usual mode, when an incoming cell engages the first path leading to the output, and with the enabled rearrangeability. All of them forward the traffic from 16 input files to 16 output files. Another group of switches for 256 outputs consists of `q_omega8x128`, `q_baseline8x128`, `q_4_cube8x128`, and `q_banyan8x128`. The performance evaluation of the first group was made with the artificial traffic patterns in the unbuffered and buffered versions and with the dynamic traffic generators with varying buffer size. The second group of networks was only tested with dynamic traffic generators under varying arrival rate because it is infeasible to create such enormous amount of traces in the input files.

ATM cells with arbitrary destination addresses placed into the input files contained in one directory form the traffic patterns. The name of the directory corresponds to the name of the traffic pattern. The arrival rate λ in a simulation with traffic patterns is always equal to 1, although it is possible to decrease the arrival rate by placing additional characters different from end of file between the groups of integers representing ATM cells. The first group of traffic patterns up to PAT5 was created only to observe the performance characteristics of all the types of fabrics without any intention to test the special features. Traffic pattern PAT5A was specially created to analyze the performance of Benes rearrangeable network. It produces a lot of internal conflicts in some networks without rearrangeability, but no conflicts at the final stage since the cells are forwarded to different output ports. To examine the performance of the sorting network, traffic patterns PAT7, PAT8, and PAT9 were created one after another. Each new pattern was an attempt to find a better correlation of output addresses producing as much as possible internal conflicts, but allowing significant reduction of the number of conflict occurrences and their duration by means of sorting.

Such a pattern could give a substantial variation of results of sorting versus unsorting networks.

The parameters evaluated are basically the same in each type of simulation and are related. Generally speaking, of the two results, measured on two different networks, one is better than the other, if it has a lower simulation time, which means that it takes less time to pull the traffic through the fabric. It is likely that the network with lower clock cycle time will produce fewer conflicts, cell delay will be lower, and the queues will be smaller, although this may not be true for buffered networks. Throughput is inversely proportional to the simulation time unless the switch has discarded some cells. Throughput is the ratio of the total number of returned cells to the simulation time normalized to time slots.

The other simulation parameters are minimum, average, and maximum cell delays. The minimum cell delay is normally a constant value for each type of network and is equal to the time required to pull one cell through without any conflicts. The maximum cell delay is the worst time with which a particular network transfers cells from an input port to an output port. The average cell delay is calculated for all the active output ports that have received any ATM cells from the fabric. The number of conflicts is also a useful performance measure, but sometimes it is not very accurate because the number of conflicts does not represent the cumulative conflict time of the system. The number of conflicts may be large, but their duration is short, so the total waiting time is quite moderate. An instance of this is shown in the buffered network simulation section where the buffered network q1_crossbar16x16 has a larger number of conflicts than q_crossbar16x16 but the simulation time is still lower. Other estimated parameters are minimum, average, and maximum queue sizes. The maximum queue size is the largest queue size ever achieved in the network during the simulation. The average queue size is the total cumulative size of a queue divided by the simulation time. The average queue size is determined according to the same rule as in [8], but the process of calculation is easier with this library. The average queue size is defined as:

$$\bar{q}(t) = \frac{\int_0^T Q(t)dt}{T}, \quad (6.1)$$

where T is the simulation time, and the part in the numerator is calculated by adding the queue length to the total each cycle. The overall average queue size is calculated as the average of all average queue sizes in the network. Besides that, the overall average is presented in two forms: calculated among all the queues and calculated only among active queues, since some of them may not be utilized. The cell loss probability is computed for the networks simulated under varying buffer size. For all fabric types throughput is defined as the ratio of the number of cells returned by the system to the simulation time in time slots. One slot is equal to 425 clock cycles, i.e. the time to admit one cell to the system. Under saturated load and with no conflicts, the throughput approaches the number of output ports of the system, but does not reach it due to additional clock cycles spent in the routing process.

6.1 Simulation of unbuffered networks

In this simulation step seven switch architectures forming the first group were simulated without internal buffering turned on. All the switch-boxes are built with option 0 as described in section 4, and the sorter is also unbuffered. The networks of this group are complemented with two stages of queues -- one stage of input queues and one shared output queue. The setup file names of those switches have prefix "q_". Tables 6.1 through 6.9 and Figures 6.1 through 6.9 show the set of results produced by testing the first group of switches. The simulation results give the number of clock cycles, number of conflicts, throughput, cell delays, and sizes of the queues.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_4_cube4x8	27244	41311	13690	27307	102580	54603	27335	29439	12437	13721
q_banyan4x8	27244	26861	23852	51084	102580	102156	27244	23437	6873	6873
q_sorter_omega4x8	27562	28371	32184	52136	108869	106728	37929	24713	6977	6977
q_crossbar16x16 -c	27673	15607	13836	27647	55749	55293	27602	13677	7210	7332
q_omega4x8	27244	41311	13690	27307	102580	54603	27335	29439	12437	13721
q_baseline4x8	27244	41311	23017	51084	102580	102156	28100	29439	12427	13721
q_benes7x8	27439	54863	54863	54863	109711	109711	109711	108430	70363	55418
q_benes7x8 -r	27250	40477	13696	27313	102586	54609	35443	40174	6924	6924

Table 6.1. Simulation time.

(a)

(b)

Figure 6.1. Simulation time.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_4_cube4x8	241	429	244	370	979	754	500	367	190	248
q_banyan4x8	241	263	338	481	979	977	500	285	0	0
q_sorter_omega4x8	0	93	110	121	253	249	107	92	0	0
q_crossbar16x16 -c	105	84	180	195	322	431	369	15	130	0
q_omega4x8	241	429	244	370	979	754	500	367	190	248
q_baseline4x8	241	429	335	481	979	977	498	367	172	248
q_benes7x8	241	497	497	497	1009	1009	1009	1003	825	755
q_benes7x8 -r	267	424	284	438	1160	918	577	491	0	0

Table 6.2. Number of conflicts.

(a)

(b)

Figure 6.2. Number of conflicts.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_4_cube4x8	0.9988	1.275	3.9738	1.993	1.0625	1.9933	3.9823	3.6975	8.7465	7.9305
q_banyan4x8	0.9988	2.0273	2.2823	1.063	1.0625	1.0625	3.995	4.641	15.827	15.831
q_sorter_omega4x8	0.986	1.9168	1.6915	1.046	0.9988	1.0158	2.8688	4.403	15.598	15.598
q_crossbar16x16 -c	0.9818	3.485	3.9313	1.968	1.9508	1.9678	3.9398	7.9475	15.088	14.833
q_omega4x8	0.9988	1.275	3.9738	1.993	1.0625	1.9933	3.9823	3.6975	8.7465	7.9305
q_baseline4x8	0.9988	1.275	2.363	1.063	1.0625	1.0625	3.8718	3.6975	8.755	7.9305
q_benes7x8	0.9903	0.9903	0.9903	0.99	0.9903	0.9903	0.9903	1.003	1.547	1.9635
q_benes7x8 -r	0.9945	1.343	3.9738	1.993	1.0625	1.9933	3.0685	2.7073	15.712	15.712

Table 6.3. Throughput.

(a)

(b)

Figure 6.3. Throughput.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_4_cube4x8	436	436	436	436	436	436	436	436	436	436
q_banyan4x8	436	436	436	436	436	436	436	436	436	436
q_omega4x8	436	436	436	436	436	436	436	436	436	436
q_baseline4x8	436	436	436	436	436	436	436	436	436	436
q_benes7x8	442	442	442	442	442	442	442	442	442	442
q_benes7x8 -r	442	442	442	442	442	442	442	442	442	442
q_sorter_omega4x8	502	502	494	499	502	499	487	502	480	480
q_crossbar16x16 -c	444	430	430	430	430	430	430	430	430	430

Table 6.4. Minimum cell delay.

Figure 6.4. Minimum cell delay.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_4_cube4x8	13195.5	20857.75	5560.5	12369	48289.5	24301	10667.16	13884.4	3107.3	3860
q_banyan4x8	13195.5	13627	10641.5	24257.5	48289.5	48077.5	10621.5	9291.94	436	436
q_sorter_omega4x8	13387.5	13291.87	14724.23	24181.7	51662.8	49431.4	15937.24	8518.43	510	510
q_crossbar16x16 -c	13316.9	6237.75	5568.86	12437.5	24665	24437.5	10672.18	3675	466.64	460
q_omega4x8	13195.5	20857.75	5560.5	12369	48289.5	24301	10667.16	13884.4	3107.3	3860
q_baseline4x8	13195.5	20857.75	10402.19	24257.5	48289.5	48077.5	10851.14	13884.4	3194.77	3860
q_benes7x8	13291.5	26139.5	26139.5	26139.5	51835.5	51835.5	51835.5	51195	31894.6	24689
q_benes7x8 -r	13197	20960.23	5556	12364.5	48273	24284.5	13121.21	20773.2	442	442

Table 6.5. Average cell delay.

(a)

(b)

Figure 6.5. Average cell delay.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_4_cube4x8	25955	38306	10685	24302	96143	48166	20898	23002	6000	7284
q_banyan4x8	25955	23856	20847	48079	96143	95719	20807	17000	436	436
q_sorter_omega4x8	26273	25366	29179	49131	102432	100291	31492	18276	540	540
q_crossbar16x16 -c	26327	12509	10755	24561	49125	48683	20983	6955	514	490
q_omega4x8	25955	38306	10685	24302	96143	48166	20898	23002	6000	7284
q_baseline4x8	25955	38306	20012	48079	96143	95719	21663	23002	5990	7284
q_benes7x8	26141	51837	51837	51837	103229	103229	103229	101948	63881	48936
q_benes7x8 -r	25952	37451	10670	24287	96104	48127	28961	33692	442	442

Table 6.6. Maximum cell delay.

(a)

(b)

Figure 6.6. Maximum cell delay.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_4_cube4x8	753.75	1355.09	1026.63	1331.03	2843.6	2664.51	2054.04	1566.98	737.28	856.45
q_banyan4x8	753.75	1346.32	1171.9	1419.99	2843.6	2842.75	2049.48	1296.02	0.994	0.994
q_sorter_omega4x8	745.29	1226.7	1199.7	1372.25	2853.57	2783.41	2223.47	1109.51	17.564	17.566
q_crossbar16x16 -c	767	1039.05	1033.89	1344.93	2697.95	2693.31	2076.08	834.7	4.167	0.945
q_omega4x8	753.75	1355.09	1026.63	1331.03	2843.6	2664.51	2054.04	1566.98	737.28	856.45
q_baseline4x8	753.75	1355.09	1187.05	1419.99	2843.6	2842.75	2033.5	1566.98	761.87	856.45
q_benes7x8	758.96	1292.64	1292.64	1436.27	2875.41	2875.41	2587.87	1616.14	1543.5	1510.89
q_benes7x8 -r	754.14	1396.23	1028.02	1334.77	2857.55	2676.15	1974.75	1746.5	0.99	0.99

Table 6.7. Average queue size calculated only on active queues.

(a)

(b)

Figure 6.7. Average queue size calculated only on active queues.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_4_cube4x8	400.42	846.93	641.65	748.7	1599.52	1498.79	1283.78	1566.98	737.28	856.45
q_banyan4x8	400.42	841.45	732.44	798.74	1599.52	1599.05	1280.93	1296.02	0.994	0.994
q_sorter_omega4x8	395.93	766.69	749.82	771.89	1605.13	1565.67	1389.67	1109.51	17.564	17.566
q_crossbar16x16 -c	407.47	649.4	646.18	756.52	1517.6	1514.99	1297.55	834.7	4.167	0.945
q_omega4x8	400.42	846.93	641.65	748.7	1599.52	1498.79	1283.78	1566.98	737.28	856.45
q_baseline4x8	400.42	846.93	741.9	798.74	1599.52	1599.05	1270.94	1566.98	761.87	856.45
q_benes7x8	403.19	807.89	807.89	807.89	1617.42	1617.42	1617.42	1616.14	1543.5	1510.89
q_benes7x8 -r	400.64	872.64	642.51	750.8	1607.37	1505.33	1234.22	1746.5	0.99	0.99

Table 6.8. Overall average queue size.

(a)

(b)

Figure 6.8. Overall average queue size.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_omega4x8	1714	3428	2572	3001	6433	6004	5146	6433	3422	3430
q_baseline4x8	1714	3428	3424	3428	6433	6433	5575	6433	3418	3430
q_benes7x8	1726	3452	3452	3452	6478	6478	6478	6478	6476	6439
q_benes7x8 -r	1720	3446	2584	3016	6472	6040	5614	6472	1	1
q_4_cube4x8	1714	3428	2572	3001	6433	6004	5146	6433	3422	3430
q_banyan4x8	1714	3428	2999	3428	6433	6433	5142	6383	1	1
q_sorter_omega4x8	1711	3427	3402	3417	6849	6834	6420	6814	65	65
q_crossbar16x16 -c	1774	3536	3524	3524	7060	7056	7066	6466	46	1

Table 6.9. Maximum queue size.

(a)

(b)

Figure 6.9. Maximum queue size.

The point of these simulation results is to demonstrate that with varying traffic patterns different switch architectures behave differently. Taking to account various parameters obtained from the simulation of each switch architecture, it can be noted that banyan4x8, omega4x8, baseline4x8, and 4_cube4x8 sometimes produce identical results because they all consist of the same number of switch boxes and the topological equivalency among them takes place. Benes network without the enabled rearrangeability option always performs worse because it has the largest number of switch-box stages. This architecture was tested in order to show the difference in the performance when rearrangeability is enabled and disabled. Benes switch 7 x 8 with turned on rearrangeability option performs better with some traffic patterns than the mentioned above four switches because it has the alternate paths allowing to reduce the number of internal conflicts.

Rearrangeable networks are useful when the traffic supplied to the inputs is highly correlated and the particular correlation model produces a large number of conflicts in the networks without rearrangeability. At the same time, in some other networks without rearrangeability the number of conflicts may be no greater than in the rearrangeable ones. For example, the number of conflicts, and clock cycles correspondingly, is usually equal in the pairs of 4_cube -- omega and banyan -- baseline. This can be explained by the fact that in those pairs, cells from the same input ports enter the corresponding switch boxes. If the destination tags of any two cells differ only in one pair of bits (0000 and 0001), then these cells are in conflict in three stages in one pair. In the other pair, these cells may not be at conflict at all because they enter different switch boxes. The effect of rearrangeability is demonstrated in Figure 6.1. Here with traffic pattern 5A pair banyan -- baseline gives the number of clock cycles of 102156, while pair 4_cube -- omega produces the number of cycles 54603, which is twice less. The rearrangeable network Benes generates 54609 cycles that is almost the same as the above mentioned pair. This traffic pattern was specially selected to produce maximum number of conflicts in the pair banyan -- baseline. Another correlated traffic pattern could cause the maximum number of conflicts in the other pair 4_cube -- omega. The rearrangeable network helps to reduce the duration of a conflict using the special algorithm described in section 4. Although the number of conflicts itself is not reduced significantly, the conflict duration is lower and the overall clock cycle count is decreased. Therefore, if the traffic pattern correlation is not known in advance, rearrangeable network is practical.

The crossbar switch performs better than the rest of the switches, unless a particular banyan switch is well suited for a given traffic distribution or special features as sorting are involved. For example, with traffic pattern PAT5 the throughput of the crossbar switch is twice greater than that of the other switches (see Fig. and Table 6.3). Its performance is much less affected under varying traffic patterns. Since there are always enough available paths in a crossbar switch, only those cells forwarded to the same destination simultaneously are in conflict. One drawback of a crossbar switch is that it has significantly larger amount of switch-boxes than that of any other network.

Another performance evaluation is done with the sorting network. The sorting network performs better than the majority of the other networks without special features when one of the specially selected traffic patterns 7, 8 or 9 are loaded. Sorting networks reduce the number of internal conflicts by properly arranging the traffic supplied to the input of the switching stage. Rearrangeable networks, on the other hand, route the cells that are in conflict by alternative paths. There is one instance, namely pattern PAT7, where the sorting network performs better than the rearrangeable one, because its clock cycle time is lower. This can be explained by the fact that the rearrangeable network may not have enough additional paths to route all the cells that are in conflict, while the sorting network eliminates all the conflicts, however at a price of greater number of switch-boxes.

6.2 Simulation of buffered networks

In this step the `q_banyan4x8` and the `q_crossbar16x16` from the first group of switches for 16 outputs were tested with the buffering turned on. The purpose of this simulation phase is to compare the performance of the same networks with and without buffers in each switch-box. Parameters evaluated and compared are the same as for the previous group. The option in all the switch-boxes was set to 1, and the buffer size was chosen large enough not to discard any cells, otherwise the results would be incomparable with those of the unbuffered versions. The other two stages of queues left from the original unbuffered setup do not affect the simulation results in buffered networks. The rearrangeable and the sorting networks were not tested with buffering turned on because they allow reordering of cells in this mode. The traces of the buffered versions are marked

with strings having prefixes "q1_". Tables 6.10 through 6.19 and Figures 6.10 through 6.19 show the set of results of testing this group of switches.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	27244	26861	23852	51084	102580	102156	27244	23437	6873	6873
q1_banyan4x8	27248	27259	23848	51080	102569	102144	27248	17905	6877	6877
q_crossbar16x16 -c	27673	15607	13836	27647	55749	55293	27602	13677	7210	7332
q1_crossbar16x16 -c	27289	14928	13656	27273	54957	54533	27297	13363	7239	7363

Table 6.10. Simulation time.

(a)

(b)

Figure 6.10. Simulation time.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	241	263	338	481	979	977	500	285	0	0
q1_banyan4x8	241	430	330	481	979	977	500	576	0	0
q_crossbar16x16 -c	105	84	180	195	322	431	369	15	130	0
q1_crossbar16x16 -c	105	132	194	217	338	449	398	240	144	0

Table 6.11. Number of conflicts.

(a)

(b)

Figure 6.11. Number of conflicts.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	0.9988	2.0273	2.282	1.0625	1.063	1.0625	3.995	4.641	15.827	15.8313
q1_banyan4x8	0.9988	1.9933	2.282	1.0668	1.063	1.0651	3.995	6.0733	15.823	15.8228
q_crossbar16x16 -c	0.9818	3.485	3.931	1.9678	1.951	1.9678	3.9398	7.9475	15.088	14.8325
q1_crossbar16x16 -c	0.9988	3.6423	3.982	1.9933	1.981	1.9933	3.9865	8.143	15.028	14.7773

Table 6.12. Throughput.

Figure 6.12. Throughput.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	436	436	436	436	436	436	436	436	436	436
q1_banyan4x8	440	440	440	440	440	440	440	440	440	440
q_crossbar16x16 -c	444	430	430	430	430	430	430	430	430	430
q1_crossbar16x16 -c	452	431	431	431	431	431	431	431	431	431

Table 6.13. Minimum cell delay.

Figure 6.13. Minimum cell delay.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	13195.5	13627	10641.5	24257.5	48290	48077.5	10621.5	9291.9	436	436
q1_banyan4x8	13199.5	12347	10641.5	24257.5	48286	48073.5	10625.5	5954	440	440
q_crossbar16x16 -c	13316.9	6237.75	5568.86	12437.5	24665	24437.5	10672.18	3675	466.6	460
q1_crossbar16x16 -c	13207.4	6156.4	5519.24	12330.8	24435	24223.2	10587.3	3592.3	484.6	476

Table 6.14. Average cell delay.

(a)

(b)

Figure 6.14. Average cell delay.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	25955	23856	20847	48079	96143	95719	20807	17000	436	436
q1_banyan4x8	25959	24254	20843	48075	96132	95707	20811	11468	440	440
q_crossbar16x16 -c	26327	12509	10755	24561	49125	48683	20983	6955	514	490
q1_crossbar16x16 -c	25943	11866	10606	24196	48345	47924	20689	6821	542	521

Table 6.15. Maximum cell delay.

(a)

(b)

Figure 6.15. Maximum cell delay.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	753.8	1346.3	1171.9	1420	2843.6	2842.8	2049.5	1296	0.994	0.994
q1_banyan4x8	426.3	542.33	531.06	849.08	1698.9	1698.4	728.68	526.26	0.995	0.995
q_crossbar16x16 -c	767	1039.1	1033.9	1344.9	2698	2693.3	2076.1	834.7	4.167	0.945
q1_crossbar16x16 -c	90.04	169.88	164.83	387.28	775.99	775.05	237.73	52.09	0.761	0.94

Table 6.16. Average queue size calculated only on active queues.

(a)

(b)

Figure 6.16. Average queue size calculated only on active queues.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	400.42	841.45	732.44	798.74	1599.52	1599.05	1280.93	1296.02	0.994	0.994
q1_banyan4x8	199.83	372.86	365.1	398.01	796.34	796.11	637.59	526.26	0.995	0.995
q_crossbar16x16 -c	407.47	649.4	646.18	756.52	1517.6	1514.99	1297.55	834.7	4.167	0.945
q1_crossbar16x16 -c	25.96	42.47	41.2	48.41	96.99	96.88	82.44	52.09	0.761	0.94

Table 6.17. Overall average queue size.

(a)

(b)

Figure 6.17. Overall average queue size.

	PAT1	PAT2	PAT3	PAT4	PAT5	PAT5A	PAT6	PAT7	PAT8	PAT9
q_banyan4x8	1714	3428	2999	3428	6433	6433	5142	6383	1	1
q1_banyan4x8	6815	6389	5936	12779	25559	25438	6815	2549	1	1
q_crossbar16x16 -c	1774	3536	3524	3524	7060	7056	7066	6466	46	1
q1_crossbar16x16 -c	1748	865	866	1742	3481	3482	1729	438	51	1

Table 6.18. Maximum queue size.

(a)

(b)

Figure 6.18. Maximum queue size.

These simulation results show that buffered networks generally behave better than those without buffers. With almost all the traffic patterns the clock cycle time is slightly lower for buffered networks, see Figure and Table 6.10. This parameter is normally reduced by 2%, and with pattern PAT2 the simulation time of the crossbar network is reduced by 5%. Only with the traffic patterns producing a few conflicts such as PAT8 and PAT9 and with pattern PAT1 where the cells are forwarded to the same output port, the simulation time is higher for the buffered versions. Clock cycle time for the buffered versions is slightly greater with no conflicts because of some details of the library implementation. The variance of the simulation time in the banyan and the crossbar switches is different. The presence of buffers inside the switching elements changes the order in which conflicts occur and their number. Cells that are in conflict block all the preceding stages in unbuffered networks, while in buffered networks the previous stages are freed soon after a conflict has occurred, thus allowing new cells to enter the fabric. New entering cells, however, may create additional conflicts; therefore the total number of conflicts may be large. The average and maximum cell delays are also generally lower in buffered networks. But there are some cases when they are higher than those of the unbuffered networks are. Such cases happen with the same traffic patterns with more conflicts in buffered switches. And finally, the queue sizes are not directly comparable between the two versions of networks. The queues in buffered networks are distributed among the switching elements, while in unbuffered networks they are concentrated in the input and the output stages. Nevertheless, queues are generally shorter in buffered networks with some exceptions. In banyan network the maximum queue size raises highly with the patterns PAT4, PAT5, and PAT5 due to some temporary conflicts, but the overall queue size stays lower than that in the unbuffered version.

6.3 Simulation of networks under varying arrival rates

This set of tests was run to measure the performance of networks under varying arrival rate of Poisson distribution. The switches for this simulation were chosen of the biggest size, for the number of outputs equal to 256. Four basic types of switches of the second group were tested. This was the longest simulation step. The crossbar switch for 256 outputs was not investigated because it took enormously long time to run even a single simulation with it. Crossbar switch is not as scalable as the other types of switches. The arrival rate of the traffic applied to each input port is spread on the interval from 0 to 1. With λ greater than 1 the system saturates and no differences in the simulation show up. The output addressing of the cells is random. Tables 6.20 through 6.27 and Figures 6.20 through 6.27 show the simulation results.

Lambda	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
q_banyan8x128	153457	102795	91940	91258	87561	91949	89378	89457	88510
q_omega8x128	153457	102795	92208	93018	94152	89371	89024	90151	88987
q_4_cube8x128	153457	102795	92208	93018	94152	89371	89024	90151	88987
q_baseline8x128	153457	102795	90894	87895	88580	88040	92665	88908	87177

Table 6.19. Simulation time.

Figure 6.19. Simulation time.

Lambda	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
q_banyan8x128	13145	21741	22973	23533	23687	23436	23686	23469	23441
q_omega8x128	13015	21677	23158	23464	23339	23587	23475	23709	23661
q_4_cube8x128	13015	21677	23158	23464	23339	23587	23475	23709	23661
q_baseline8x128	13052	21929	23094	23325	23539	23516	23577	23423	23662

Table 6.20. Number of conflicts.

Figure 6.20. Number of conflicts.

Lambda	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
q_banyan8x128	35.449	52.917	59.169	59.611	62.127	59.164	60.864	60.813	61.464
q_omega8x128	35.449	52.917	58.999	58.48	57.779	60.869	61.107	60.342	61.132
q_4_cube8x128	35.449	52.917	58.999	58.48	57.779	60.869	61.107	60.342	61.132
q_baseline8x128	35.449	52.917	59.849	61.893	61.413	61.791	58.705	61.187	62.403

Table 6.21. Throughput.

Figure 6.21. Throughput.

Lambda	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
q_banyan8x128	444	444	444	444	444	444	444	444	444
q_omega8x128	444	444	444	444	444	444	444	444	444
q_4_cube8x128	444	444	444	444	444	444	444	444	444
q_baseline8x128	444	444	444	444	444	444	444	444	444

Table 6.22. Minimum cell delay.

Figure 6.22. Minimum cell delay.

Lambda	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
q_banyan8x128	1186.45	7558.64	14743.97	19775.56	23026.79	25248.79	27531.53	28148.06	28994.01
q_omega8x128	1192.15	7290.49	15515.88	20173.73	23444.17	26003.71	26910.14	28659.4	28902.72
q_4_cube8x128	1192.15	7290.49	15515.88	20173.73	23444.17	26003.71	26910.14	28659.4	28902.72
q_baseline8x128	1230.9	7209.23	14821.13	19422.33	23210.71	25786.67	27227.72	28207.08	28826.6

Table 6.23. Average cell delay.

Figure 6.23. Average cell delay.

Lambda	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
q_banyan8x128	12170	40449	53853	59736	61075	67804	65405	66125	65798
q_omega8x128	9839	44019	52817	58240	59766	66036	63790	68241	67023
q_4_cube8x128	9839	44019	52817	58240	59766	66036	63790	68241	67023
q_baseline8x128	8609	39314	47232	53851	59265	60235	66203	67264	65409

Table 6.24. Maximum cell delay.

Figure 6.24. Maximum cell delay.

Lambda	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
q_banyan8x128	52.075	748.13	1682.37	2291.75	2790.45	2918.88	3279.29	3351.02	3490.35
q_omega8x128	52.48	719.91	1768.07	2294.71	2643.11	3094.49	3216.82	3386.59	3460.52
q_4_cube8x128	52.48	719.91	1768.07	2294.71	2643.11	3094.49	3216.82	3386.59	3460.52
q_baseline8x128	55.2	711.36	1710.91	2335.96	2780.81	3114.59	3127.48	3378.92	3522.92

Table 6.25. Average queue size.

Figure 6.25. Average queue size.

Lambda	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	1
q_banyan8x128	3029	11689	13421	15153	16452	17318	17751	17751	19044
q_omega8x128	3456	12118	15139	16019	17316	17745	18184	17749	18182
q_4_cube8x128	3456	12118	15139	16019	17316	17745	18184	17749	18182
q_baseline8x128	4318	12122	14466	15586	16017	18182	17312	18182	18184

Table 6.26. Maximum queue size.

Figure 6.26. Maximum queue size.

The simulation results show that all parameters except minimum cell delay and simulation time increase monotonically with the arrival rate changing from 0 to 1. The reason for this is the increased saturation of the system with the increasing arrival rate. The results for the networks q_omega8x128 and q_4_cube8x128 are absolutely identical. The results for the rest of the networks vary from one point of simulation to another, and there is no rule according to which the results vary because the output addressing is not systematic. The minimum cell delay is uniform for all the switch types because the number of stages is the same in all networks. The simulation time decreases with the increasing arrival rate because the system is idling with low arrival rates. This simulation is conducted under non-instantaneous arrival events, in which case the graphs are saturated at the arrival rate equal to 1. Otherwise the simulation graphs of the cell delays and queue sizes asymptotically approach the vertical line corresponding to the arrival rate of 1. It is shown in the various graphs of analytical and simulation evaluation in [20] and [21].

6.4 Simulation of buffered networks under varying buffer sizes

This test suite is designed to assess switch performance under varying buffer sizes. The first group of networks for 16 outputs is tested. The Benes rearrangeable and sorting networks are not included in the simulation because buffered simulation is unacceptable when the cell rearranging capability is enabled in a switch. In these simulations the additional factor of loss probability is considered apart from the commonly estimated parameters. The clock cycle time does not make sense in this simulation because cells are dropped when buffers overflow, throughput is more appropriate to evaluate the performance. The results are shown in tables and Figures 6.28 through 6.35.

Buffer size	25	24	23	22	21	20	19	18	17
q_banyan4x8	9.4053	9.4435	9.4775	9.4563	9.396	9.51023	9.52	9.435	9.58
q_4_cube4x8	9.486	9.4733	9.4605	9.4988	9.486	9.46475	9.435	9.33725	9.639
q_omega4x8	9.486	9.4733	9.4605	9.4988	9.486	9.46475	9.435	9.33725	9.639
q_baseline4x8	9.7155	9.707	9.6433	9.6305	9.665	9.6815	9.8983	10.0173	10.05
q_crossbar16x16	10.472	10.472	10.472	10.472	10.47	10.472	10.472	10.472	10.47

(a)

Buffer size	16	14	12	10	8	6	4	2	1
q_banyan4x8	9.3118	9.5115	9.656	9.529	9.2693	9.061	8.56205	7.73075	6.85228
q_4_cube4x8	9.4053	9.8005	9.6093	9.682	9.3798	9.1205	8.5765	7.64575	6.69375
q_omega4x8	9.4053	9.8005	9.6093	9.682	9.3798	9.1205	8.5765	7.64575	6.69375
q_baseline4x8	10.013	9.6943	9.877	9.648	9.4393	9.09075	8.602	8.43625	6.766
q_crossbar16x16	10.472	10.472	10.472	10.47	10.472	10.4665	10.4338	10.2553	9.4265

(b)

Table 6.27. Throughput.

Figure 6.27. Throughput.

Buffer size	25	24	23	22	21	20	19	18	17
q_banyan4x8	2073	2076	2059	2079	2078	2061	2032	2016	1997
q_4_cube4x8	2047	2042	2034	2027	2028	2015	2009	2011	1990
q_omega4x8	2047	2042	2034	2027	2028	2015	2009	2011	1990
q_baseline4x8	2070	2075	2076	2077	2080	2058	2014	2046	2024
q_crossbar16x16	1198	1198	1198	1198	1198	1198	1198	1198	1198

(a)

Buffer size	16	14	12	10	8	6	4	2	1
q_banyan4x8	2015	2004	1936	1866	1875	1796	1756	1673	1644
q_4_cube4x8	2000	1981	1952	1899	1837	1760	1730	1670	1680
q_omega4x8	2000	1981	1952	1899	1837	1760	1730	1670	1680
q_baseline4x8	2008	1972	1953	1915	1880	1794	1763	1682	1653
q_crossbar16x16	1198	1198	1198	1198	1198	1198	1196	1178	1145

(b)

Table 6.28. Number of conflicts.

Figure 6.28. Number of conflicts.

Buffer size	25	24	23	22	21	20	19	18	17
q_banyan4x8	0.004375	0.00625	0.00875	0.01063	0.01125	0.01688	0.02438	0.025	0.03375
q_4_cube4x8	0.000625	0.00188	0.00313	0.005	0.00625	0.00875	0.01188	0.0163	0.02063
q_omega4x8	0.000625	0.00188	0.00313	0.005	0.00625	0.00875	0.01188	0.0163	0.02063
q_baseline4x8	0.003125	0.00375	0.00438	0.00563	0.00813	0.0125	0.015	0.0213	0.01938
q_crossbar16x16	0	0	0	0	0	0	0	0	0

(a)

Buffer size	16	14	12	10	8	6	4	2	1
q_banyan4x8	0.041875	0.055	0.06375	0.08813	0.11563	0.13625	0.18313	0.2625	0.34625
q_4_cube4x8	0.025	0.04125	0.06125	0.07625	0.105	0.13	0.18188	0.2713	0.36125
q_omega4x8	0.025	0.04125	0.06125	0.07625	0.105	0.13	0.18188	0.2713	0.36125
q_baseline4x8	0.03375	0.03563	0.05438	0.07938	0.09938	0.1325	0.17938	0.2432	0.355
q_crossbar16x16	0	0	0	0	0	0.00063	0.00375	0.0206	0.1

(b)

Table 6.29. Loss probability.

Figure 6.29. Loss probability.

	25	24	23	22	21	20	19	18	17	16	14	12	10	8	6	4	2	1
q_banyan4x8	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440
q_4_cube4x8	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440
q_omega4x8	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440
q_baseline4x8	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440	440
q_crossbar16x16	431	431	431	431	431	431	431	431	431	431	431	431	431	431	431	431	431	431

Table 6.30. Minimum cell delay.

Figure 6.30. Minimum cell delay.

Buffer size	25	24	23	22	21	20	19	18	17
q_banyan4x8	6861.19	6852.48	6797.1	6789.58	6775.53	6718.83	6467.89	6513.41	6263.73
q_4_cube4x8	6763.95	6717.71	6715.81	6672.47	6656.31	6612.69	6585.13	6510.35	6337
q_omega4x8	6763.95	6717.71	6715.81	6672.47	6656.31	6612.69	6585.13	6510.35	6337
q_baseline4x8	6728.85	6724.98	6725.6	6716.26	6720.59	6587.45	6433.53	6426.71	6320.78
q_crossbar16x16	1213.55	1213.55	1213.55	1213.55	1213.55	1213.55	1213.55	1213.55	1213.55

(a)

Buffer size	16	14	12	10	8	6	4	2	1
q_banyan4x8	6261.79	5949.42	5370.91	4877.69	4489.61	3532.78	2630.38	1520.17	926.67
q_4_cube4x8	6479.36	6038.47	5872.27	5214.82	4461.44	3519.72	2587.8	1498.12	952.24
q_omega4x8	6479.36	6038.47	5872.27	5214.82	4461.44	3519.72	2587.8	1498.12	952.24
q_baseline4x8	6090.41	5682.27	5566.71	5176.8	4552.36	3603.1	2633.65	1656.85	907.35
q_crossbar16x16	1213.55	1213.55	1213.55	1213.55	1213.55	1203.94	1178.76	1064.82	816.83

(b)

Table 6.31. Average cell delay.

Figure 6.31. Average cell delay.

Buffer size	25	24	23	22	21	20	19	18	17
q_banyan4x8	21888	21156	21651	21651	21226	21757	20281	20538	20431
q_4_cube4x8	22606	22606	22606	23031	23031	23031	22606	22178	18933
q_omega4x8	22606	22606	22606	23031	23031	23031	22606	22178	18933
q_baseline4x8	25060	25060	25485	25485	24208	23026	22519	20963	21035
q_crossbar16x16	7908	7908	7908	7908	7908	7908	7908	7908	7908

(a)

Buffer size	16	14	12	10	8	6	4	2	1
q_banyan4x8	20562	18080	16895	15184	13252	9835	6228	3833	2023
q_4_cube4x8	20956	17998	18020	13886	12302	9338	6343	3616	2182
q_omega4x8	20956	17998	18020	13886	12302	9338	6343	3616	2182
q_baseline4x8	18876	20892	17396	14679	13919	9964	6874	3924	2071
q_crossbar16x16	7908	7908	7908	7908	7908	7482	6630	5080	2956

(b)

Table 6.32. Maximum cell delay.

Figure 6.32. Maximum cell delay.

Buffer size	25	24	23	22	21	20	19	18	17
q_banyan4x8	950.9	953.43	948.5	945.57	938.17	942.67	906.53	904	881.16
q_4_cube4x8	944.1	936.05	934.6	931.98	928.4	920.69	912.6	894.55	900.27
q_omega4x8	944.1	936.05	934.6	931.98	928.4	920.69	912.6	894.55	900.27
q_baseline4x8	963.3	962.75	956.6	954.83	959.32	941.8	939.24	949.48	933.07
q_crossbar16x16	16.1	16.1	16.1	16.1	16.1	16.1	16.1	16.1	16.1

(a)

Buffer size	16	14	12	10	8	6	4	2	1
q_banyan4x8	861.9	834.71	757.8	675.39	610.06	456.49	309.81	146.62	62.87
q_4_cube4x8	898.5	870.09	839.4	740.97	611.97	455.87	307.59	143.81	65.65
q_omega4x8	898.5	870.09	839.4	740.97	611.97	455.87	307.59	143.81	65.65
q_baseline4x8	895.5	804.51	804.6	734.8	629.2	469.45	314.34	147.31	62.23
q_crossbar16x16	16.1	16.1	16.1	16.1	16.1	15.96	15.43	12.86	7.579

(b)

Table 6.33. Average queue size.

Figure 6.33. Average queue size.

	25	24	23	22	21	20	19	18	17
q_banyan4x8	9772	9343	9342	8913	7721	8131	7629	7292	7158
q_4_cube4x8	7576	7538	7147	7099	6670	6680	7107	6242	6250
q_omega4x8	7576	7538	7147	7099	6670	6680	7107	6242	6250
q_baseline4x8	7731	7731	7731	7731	7731	7731	7302	7316	7226
q_crossbar16x16	2338	2338	2338	2338	2338	2338	2338	2338	2338

(a)

	16	14	12	10	8	6	4	2	1
q_banyan4x8	6863	6005	5034	4292	3432	2581	1721	868	454
q_4_cube4x8	5576	5285	5147	4132	3382	2579	1748	965	552
q_omega4x8	5576	5285	5147	4132	3382	2579	1748	965	552
q_baseline4x8	6797	5881	5099	4291	3432	2579	1730	954	512
q_crossbar16x16	2338	2338	2338	2338	2338	2024	1601	881	442

(b)

Table 6.34. Maximum queue size.

Figure 6.34. Maximum queue size.

The most important parameter, evaluated in this simulation, is the cell loss probability. This measure increases as the buffer size is lowered. With the buffer size equal to 25 the loss probability is equal to 0, while with the reduced buffer size down to 1 its value reaches 0.4 for all the networks except crossbar. If buffer size equals to 0, the cell loss probability should theoretically become 1, but the ATM library does not allow reducing buffers to sizes less than 1. In the crossbar the loss probability stays at 0 even with quite short buffers. The delays and the queue sizes dwindle as the cell loss probability grows. The

throughput stays relatively long at the original value because with the cells dropped the simulation time also decreases. Only at very low buffers the throughput goes down. As in the previous simulation step, simulation results of $q_{\omega 4 \times 8}$ and $q_{4 \text{ cube } 4 \times 8}$ are absolutely identical.

6.5 Summary of testing results.

The simulation results in this section suggest that the performance of each particular network is unique with each traffic distribution. However, the evaluation of several networks occasionally gives the same result because of the isomorphism among them that leads to the functional equivalency. If the order of input connections were changed, some other isomorphic networks would also behave similarly. But the problem is that input connections are permanent. The performance of one type of network may be better than that of another type with some particular traffic distribution. The main difficulty in the routing of cells is the case when two cells with the same output address enter the switching fabric. In this situation it is not possible to eliminate the conflict and the waiting time is involved. Various techniques are employed in reducing of the internal conflicts before the final stage, among them are rearrangeability and sorting. Any attempt to create a network that could handle various traffics is costly as far as hardware implementation is concerned. Usually extra stages are added to a network to make it capable of handling changing traffic distributions without conflicts. The better performance is desired, more overhead must be added to the network hardware.

The crossbar switch is considered to have the best and the most stable performance, however it is not very scalable. The number of switch-boxes is in the order of square of output destinations. On the other hand, banyan-type multistage interconnection networks are scalable, but very inefficient in handling traffic with unknown in advance distribution of interarrival time and destination addresses. Simple banyan networks may be designed for predetermined traffic distribution only; otherwise additional hardware overhead is necessary to improve the process of routing.

To compare the results of this work with other results gained analytically and by simulation some graphs are given below. Figures 6.35 and 6.36 give the simulation results of banyan networks with 16 outputs under varying traffic load. The results show that when the

system saturates, the delay grows significantly up to infinity. The technical implementation aspects can make this happen. From the delay graph in Fig. 6.35 it can be concluded that the total delay consists of the delay in the routing network and an additional delay most likely caused by the output multiplexing. The delay in the routing network (graph marked with RN) is bounded by some value as in the ATM library simulation model.

Figure 6.35. Delay and throughput (carried load) of 16-output banyan networks with and without cut-through routing [20].

To expedite the packet routing, the method of bypass queuing is suggested in [20]. Bypass queuing allows some packets bypass the queue when it is congested. As it is shown

Figure 6.36. Delay and throughput (carried load) of 16-output banyan networks with and without effect of bypass queuing [20].

in Fig. 6.36, the delay significantly decreases with this approach. Another way to decrease the delay and increase the throughput is to use cut-through routing as shown in Fig. 6.35. This method was also tested with the ATM library by introducing the internal buffers in the switching elements (see Figure and Table 6.12). In both cases the performance with cut-through routing slightly improves. In the case of ATM library, the throughput was estimated for predetermined traffic patterns with offered load equal to 1, whereas in the other considered cases, the throughput was determined for varying offered load. In the figures from sources [16] and [20], the throughput is normalized to the maximum achievable value. In the ATM library the maximum achievable value for throughput is the number of outputs. The throughput graphs in Figures 6.35 and 6.36 may be compared to those in Figures 6.21 for the ATM library and 6.37 produced by the Markov chain approximation model [16]. The

Figure 6.37. Throughput of a 16-output omega network. Markov chain approximation versus simulation [16].

highest point of the throughput graph in Fig. 6.21 is 0.244 with $\lambda = 1$ for the 256-output baseline8x128 network, if the throughput is normalized to the maximum value of 256. In the other graphs, throughput reaches the 0.7 point, although the curve has a similar shape. Different numbers of conflicts in the compared networks caused by different output addressing may be the reason for the difference of the throughput values. Moreover, the graphs in Fig. 6.35 and 6.36 are for a copy network; in such networks throughput twice exceeds the corresponding value for usual networks. Different size of the networks could also influence the results.

The graph in figure 6.37 is the best to be compared with the ATM library graph because there is no technical overhead involved in both analytical and simulation versions of this graphs.

The delay graphs in Figures 6.35 and 6.36 may be compared to those in Figure 6.23 for the ATM library. If normalized to time slots, the delay in Figure 6.23 reaches the point of 68.22 slots for the banyan network with the arrival rate of 1. In the other graphs, the delay is fixed at 10 time slots, when the routing network delay is considered or bypass queuing is applied. In the other cases it raises to infinity. The delay reaches the highest point in the proposed methods at the arrival rate of 0.5. The difference in the delay between the created model and the models proposed in [16] and [20] may be caused by the different levels of message abstraction: bit level in the ATM libraries and cell level in the other models. Another reason is that the output address distribution pattern may also differ. Nevertheless, the graphs for all the methods have a similar shape, and several traffic and switching fabric parameters may adjust the elevation of a curve.

Another good example of graph comparison would be throughput for the 256-output banyan network shown in Fig. 4.1 versus the corresponding graph in Fig. 6.21 obtained by the simulation with the ATM library. In Fig 4.1 the throughput is not normalized to the maximum value, but rather represented by the absolute value in cells per time slot. The different numbers of conflicts in the two systems may again explain the difference in the results: 170 cells per slot by the analytical method versus 62 cells per slot by the ATM library at $\lambda = 1$.

This section described the results of the simulation of various switching fabric types using the ATM library -- the new discrete time software simulation tool. These results are comparable with the other results obtained analytically and by alternative simulation methods. The next section summarizes this work, gives suggestions regarding the utilization of different types of switching fabric, and proposes the ways to improve the simulation tool functions.

7 Conclusion

7.1 Summary

The outcome of this thesis work is the creation of the ATM library – a software package for the simulation of ATM switch architectures. The library is a set of control driven "software chips" describing the basic elements of an ATM switch. The library is implemented as a discrete time simulation tool simulating synchronous networks. Utilizing this library, it is possible to evaluate a set of basic parameters describing the behaviour of a generic interconnection network employed in an ATM switch. These parameters include cell delay, queue length, cell loss probability, throughput, and the number of conflicts.

The set of tests in this thesis work was run on a series of banyan networks and a crossbar network. In addition, the features of rearrangeability and sorting improving the ATM switching fabric performance were explored in this work. Rearrangeability reduces the number of internal conflicts in a network by selecting the least busy path for a cell. Sorting eliminates the internal conflicts by arranging the cells entering the network into an ordered sequence. While the two features improve network performance, they also introduce additional overhead into the system. As the simulation results showed, simple banyan networks are extremely scalable and performed well with traffic loads known in advance, but they are not adaptable to random traffic distributions. On the other hand, enhanced networks, such as rearrangeable, sorting, crossbar, and many other types discussed in section 3, are very adjustable to various traffic loads but their implementation is complex and costly. The main goal of the simulation is to find a compromise between performance and complexity for a particular topology and a traffic distribution; therefore, the use of complex components must be justified for each specific network design. There is no network design that would be cheap and effective under all conditions, otherwise the theory of switching would not exist.

7.2 Future work

After verifying that the C++ modules successfully accomplished their function of simulating synchronous networks, it will be possible to make the description of the ATM switch hardware in VHDL. Verification of the switch functionality with VHDL will be a step towards the manufacturing of a real ATM switch, which will better meet the requirements imposed by some particular network configuration and a traffic distribution. Another possible direction is enhancing the functionality of the existing C++ classes, thus adding new capabilities to the simulation model. The new capabilities may include testing of buffer management policies and reducing conflicts using dilated links to interconnect the stages. Additional modules can be added to the library as well, expanding the area of application of the ATM library. The library can possibly be applied to a complete ATM switch including input and output modules. One step further, after the verification of the whole ATM switch, may be the simulation of a part of a network or a whole network consisting of several ATM switches. The simulation of a network would be valuable to verify effectiveness of the data transfer among several actual or virtual locations. Finally, the library can be adjusted for the simulation of multiprocessor interconnection networks.

Bibliography

1. Gunter Haring, Gabriele Kotsis, "Computer Performance Evaluation. Modeling Techniques and Tools." 7th International Conference Vienna, Austria, May 1994.
2. Josepmaria Malgosa-Sanahuja, Jordi Castells-Cusculola, Joan Garcia-Haro, "An ATM Switch Simulation Tool Based on the C++ Object Oriented Programming Language." 1997 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing, Vol. 2.
3. Sofiene Tahar, Xiaoyu Song, et. al. "Modeling and Formal Verification of the Fairisle ATM Switch Fabric Using MDG's." IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 18, No. 7, July 1999.
4. Daniel Sobirk, Johan M Karlsson, "ATM Switching Structures -- A Performance Comparison." Dept. of Communication Systems. Lund Institute of Technology. Sweden. <http://www.tts.lth.se/Personal/daniels/papers/nts12.abstract.html>
5. Andrew Tanenbaum, "Computer Networks." Prentice Hall, Inc., New Jersey, 1996.
6. Achille Pattavina, "Switching theory: architecture and performance in broadband ATM networks." Chichester, West Sussex, England; New York: J. Wiley, c1998
7. Jim Lane, "Asynchronous Transfer Mode: Bandwidth for the Future." Telco Systems, Inc. Norwood, Massachusetts, 1992.
8. Adam Lange-Pearson, "Self-Similarity in a Multi-Stage Queuing ATM Switch Fabric." Graduate thesis. Rochester Institute of Technology, May 1999.
9. Darin Murphy, "The design and modeling of input and output modules for an ATM network switch." Graduate thesis. Rochester Institute of Technology, October 1997.

10. James Heliotis, "ARCH library." Software tool for the simulation of computer architectures. Rochester Institute of Technology, January 1994.
11. "A Framework for Virtual Channel onto Virtual Path Multiplexing in ATM-ABR," ATM Forum/99-0403, July 1999. <http://www.cis.ohio-state.edu/~jain/index.html>
12. Raj Jain, "A Survey of ATM Switching Techniques". <http://www.cis.ohio-state.edu/~jain/index..html>
13. "ATM Switch Network Configuration Manual", <http://www.marconi.com/>
14. "A Survey of ATM Switching Techniques" http://www.cis.ohio-state.edu/~jain/cis788-95/atm_switching/index.html (21 of 25) [2/7/2000 11:06:16 AM].
15. Patrick G. Sobalvarro. "Analytical Modeling of Multistage, Multipath Networks". IEEE Transactions on Parallel and Distributed Systems. Vol. 7, №. 10, October 1996.
16. Arif Merchant. "A Markov chain Approximation for the Analysis of Banyan Networks". ACM Sigmetrics and performance Int'l. Conference on Measurement and Modeling of Computer Systems. Volume 20, number 1, June, 1992. Newport, Rhode Island, USA.
17. T. Lin, L. Kleinrock. "Performance Analysis of Finite-Buffered Multistage Interconnection Networks with a General Traffic Pattern". ACM Sigmetrics and performance Int'l. Conference on Measurement and Modeling of Computer Systems. Volume 20, number 1, June, 1992. Newport, Rhode Island, USA.
18. Abhijit K. Choudhury, Ellen L. Hahne. "A new Buffer Management Scheme for Hierarchical Shared Memory Switches". IEEE/ACM Transactions on Networking. Vol. 5, number 5, October 1997.

19. Mohamed Abdelaziz, Ioannis Stavrakakis. "Some Optimal Traffic Regulation Schemes for ATM Networks: A Markov Decision Approach", IEEE/ACM Transactions on Networking, Vol. 2, number 5, October, 1994.
20. Thomas G. Robertazzi. "Performance Evaluation of High Speed Switching Fabrics and Networks". Department of Electrical Engineering. State University of New York at Stony Brook. 1992.
21. Hakyong Kim, Changwan Oh, Kiseon Kim. "A High-Speed ATM Switch Architecture Using Random Access Input Buffers and Multi-Cell-Time Arbitration". GLOBECOM'97 -- IEEE Global Telecommunications Conference.
22. Averill M. Law, W. David Kelton. "Simulation Modeling and Analysis". McGraw-Hill, Inc.
23. Kenneth Reek. "Course materials for the Data Communications and Networks class". Rochester Institute of Technology. May, 1999.

APPENDIX A

This appendix contains the utility programmes facilitating the construction of network simulation models based on the ATM library.

```
/*-----e l e m e n t . C-----*/

#include<SwElement.H>
#include<InSocket.H>
#include<OutSocket.H>
#include<Wire.H>
#include<Clock.H>
#include<Manager.H>

int main()
{
    //BaseClass::debug =
        //BaseClass::create;// | BaseClass::trace;

    Manager::output_allowed = 1;

    SwElement aa("SE", 0, 0, 2, 0, 0, 0);
    SwElement ab("SE", 1, 0, 2, 0, 0, 0);
    SwElement ac("SE", 2, 0, 2, 0, 0, 0);
    Wire wr0("WR", 0, 0, 2);
    Wire wr1("WR", 1, 1, 2);
    Wire wr2("WR", 2, 1, 2);
    Wire wr4("WR", 3, 2, 2);
    InSocket soc("SI", 0, 2, 0);
    OutSocket sout("SO", 0, 2, 1);

    soc.connectsTo( wr0.IN(), 0);
    aa.connectsTo( wr1.IN(), 0);
    aa.connectsTo( wr0.OUT(), 1000);
    ab.connectsTo( wr1.OUT(), 1000);
    ab.connectsTo( wr2.IN(), 0);
    ac.connectsTo( wr2.OUT(), 1000);
    ac.connectsTo( wr4.IN(), 0);
    sout.connectsTo( wr4.OUT(), 1000);

    while( !Manager::stop )
    {
        wr0.IN().pull();
        aa.latch();
        wr1.IN().pull();
        ab.latch();
        wr2.IN().pull();
        ac.latch();
        wr4.IN().pull();
        sout.latch();
        Clock::tick();
    }
    return( 0 );
}
```

```

/*-----R u n.C-----*/
// Driver of an ATM switch

#include <Switch.H>
void usage();

int main( int argc, char** argv )
{
    //BaseClass::debug = //BaseClass::trace;
    //BaseClass::create;// | BaseClass::trace;

    char** temp = argv;
    int count = 1;
    int traffic_argc = 0;
    int buffer_size = 0;
    temp++;

    if( argc == 1 )
    {
        usage();
        exit( 0 );
    }

    while( *++temp )
    {
        count++;
        if( !strcmp( *temp, "-c" ) ) // special case of routing
            // table for a crossbar switch
            Manager::crossbar_option = 1;
        else if( !strcmp( *temp, "-o" ) )
            Manager::output_allowed = 1; // write output to files

        else if( !strcmp( *temp, "-r" ) )
            Manager::no_rearrangeability = 0; // rearrangeability allowed
            // whaen there are no internal
            // buffers

        else if( !strcmp( *temp, "-tr" ) )
            traffic_argc = count;

        else if ( !strcmp( *temp, "-buff" ) )
        {
            count++;
            if( count >= argc )
            {
                cout << "Error: Must give buffer size." << endl;
                exit( 1 );
            }
            buffer_size = strtod( *++temp, NULL );
        }
    }

    Switch sw( argv[1], traffic_argc, argv, buffer_size );
    sw.work();

    return 0;
}

```

```

void usage()
{
    cout << endl;
    cout << "Usage:" << endl << "< run or run1 > "
        << "< switch setup file name without extension .des > "
        << "[ options ]" << endl;
    cout << endl << "For programme run setup files are in the directory INIT"
        << endl << "For programme run1 setup files are in the directory INIT1"
        << endl << endl;
    cout << "Options:" << endl;
    cout << "-o write the simulation traces in the output files in\n"
        << "  the directory OUTPUTS" << endl
        << "-r force rearrangeability; cannot use this option with\n"
        << "  internal buffers available\n"
        << "-c use simplified routing table for a crossbar switch;\n"
        << "  this option is needed for large switches, since the\n"
        << "  routing table of usual form is too large in this case\n"
        << "-b buff override the bubble size in a buffered network\n"
        << "-tr pass the argument line that follows this option to the\n"
        << "  traffic generator; nothing else can follow the argument\n"
        << "  line\n\n";
    cout << "Be sure to check the simulation results in OUTPUTS/postscript.txt"
        << endl << endl;
}

```

```

/*-----S w i t c h . C-----*/

#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<string.h>
#include<math.h>
#include<Switch.H>

Switch::Switch( char* descriptor, int traffic_argc, char** traffic_argv, int buffer_size )
{
    strcpy( Manager::network_type, descriptor );

    int i, j;
    char SetUpFile[ 20 ];
    SetUpFile[ 0 ] = '\0';

    // Open the setup file

    const char* DIR = "INIT/";
    const char* suffix = ".des";
    strcat( SetUpFile, DIR );
    strcat( SetUpFile, descriptor );
    strcat( SetUpFile, suffix );
    ifstream fin( SetUpFile, ios::in );

    if( !fin )
    {
        cout << "Description file could not be opened.\n";
        exit( 1 );
    }

    fin >> max_rows >> stages >> max_outs;
    check_stream( fin, "I" );

    max_outs = ( int )( log( max_outs ) / log( 2 ) + 1 );

/*-----INITIALIZING OF THE SWITCH ELEMENTS-----*/

    if( !( se = ( SwElement** ) calloc( max_rows*stages, sizeof( SwElement* ) ) ) )
    {
        cout << "Allocation failed in main 1.\n";
        exit( 1 );
    }

    for( i = 0; i < stages; i++ )
    {
        fin >> rows_in_this_stage;
        check_stream( fin, "II" );

        for( j = 0; j < rows_in_this_stage; j++ )
        {
            if( !se[ i * max_rows + j ] )
                se[ i * max_rows + j ] = new SwElement( "SE", i, j, max_outs, 0, 0, 0 );
            else
            {
                cout << "\nMultiple declarations of SwElements in Switch." << endl;
            }
        }
    }
}

```

```

        exit( 1 );
    }
    if( !se[ i * max_rows + j ] )
    {
        cout << "Allocation of an SwElement failed in Switch.\n";
        exit( 1 );
    }
}

fin >> max_sw_pins >> insockets >> outsockets;
check_stream( fin, "III" );

if( !( ins = ( InSocket** )calloc( insockets, sizeof( InSocket* ) ) ) )
{
    cout << "Allocation failed in main 2.\n";
    exit( 1 );
}

if( !( tg = ( TGen** )calloc( insockets, sizeof( TGen* ) ) ) )
{
    cout << "Allocation failed in main 2a.\n";
    exit( 1 );
}

if( !( outs = ( OutSocket** )calloc( outsockets, sizeof( OutSocket* ) ) ) )
{
    cout << "Allocation failed in main 3.\n";
    exit( 1 );
}

fin >> traffic_generator_present;
check_stream( fin, " in insockets loop 0." );

if( traffic_generator_present )
{
    for( j = 0; j < 255; j++ )
        tgen_buffer[ j ] = '\0';
    fin.getline( tgen_buffer, 255 );
    if( traffic_argc )
    {
        for( j = 0; j < 255; j++ )
            tgen_buffer[ j ] = '\0';
        traffic_argv += traffic_argc;
        while( *++traffic_argv )
        {
            strcat( tgen_buffer, " " );
            strcat( tgen_buffer, *traffic_argv );
        }
    }
    tm = new TManager( tgen_buffer, outsockets );
}

for( i = 0; i < insockets; i++ )
{
    fin >> input_option;
    check_stream( fin, " in insockets loop 1." );
}

```

```

        if( !ins[ i ] )
            ins[ i ] = new InSocket( "IS", i, max_outs, input_option );
        else
        {
            cout << "\nMultiple declarations of "
                  << "InSockets in Switch." << endl;
            exit( 1 );
        }
        if( !ins[ i ] )
        {
            cout << "Allocation of an InSocket failed in Switch.\n";
            exit( 1 );
        }
        if( input_option )
        {
            if( !tg[ i ] )
                tg[ i ] = new TGen( tm, i );
            else
            {
                cout << "\nMultiple declarations of "
                      << "TGen in Switch." << endl;
                exit( 1 );
            }
            if( !tg[ i ] )
            {
                cout << "Allocation of a TGen failed in Switch.\n";
                exit( 1 );
            }
            tg[ i ]->plugsInto( ins[ i ] );
        }
    }

    for( i = 0; i < outsockets; i++ )
    {
        if( !outs[ i ] )
            outs[ i ] = new OutSocket( "OS", i, max_outs, insockets );
        else
        {
            cout << "\nMultiple declarations of "
                  << "OutSockets in Switch." << endl;
            exit( 1 );
        }
        if( !outs[ i ] )
        {
            cout << "Allocation of an OutSocket failed in Switch.\n";
            exit( 1 );
        }
    }

    fin >> q_max_rows >> q_stages;
    check_stream( fin, "IV" );

    if( !( q = ( Queue** ) calloc( q_stages * q_max_rows, sizeof( Queue* ) ) ) )
    {
        cout << "Allocation failed in main 4.\n";
        exit( 1 );
    }

```

```

for(i = 0; i < q_stages; i++)
{
    fin >> queues_in_stage;
    check_stream(fin, "V");

    for(j = 0; j < queues_in_stage; j++)
    {
        fin >> q_size;
        check_stream(fin, "VI");

        if(i < q_stages - 1 )
        {
            if(!q[i * q_max_rows + j])
                q[i * q_max_rows + j] = new Queue("Q", i, j, max_outs,
                q_size, stages, 'i', 0);
            else
            {
                cout << "\nMultiple declarations of Queues in Switch."
                << endl;
                exit(1);
            }
            if(!q[i * q_max_rows + j])
            {
                cout << "Allocation of a Queue failed in Switch. I\n";
                exit(1);
            }
        }
        else
        {
            fin >> suppressor;
            check_stream(fin, "VII");

            if(!q[i * q_max_rows + j])
                q[i * q_max_rows + j] = new Queue("Q", i, j, max_outs,
                q_size, stages, 'o', suppressor);
            else
            {
                cout << "\nMultiple declarations of Queues in Switch."
                << endl;
                exit(1);
            }
            if(!q[i * q_max_rows + j])
            {
                cout << "Allocation of a Queue failed in Switch. II\n";
                exit(1);
            }
        }
    }
}

max_wires_in_acol = max_sw_pins * max_rows;
if(!(wr = (Wire**)calloc((stages + q_stages + 1) * max_wires_in_acol, sizeof(Wire*))))
{
    cout << "Allocation failed in main 5.\n";
    exit(1);
}

```



```
/*-----D E FAULT ASSIGNMENTS-----*/
```

```

for(i = 0; i <= stages; i++)
{
    con_id = 0;

    fin >> max_pins_in_col;
    check_stream( fin, "VIII" );

    wires_in_this_col = max_pins_in_col*max_rows;

    for(j = 0; j < wires_in_this_col; j++)
    {
        if(i == 0)
        {
            if(j < insockets)
            {
                if(!wr[j])
                    wr[j] = new Wire( "WR", i, j, max_outs );
                else
                {
                    cout << "\nMultiple declarations of "
                        << "Wires in Switch." << endl;
                    exit( 1 );
                }
                if(!wr[j])
                {
                    cout << "Allocation of a Wire failed in Switch. I\n";
                    exit( 1 );
                }
                ins[j]->connectsTo( wr[j]->IN(), 0 );
            }
        }
        else
        {
            if( se[ (i-1)*max_rows + j/max_pins_in_col ])
            {
                if(!wr[i*max_wires_in_acol + j])
                    wr[i*max_wires_in_acol + j] =
                    new Wire( "WR", i, j, max_outs );
                else
                {
                    cout << "\nMultiple declarations of "
                        << "Wires in Switch." << endl;
                    exit( 1 );
                }
                if(!wr[i*max_wires_in_acol + j])
                {
                    cout << "Allocation of a Wire failed in Switch. II\n";
                    exit( 1 );
                }

                se[ (i-1)*max_rows + j/max_pins_in_col ]->
                connectsTo( wr[i*max_wires_in_acol + j]->IN(), con_id );
                con_id++;
                if( con_id == max_pins_in_col )
                    con_id = 0;
            }
        }
    }
}

```

```

    }
    }
}

for(i = 0; i < q_stages; i++)
{
    con_id = 0;

    fin >> max_pins_in_col;
    check_stream( fin, "IX" );

    wires_in_this_col = max_pins_in_col*q_max_rows;

    for(j = 0; j < wires_in_this_col; j++)
    {
        if( q[ i*q_max_rows + j/max_pins_in_col ] )
        {
            if( !wr[ ( i + stages + 1)*max_wires_in_acol + j ] )
                wr[ ( i + stages + 1)*max_wires_in_acol + j ] =
                    new Wire( "WR", i + stages + 1, j, max_outs );
            else
            {
                cout << "\nMultiple declarations of "
                    << "Wires in Switch." << endl;
                exit( 1 );
            }
            if( !wr[ ( i + stages + 1)*max_wires_in_acol + j ] )
            {
                cout << "Allocation of a Wire failed in Switch. III\n";
                exit( 1 );
            }

            q[ i*q_max_rows + j/max_pins_in_col ]->
            connectsTo( wr[ ( i + stages + 1)*max_wires_in_acol
                        + j ]->IN(), con_id );

            con_id++;
            if( con_id == max_pins_in_col )
                con_id = 0;
        }
    }
}

/*-----ASSIGNMENTS WITH INITIAL SETTINGS TAKEN FROM FILE-----*/

int row, col, _row, _col, num_con, stop = 0;

for(i = 0; i < stages * max_rows; i++ )
{
    stop = searchFor( 's', fin );
    if( stop )
        break;
    check_stream( fin, "X" );

    fin >> col >> row;
    check_stream( fin, "XI" );
}

```

```

searchFor( 'i', fin );
check_stream( fin, "XII" );

fin >> num_con;
check_stream( fin, "XIII" );

con_id = 1000;
for( ; num_con; num_con-- )
{
    fin >> _col >> _row;
    check_stream( fin, "XIV" );

    if( se[ col*_max_rows + row ] )
        se[ col*_max_rows + row ]->
            connectsTo( wr[ _col*_max_wires_in_acol + _row ]
                ->OUT(), con_id++ );
    else
    {
        cout << "Initial settings out of range 1.\n";
        exit( 1 );
    }
}

for( i = 0; i < outsockets; i++ )
{
    stop = searchFor( 'o', fin );
    if( stop )
        break;
    check_stream( fin, "XV" );

    fin >> row;
    check_stream( fin, "XVI" );

    fin >> _col >> _row;
    check_stream( fin, "XVII" );

    if( outs[ row ] )
        outs[ row ]->
            connectsTo( wr[ _col*_max_wires_in_acol + _row ]->OUT(), 1000 );
    else
    {
        cout << "Initial settings out of range 2.\n";
        exit( 1 );
    }
}

for( i = 0; i < q_stages * q_max_rows; i++ )
{
    int row_dup, col_dup, _row_dup, _col_dup, num_con_dup;

    stop = searchFor( 'q', fin );
    if( stop )
        break;
    check_stream( fin, "XVIII" );

    fin >> col_dup >> row_dup;

```

```

        check_stream( fin, "XIX" );

        searchFor( 'I', fin );
        check_stream( fin, "XX" );

        fin >> num_con_dup;
        check_stream( fin, "XXI" );

        con_id = 1000;
        for( ; num_con_dup; num_con_dup-- )
        {
            fin >> _col_dup >> _row_dup;
            check_stream( fin, "XXII" );

            if( q[ col_dup*_q_max_rows + row_dup ]
                q[ col_dup*_q_max_rows + row_dup ]->
                connectsTo( wr[ _col_dup*_max_wires_in_acol + _row_dup ]
                    ->OUT(), con_id++ );

            else
            {
                cout << "Initial settings out of range 3.\n";
                exit( 1 );
            }
        }
    }
}

/*-----*/

/*-----D S E T R U C T I O N  O F  T H E  S W I T C H-----*/

Switch::~Switch()
{
    sed = se;
    insd = ins;
    outsd = outs;
    qd = q;
    wrd = wr;
    tgd = tg;
    int i;
    for( i = 0; i < max_rows*stages; i++ )
    {
        if( *sed )
            delete( *sed );
        sed++;
    }
    for( i = 0; i < q_max_rows*q_stages; i++ )
    {
        if( *qd )
            delete( *qd );
        qd++;
    }
    for( i = 0; i < ( stages + q_stages + 1 )*_max_wires_in_acol; i++ )
    {
        if( *wrd )
            delete( *wrd );
        wrd++;
    }
}

```

```

for(i = 0; i < insockets; i++)
{
    if( *insd )
        delete( *insd );
    insd++;
}
for(i = 0; i < insockets; i++)
{
    if( *tgd )
        delete( *tgd );
    tgd++;
}
for(i = 0; i < outsockets; i++)
{
    if( *outsd )
        delete( *outsd );
    outsd++;
}
if( traffic_generator_present )
    delete( tm );

free( se );
free( ins );
free( wr );
free( outs );
free( q );
free( tg );
}
/*-----*/

/*-----EXECUTION OF THE SWITCH-----*/
void Switch::work()
{
    int i;
    while( !Manager::stop )
    {
        for(i = 0; i <= stages + q_stages; i++)
        {
            for( int j = 0; j < max_wires_in_acol; j++)
            {
                if( wr[ i*max_wires_in_acol + j ] )
                {
                    ( wr[ i*max_wires_in_acol + j ]->IN0 ).pull();
                }
                if( i == stages )
                {
                    if( j < outsockets )
                    {
                        outs[ j ]->latch();
                    }
                }
                else
                {
                    if( i < stages )
                    {
                        if( j < max_rows )
                        {

```

```

                                if( se[ i*max_rows + j ])
                                {
                                    se[ i*max_rows + j ]->latch();
                                }
                            }
                        }
                    }
                }
            }
        }
    }
    Clock::tick();
}

/*-----*/
/*-----*/
void Switch::work()
{
    int i;
    while( !Manager::stop )
    {
        sed = se;
        insd = ins;
        outsd = outs;
        qd = q;
        wrd = wr;

        for(i = 0; i < ( stages + q_stages + 1 ) * max_wires_in_acol; i++)
        {
            if( *wrd )
                ( (*wrd)->IN() ).pull();
            wrd++;
        }
        for(i = 0; i < max_rows*stages; i++)
        {
            if( *sed )
                ( *sed )->latch();
            sed++;
        }
        for(i = 0; i < q_max_rows*q_stages; i++)
        {
            if( *qd )
                ( *qd )->latch();
            qd++;
        }
        for(i = 0; i < outsockets; i++)
        {
            if( *outsd )
                ( *outsd )->latch();
        }
    }
}

```

```

        outsd++;
    }
    Clock::tick();
}

/*-----*/

int Switch::searchFor( char c, ifstream& fin )
{
    char temp;
    do
    {
        temp = fin.get();
        if( temp == c )
            return 0;
        if( temp > 64 && temp < 123 )
        {
            fin.putback( temp );
            return 1;
        }
    } while( fin.eof() )
    return 1;
} while( temp != c );
return 0; // To satisfy compiler
}

void Switch::check_stream( ifstream& fin, const char* c )
{
    if( fin.fail() )
    {
        cout << "Data in the descriptor file corrupted." << c << endl;
        exit( 1 );
    }
}

```

```

/*-----S witch1.C-----*/

#include<stdlib.h>
#include<iostream.h>
#include<fstream.h>
#include<string.h>
#include<math.h>
#include<Switch.H>
#include<Shuffle.H>

Switch::Switch( char* descriptor, int traffic_argc, char** traffic_argv, int buffer_size )
{
    strcpy( Manager::network_type, descriptor );

    int i, j, k;
    char SetUpFile[ 20 ];
    SetUpFile[ 0 ] = '\0';

    // Open the setup file

    const char* DIR = "INIT1/";
    const char* suffix = ".des";
    strcat( SetUpFile, DIR );
    strcat( SetUpFile, descriptor );
    strcat( SetUpFile, suffix );
    ifstream fin( SetUpFile, ios::in );

    if( !fin )
    {
        cout << "Description file could not be opened.\n";
        exit( 1 );
    }

    fin >> max_rows >> sw_stages >> sorting_stages >> max_outs;
    check_stream( fin, "I" );

    stages = sw_stages + sorting_stages;
    max_outs = ( int )( log( max_outs ) / log( 2 ) + 1 );

    /*-----INITIALIZING OF THE SWITCH ELEMENTS-----*/

    if( !( se = ( SwElement** )calloc( max_rows*stages, sizeof( SwElement* ) ) ) )
    {
        cout << "Allocation failed in main 1.\n";
        exit( 1 );
    }

    fin >> uniform_q_size_for_sw;
    check_stream( fin, "I a" );

    if( uniform_q_size_for_sw )
    {
        fin >> q_size;
        if( buffer_size )
            q_size = buffer_size;
    }
}

```



```

for( i = 0; i < stages; i++ )
{
    fin >> rows_in_this_stage;
    check_stream( fin, "II" );

    for( j = 0; j < rows_in_this_stage; j++ )
    {
        if( !uniform_q_size_for_sw )
        {
            fin >> q_size;
            if( buffer_size )
                q_size = buffer_size;
        }
        fin >> sort_option;
        check_stream( fin, "II a" );

        if( !se[ i * max_rows + j ] )
            se[ i * max_rows + j ] = new SwElement( "SE", i, j, max_outs,
                sw_stages, q_size, sort_option );
        else
        {
            cout << "\nMultiple declarations of SwElements in Switch."
                << endl;
            exit( 1 );
        }
        if( !se[ i * max_rows + j ] )
        {
            cout << "Allocation of an SwElement failed in Switch.\n";
            exit( 1 );
        }
    }
}

fin >> max_sw_pins >> insockets >> outsockets;
check_stream( fin, "III" );

if( !( ins = ( InSocket** )calloc( insockets, sizeof(InSocket*) ) ) )
{
    cout << "Allocation failed in main 2.\n";
    exit( 1 );
}

if( !( tg = ( TGen** )calloc( insockets, sizeof(TGen*) ) ) )
{
    cout << "Allocation failed in main 2a.\n";
    exit( 1 );
}

if( !( outs = ( OutSocket** )calloc( outsockets, sizeof(OutSocket*) ) ) )
{
    cout << "Allocation failed in main 3.\n";
    exit( 1 );
}

fin >> traffic_generator_present;
check_stream( fin, " in insockets loop 0." );

```

```

if( traffic_generator_present )
{
    for( j = 0; j < 255; j++ )
    {
        tgen_buffer[ j ] = '\0';
        fin.getline( tgen_buffer, 255 );
        if( traffic_argc )
        {
            for( j = 0; j < 255; j++ )
            {
                tgen_buffer[ j ] = '\0';
                traffic_argv += traffic_argc;
                while( *++traffic_argv )
                {
                    strcat( tgen_buffer, " " );
                    strcat( tgen_buffer, *traffic_argv );
                }
            }
            tm = new TManager( tgen_buffer, outsockets );
        }
    }

    for( i = 0; i < insockets; i++ )
    {
        fin >> input_option;
        check_stream( fin, " in insockets loop 1." );
        if( !ins[ i ] )
            ins[ i ] = new InSocket( "IS", i, max_outs, input_option );
        else
        {
            cout << "\nMultiple declarations of "
                  << "InSockets in Switch." << endl;
            exit( 1 );
        }
        if( !ins[ i ] )
        {
            cout << "Allocation of an InSocket failed in Switch.\n";
            exit( 1 );
        }
        if( input_option )
        {
            if( !tg[ i ] )
                tg[ i ] = new TGen( tm, i );
            else
            {
                cout << "\nMultiple declarations of "
                      << "TGens in Switch." << endl;
                exit( 1 );
            }
            if( !tg[ i ] )
            {
                cout << "Allocation of a TGen failed in Switch.\n";
                exit( 1 );
            }
            tg[ i ]->plugsInto( ins[ i ] );
        }
    }
}

for( i = 0; i < outsockets; i++ )
{

```

```

        if(!outs[ i ])
            outs[ i ] = new OutSocket( "OS", i, max_outs, insockets );
        else
        {
            cout << "\nMultiple declarations of "
                << "OutSockets in Switch." << endl;
            exit( 1 );
        }
        if(!outs[ i ])
        {
            cout << "Allocation of an OutSocket failed in Switch.\n";
            exit( 1 );
        }
    }

    fin >> q_max_rows >> q_stages;
    check_stream( fin, "IV" );

    if(!( q = ( Queue** )calloc( q_stages*q_max_rows, sizeof(Queue*) ) ) )
    {
        cout << "Allocation failed in main 4.\n";
        exit( 1 );
    }

    for( i = 0; i < q_stages; i++ )
    {
        fin >> queues_in_stage;
        check_stream( fin, "V" );

        for( j = 0; j < queues_in_stage; j++ )
        {
            fin >> q_size;
            check_stream( fin, "VI" );

            if( i < q_stages - 1 )
            {
                if(!q[ i * q_max_rows + j ])
                    q[ i * q_max_rows + j ] = new Queue
                        ( "Q", i, j, max_outs, q_size, sw_stages, 'i', 0 );
                else
                {
                    cout << "\nMultiple declarations of Queues in Switch."
                        << endl;
                    exit( 1 );
                }
                if(!q[ i * q_max_rows + j ])
                {
                    cout << "Allocation of a Queue failed in Switch. I\n";
                    exit( 1 );
                }
            }
            else
            {
                fin >> suppressor;
                check_stream( fin, "VII" );

                if(!q[ i * q_max_rows + j ])

```

```

        q[ i * q_max_rows + j ] = new Queue
        ( "Q", i, j, max_outs, q_size, sw_stages, 'o', suppressor );
    else
    {
        cout << "\nMultiple declarations of Queues in Switch."
        << endl;
        exit( 1 );
    }
    if( !q[ i * q_max_rows + j ] )
    {
        cout << "Allocation of a Queue failed in Switch. II\n";
        exit( 1 );
    }
}
}

max_wires_in_acol = max_sw_pins*max_rows;

if( !( wr = ( Wire** )calloc( ( stages + q_stages + 1 ) * max_wires_in_acol, sizeof(Wire*) ) ) )
{
    cout << "Allocation failed in main 5.\n";
    exit( 1 );
}

/*-----D EFAULT ASSIGNMENTS-----*/

for( i = 0; i <= stages; i++ )
{
    con_id = 0;

    fin >> max_pins_in_col;
    check_stream( fin, "VIII" );

    wires_in_this_col = max_pins_in_col*max_rows;

    for( j = 0; j < wires_in_this_col; j++ )
    {
        if( i == 0 )
        {
            if( j < insockets )
            {
                if( !wr[ j ] )
                    wr[ j ] = new Wire( "WR", i, j, max_outs );
                else
                {
                    cout << "\nMultiple declarations of "
                    << "Wires in Switch." << endl;
                    exit( 1 );
                }
            }
            if( !wr[ j ] )
            {
                cout << "Allocation of a Wire failed in Switch. I\n";
                exit( 1 );
            }
            ins[ j ]->connectsTo( wr[ j ]->IN(), 0 );
        }
    }
}

```

```

    }
    else
    {
        if( se[ (i-1)*max_rows + j/max_pins_in_col ] )
        {
            if( !wr[ i*max_wires_in_acol + j ] )
                wr[ i*max_wires_in_acol + j ] =
                    new Wire( "WR", i, j, max_outs );
            else
            {
                cout << "\nMultiple declarations of "
                    << "Wires in Switch." << endl;
                exit( 1 );
            }
            if( !wr[ i*max_wires_in_acol + j ] )
            {
                cout << "Allocation of a Wire failed in Switch. II\n";
                exit( 1 );
            }

            se[ (i-1)*max_rows + j/max_pins_in_col ]->
                connectsTo( wr[ i*max_wires_in_acol + j ]->IN(), con_id );
            con_id++;
            if( con_id == max_pins_in_col )
                con_id = 0;
        }
    }
}

for( i = 0; i < q_stages; i++ )
{
    con_id = 0;

    fin >> max_pins_in_col;
    check_stream( fin, "IX" );

    wires_in_this_col = max_pins_in_col*q_max_rows;

    for( j = 0; j < wires_in_this_col; j++ )
    {
        if( q[ i*q_max_rows + j/max_pins_in_col ] )
        {
            if( !wr[ (i + stages + 1)*max_wires_in_acol + j ] )
                wr[ (i + stages + 1)*max_wires_in_acol + j ] =
                    new Wire( "WR", i + stages + 1, j, max_outs );
            else
            {
                cout << "\nMultiple declarations of "
                    << "Wires in Switch." << endl;
                exit( 1 );
            }
            if( !wr[ (i + stages + 1)*max_wires_in_acol + j ] )
            {
                cout << "Allocation of a Wire failed in Switch. III\n";
                exit( 1 );
            }
        }
    }
}

```

```

        q[ i*q_max_rows + j/max_pins_in_col ]->
            connectsTo( wr[ (i + stages + 1)*max_wires_in_acol + j ]->
                IN(), con_id );
        con_id++;
        if( con_id == max_pins_in_col )
            con_id = 0;
    }
}

/*-----ASSIGNMENTS WITH INITIAL SETTINGS TAKEN FROM FILE-----*/

int this_stage, this_row, contig_stage, contig_row;
int clusters, in_cluster, pattern, num_con, type, stop = 0;

searchFor( 'C', fin );
fin >> type;
check_stream( fin, "X" );

switch( type )
{
    case 1: // Multistage Banians
    {
        for( i = 0; i < stages; i++ )
        {
            stop = searchFor( 'S', fin );
            if( stop )
                break;
            check_stream( fin, "XI" );

            fin >> this_stage >> contig_stage;
            check_stream( fin, "XII" );

            fin >> clusters >> in_cluster;
            check_stream( fin, "XIII" );

            fin >> pattern >> num_con;
            check_stream( fin, "XIV" );

            con_id = 1000;
            for( j = 0; j < clusters; j++ )
            {
                for( k = 0; k < in_cluster; k++ )
                {
                    this_row = j * in_cluster + k;
                    contig_row =
                        shuffle( in_cluster, j, k, pattern );

                    if( se[ this_stage*max_rows + this_row/num_con ] )
                    {
                        se[ this_stage*max_rows +
                            this_row/num_con ]->
                            connectsTo( wr[
                                contig_stage*max_wires_in_acol
                                + contig_row ]->OUT(), con_id++ );
                        if( con_id == 1000 + num_con )
                            con_id = 1000;
                    }
                }
            }
        }
    }
}

```

```

        }
        else
        {
            cout << "Initial settings out of range 1.\n";
            exit( 1);
        }
    }
}

searchFor( 'o', fin );
fin >> contig_stage;
check_stream( fin, "XV" );

for( i = 0; i < outsockets; i++ )
{
    if( outs[ i ] )
        outs[ i ]->connectsTo( wr[ contig_stage*
            max_wires_in_acol + i ]->OUT(), 1000 );
    else
    {
        cout << "Initial settings out of range 2.\n";
        exit( 1);
    }
}

for( i = 0; i < q_stages; i++ )
{
    stop = searchFor( 'Q', fin );
    if( stop )
        break;
    check_stream( fin, "XVI" );

    fin >> this_stage >> contig_stage;
    check_stream( fin, "XVII" );

    fin >> clusters >> in_cluster;
    check_stream( fin, "XVIII" );

    for( j = 0; j < clusters; j++ )
    {
        con_id = 1000;
        for( k = 0; k < in_cluster; k++ )
        {
            this_row = j * in_cluster + k;
            if( q[ this_stage*q_max_rows + this_row/in_cluster ] )
                q[ this_stage*q_max_rows +
                    this_row/in_cluster ]->
                    connectsTo( wr[
                        contig_stage*max_wires_in_acol
                        + this_row ]->OUT(), con_id++ );
            else
            {
                cout << "Initial settings out of range 3.\n";
                exit( 1);
            }
        }
    }
}

```

```

    }
}
break;
case 2: // Crossbars
{
    searchFor('F', fin); // This is only for the first stage
    check_stream(fin, "XIX");

    fin >> contig_stage;
    check_stream(fin, "XX");

    for(i = 0; i < max_rows; i++)
    {
        if(se[i])
        {
            se[i]->connectsTo(wr[contig_stage*max_wires_in_acol + i]
->OUT(), 1000);
            if(i != max_rows - 1)
            {
                se[i]->
connectsTo(wr[max_wires_in_acol + 2*(i + 1)]
->OUT(), 1001);
            }
        }
        else
        {
            cout << "Initial settings out of range 4.\n";
            exit(1);
        }
    }

    for(i = 1; i < stages; i++)
    {
        stop = searchFor('S', fin);
        if(stop)
            break;
        check_stream(fin, "XXI");

        fin >> this_stage >> contig_stage;
        check_stream(fin, "XXII");

        for(j = 0; j < max_rows; j++)
        {
            if(se[this_stage*max_rows + j])
            {
                se[this_stage*max_rows + j]->
connectsTo(wr[contig_stage*max_wires_in_acol
+ 2*j + 1]->OUT(), 1000);

                if(j != max_rows - 1)
                {
                    if(i < stages - 1)
                    {
                        se[this_stage*max_rows + j]->
connectsTo(wr[(this_stage + 1)*
max_wires_in_acol

```



```

        + 2*(j + 1) ]->OUT(), 1001);
    }
    else
    {
        se[ this_stage*max_rows + j ]->
        connectsTo( wr[ ( this_stage + 1 ) *
        max_wires_in_acol
        + j + 1 ]->OUT(), 1001 );
    }
}
}
else
{
    cout << "Initial settings out of range 5.\n";
    exit( 1 );
}
}

int conn_type;    // 0 - without queue; 1 - with queue.

searchFor( 'o', fin );
fin >> conn_type;
check_stream( fin, "XXIII" );

switch( conn_type )
{
    case 0:
    {
        for( i = 0; i < outsockets; i++ )
        {
            if( outs[ i ] )
                outs[ i ]->connectsTo( wr[
                max_wires_in_acol *
                ( i + 1 ) ]->OUT(), 1000 );
            else
            {
                cout << "Initial settings out of range 6.\n";
                exit( 1 );
            }
        }
    }
    break;
    case 1:
    {
        fin >> contig_stage;
        check_stream( fin, "XXIV" );
        for( i = 0; i < outsockets; i++ )
        {
            if( outs[ i ] )
                outs[ i ]->connectsTo( wr[ contig_stage *
                max_wires_in_acol + i ]->OUT(), 1000 );
            else
            {
                cout << "Initial settings out of range 7.\n";
                exit( 1 );
            }
        }
    }
}

```

```

    }
    }
    break;
}

for( i = 0; i < q_stages; i++ )
{
    stop = searchFor( 'Q', fin );
    if( stop )
        break;
    check_stream( fin, "XXV" );

    fin >> this_stage >> contig_stage;
    check_stream( fin, "XXVI" );

    fin >> clusters >> in_cluster;
    check_stream( fin, "XVII" );

    fin >> conn_type; // 0 - at an intermediate stage; 1 -
                     // at the output stage
    check_stream( fin, "XXVIII" );

    for( j = 0; j < clusters; j++ )
    {
        con_id = 1000;
        for( k = 0; k < in_cluster; k++ )
        {
            this_row = j * in_cluster + k;
            if( q[ this_stage * q_max_rows + this_row / in_cluster ] )
            {
                switch( conn_type )
                {
                    case 0:
                        q[ this_stage * q_max_rows
                          + this_row / in_cluster ]->
                          connectsTo( wr[
                          max_wires_in_acol *
                          contig_stage + this_row ]
                          ->OUT(), con_id++ );
                        break;

                    case 1:
                        q[ this_stage * q_max_rows
                          + this_row / in_cluster ]->
                          connectsTo( wr[ ( this_row
                          + 1 ) * max_wires_in_acol
                          ]->OUT(), con_id++ );
                        break;

                }
            }
            else
            {
                cout << "Initial settings out of range 8.\n";
                exit( 1 );
            }
        }
    }
}
}

```

```

        }
        break;
    }
}

/*-----D SE TRUCTION OF THE SWITCH-----*/

Switch::~Switch()
{
    sed = se;
    insd = ins;
    outsd = outs;
    qd = q;
    wrd = wr;
    tgd = tg;
    int i;

    for(i = 0; i < max_rows*stages; i++)
    {
        if( *sed )
            delete( *sed );
        sed++;
    }
    for(i = 0; i < q_max_rows*q_stages; i++)
    {
        if( *qd )
            delete( *qd );
        qd++;
    }
    for(i = 0; i < ( stages + q_stages + 1)*max_wires_in_acol; i++)
    {
        if( *wrd )
            delete( *wrd );
        wrd++;
    }
    for(i = 0; i < insockets; i++)
    {
        if( *insd )
            delete( *insd );
        insd++;
    }
    for(i = 0; i < insockets; i++)
    {
        if( *tgd )
            delete( *tgd );
        tgd++;
    }
    for(i = 0; i < outsockets; i++)
    {
        if( *outsd )
            delete( *outsd );
        outsd++;
    }
    if( traffic_generator_present )
        delete( tm );
}

```

```

    free( se );
    free( ins );
    free( wr );
    free( outs );
    free( q );
    free( tg );
}
/*-----*/

/*-----EXECUTION OF THE SWITCH-----*/

void Switch::work()
{
    int i;
    while( !Manager::stop )
    {
        for( i = 0; i <= stages + q_stages; i++ )
        {
            for( int j = 0; j < max_wires_in_acol; j++ )
            {
                if( wr[ i*max_wires_in_acol + j ] )
                {
                    ( wr[ i*max_wires_in_acol + j ]->IN() ).pull();
                }
                if( i == stages )
                {
                    if( j < outsockets )
                    {
                        outs[ j ]->latch();
                    }
                }
                else
                {
                    if( i < stages )
                    {
                        if( j < max_rows )
                        {
                            if( se[ i*max_rows + j ] )
                            {
                                se[ i*max_rows + j ]->latch();
                            }
                        }
                    }
                    else
                    {
                        if( j < q_max_rows )
                        {
                            if( q[ i*q_max_rows + j ] )
                            {
                                q[ i*q_max_rows + j ]->latch();
                            }
                        }
                    }
                }
            }
        }
    }
    Clock::tick();
}

```

```

    }
}
-----*/
/*-----*/

void Switch::work()
{
    int i;
    while( !Manager::stop )
    {
        sed = se;
        insd = ins;
        outsd = outs;
        qd = q;
        wrd = wr;

        for(i = 0; i < ( stages + q_stages + 1 ) *
            max_wires_in_acol; i++)
        {
            if( *wrd )
                ( ( *wrd )->IN() ).pull();
            wrd++;
        }
        for(i = 0; i < max_rows*stages; i++)
        {
            if( *sed )
                ( *sed )->latch();
            sed++;
        }
        for(i = 0; i < q_max_rows*q_stages; i++)
        {
            if( *qd )
                ( *qd )->latch();
            qd++;
        }
        for(i = 0; i < outsockets; i++)
        {
            if( *outsd )
                ( *outsd )->latch();
            outsd++;
        }
        Clock::tick();
    }
}
-----*/

int Switch::searchFor( char c, ifstream& fin )
{
    char temp;
    do
    {
        temp = fin.get();
        if( temp == c )
            return 0;
        if( temp > 64 && temp < 123 )
        {
            fin.putback( temp );
        }
    }
}

```

```

        return 1;
    }
    if( fin.eof() )
        return 1;
    }while( temp != c );
    return 0; //To satisfy compiler
}

void Switch::check_stream( ifstream& fin, const char* c )
{
    if( fin.fail() )
    {
        cout << "Data in the descriptor file corrupted." << c << endl;
        exit( 1 );
    }
}

```

```
/*-----S w itch.H-----*/
```

```
#ifndef _Switch_
#define _Switch_
```

```
#include<SwElement.H>
#include<InSocket.H>
#include<OutSocket.H>
#include<Queue.H>
#include<Wire.H>
#include<Clock.H>
#include<Manager.H>
#include<TManager.H>
#include<TGen.H>
```

```
class Switch
```

```
{
    public:

    Switch( char*, int, char**, int buffer_size );
    ~Switch();

    void work();

    private:

    // Lists of pointers to the objects that are components of an ATM switch

    SwElement** se, **sed;
    InSocket** ins, **insd;
    OutSocket** outs, **outsd;
    Queue** q, **qd;
    Wire** wr, **wrd;
    TManager* tm;
    TGen** tg, **tgd;
    // Variable names are self-explanatory

    int max_rows;
    int rows_in_this_stage;
    int stages;
    int sorting_stages;
    int sw_stages;
    int max_outs;
    int insockets;
    int outsockets;
    int max_wires_in_acol;
    int wires_in_this_col;
    int max_sw_pins;
    int max_pins_in_col;
    int q_stages;
    int q_max_rows;
    int queues_in_stage;
    int q_size;
    int sort_option;
    int uniform_q_size_for_sw;
    int suppressor;
    int con_id;
```

```

int traffic_generator_present;
int input_option;
char tgen_buffer[ 255 ];

// Auxiliary functions
int searchFor( char, ifstream& );
void check_stream( ifstream&, const char* );
};

#endif

/*-----S h u f f l e . C-----*/

#include<stdlib.h>
#include<iostream.h>
#include<math.h>
#include<Shuffle.H>

int n_to_s( int number, int mask, int** str );
int s_to_n( int** str, int size );

int shuffle( int in_cluster, int cluster_on, int in_cluster_on, int pattern )
{
    // Description of patterns in Achille Pattavina, page 66

    int cl = in_cluster;
    int ret, size, bit, i;
    int* temp;
    while( cl != 1 )
    {
        if( cl%2 )
        {
            cout << "Uneven number in_cluster." << endl;
            exit( 1 );
        }
        cl /= 2;
    }

    size = n_to_s( in_cluster_on, in_cluster, &temp );
    switch( pattern ) // All the transformations are performed in the
    {                // reversed order to find the source connection
        // from the destination
        case 0:
            break;
        case 1:    // sigma
            {
                bit = temp[ 0 ];
                for( i = 0; i < size - 1; i++ )
                    temp[ i ] = temp[ i + 1 ];
                temp[ size - 1 ] = bit;
            }
            break;
        case 2:    // inverse sigma
            {
                bit = temp[ size - 1 ];
                for( i = size - 1; i > 0; i-- )
                    temp[ i ] = temp[ i - 1 ];
            }
    }
}

```



```

        temp[ 0 ] = bit;
    }
    break;
case 3:          // beta
{
    bit = temp[ 0 ];
    for( i = 0; i < size - 1; i++ )
        temp[ i ] = temp[ i + 1 ];
    temp[ size - 1 ] = bit;
    bit = temp[ size - 2 ];
    for( i = size - 2; i > 0; i-- )
        temp[ i ] = temp[ i - 1 ];
    temp[ 0 ] = bit;
}
break;
case 4:          // delta
{
    int t_size = ( size - 1 )/2;
    for( i = 0; i < t_size; i++ )
    {
        bit = temp[ i + 1 ];
        temp[ i + 1 ] = temp[ size - 1 - i ];
        temp[ size - 1 - i ] = bit;
    }
}
break;
case 5:          // ro
{
    int t_size = ( size - 1 )/2;
    for( i = 0; i < t_size; i++ )
    {
        bit = temp[ i + 1 ];
        temp[ i + 1 ] = temp[ size - 1 - i ];
        temp[ size - 1 - i ] = bit;
    }
    bit = temp[ 0 ];
    for( i = 0; i < size - 1; i++ )
        temp[ i ] = temp[ i + 1 ];
    temp[ size - 1 ] = bit;
}
break;
}
ret = s_to_n( &temp, size );
ret += cluster_on * in_cluster;
return ret;
}

```

```

// Number to bit string
int n_to_s( int number, int mask, int** str )
{
    int size, scaler, power;

    if( mask == 1 )
        mask = 2;
    size = ( int )( log( mask - 1 ) / log( 2 ) + 1 );
    *str = ( int* ) malloc( size * sizeof( int ) );
    for( int count = size - 1; count > -1; count-- )

```

```

    {
        scaler = 1;
        power = count;
        while( power-- )
            scaler *= 2;

        int outp = 0;
        if( number && scaler )
            outp = 1;
        ( *str )[ count ] = outp;
    }
    return size;
}

// Bit string to number
int s_to_n( int** str, int size )
{
    int scaler, power, accum = 0;
    for( int count = size - 1; count > -1; count-- )
    {
        scaler = 1;
        power = count;
        while( power-- )
            scaler *= 2;
        accum += scaler * ( *str )[ count ];
    }
    free( *str );
    return accum;
}

```

/*-----S h u f f l e . H-----*/

// Class facilitating the construction of a switching element by
 // parametrized interconnecting of the stages.

```

#ifndef _Shuffle_
#define _Shuffle_

```

```

int shuffle( int in_cluster, int cluster_on, int in_cluster_on, int pattern );

```

```

#endif

```

APPENDIX B

This appendix contains a sample output trace of a programme execution and a sample output file "output1.cel" with the traces of ATM cells.

/*-----Very short sample output trace-----*/

ATM SWITCH Simulator, v1.0(Aug 16 2000)

OutSocket OS_5 returned 1 cell
OutSocket OS_6 returned 1 cell
OutSocket OS_5 returned 2 cells
OutSocket OS_3 returned 1 cell
OutSocket OS_6 returned 2 cells
OutSocket OS_0 returned 1 cell
OutSocket OS_4 returned 1 cell
OutSocket OS_7 returned 1 cell
OutSocket OS_2 returned 1 cell
OutSocket OS_4 returned 2 cells
OutSocket OS_5 returned 3 cells
OutSocket OS_6 returned 3 cells
No more data in IS_4
No more data in IS_7
No more data in IS_2
No more data in IS_6
OutSocket OS_5 returned 4 cells
OutSocket OS_0 returned 2 cells
OutSocket OS_6 returned 4 cells
OutSocket OS_4 returned 3 cells
No more data in IS_0
OutSocket OS_6 returned 5 cells
OutSocket OS_5 returned 5 cells
OutSocket OS_7 returned 2 cells
OutSocket OS_6 returned 6 cells
OutSocket OS_1 returned 1 cell
No more data in IS_1
OutSocket OS_6 returned 7 cells
OutSocket OS_1 returned 2 cells
OutSocket OS_0 returned 3 cells
OutSocket OS_3 returned 2 cells
OutSocket OS_6 returned 8 cells
OutSocket OS_3 returned 3 cells
OutSocket OS_1 returned 3 cells
OutSocket OS_0 returned 4 cells
No more data in IS_3
OutSocket OS_6 returned 9 cells
OutSocket OS_0 returned 5 cells
No more data in IS_5
OutSocket OS_1 returned 4 cells

Manager stopped the simulation.

Simulated time 5267 cycles

LAST OBJECT DESTROYED; END OF SIMULATION

/*-----output1.cel-----*/

b0 1e 64 24 52 8b c4 3b 5d bb
35 18 a2 d3 89 ff b2 a0 59 30
f2 db d5 c1 4d 6a 4b 36 9c 5d
78 e6 d0 a3 92 0d e5 90 11 b0
86 0f 41 34 80 a6 89 bd e9 2f
78 00 06

b0 1d ca 35 2e c3 fd 59 f7 01
15 2a da 0f 01 44 ca 47 db a7
67 13 1c 7a 0b 03 82 81 93 b1
bc 60 ed 55 db 8d 66 27 79 16
b1 78 a7 18 b6 8f 98 fb 20 44
0e 00 04

e0 15 5e 88 26 14 ae 28 56 20
e8 66 ed ee 44 77 92 60 d8 7b
60 1f b4 69 61 6b bb bb cc a2
44 d9 fe 91 74 46 3a 7e 59 8c
21 f1 c7 e8 f0 46 f3 b6 7b f4
d1 00 00

c0 19 d0 5a 12 51 d0 00 75 87
a8 4f ba 66 c0 92 d5 d0 f7 b4
86 e5 3f af 55 55 f5 b8 4e 66
01 2c 7d c4 b2 38 28 0c 56 4b
cf 17 9c 3d e4 07 ab 3c 4a 12
fe 01 06

b0 1a 57 a2 23 11 a8 7b 6e df
29 c4 fe 58 fe e2 3b 9b bd bf
4a 7f b6 1e 3b 4f 04 be f5 46
55 36 e1 c9 79 8b 59 66 ab 92
cf 90 a2 09 bd 18 57 73 50 6b
95 00 03

70 1b 70 42 b6 79 36 2f e8 92
7c 53 17 1a 53 3a 66 8d f9 47
0f 27 17 3f fb 56 c0 91 23 e0
2a 66 a9 d4 2a 6d b5 32 71 64
10 5a 89 65 83 35 8d f6 ed 25
ab 01 03

90 1b c5 03 cf f7 be 7d 3c 5a
ad 63 af 32 4e 0d 9f 61 42 e0
ff 1d 75 47 93 47 53 72 f5 8c
aa d1 66 3e 5d 4f 56 f0 ce 4f
b1 d3 09 67 ef d3 b1 df da 71
ab 01 00

e0 12 cb 04 34 6e ed 33 18 a3
71 46 cc eb c3 96 4b eb 6c a7
45 50 2f 15 97 9e e5 3c 8c b7
18 aa d6 2e 62 46 4a 49 84 3d
5f d8 45 b0 64 b8 11 1e 84 26
4d 00 01

APPENDIX C

Guide for using the ATM library.

This appendix contains the guide for using the ATM library and the two example setup files used in the construction of network simulation models.

To construct and simulate a network with the ATM library, the executable file must be linked with the library, and the path to it should be provided in the Makefile. The library must be compiled in the directory home/ATM/CLOCK/LIB. Using the Switch and Switch1 constructors facilitates the construction. Manual for the construction of ATM switch models with Switch and Switch1 constructors is given below. The executables run and run1 are to be compiled in the directory home/ATM/CLOCK.

The setup file structure for Switch and Switch1 constructors

(Figures 1 and 2 are only available in MS Word version of the description file)

Each parameter is marked with a symbol. Explanation of each symbol is provided. Parameters, used only in Switch1 constructor, are given in <>, in Switch constructor those parameters are skipped. Parameters given in [] are placed only in specified conditions, otherwise skipped. The symbols appear in the same order as the parameters are placed in the setup file. For better understanding of setup files it would be useful to look at an example file and Figures 1 and 2 in this manual.

mr, st, <srs>, mo

mr -maximum number of rows that any stage of switching elements can have. **st** - maximum number of stages not including sorting stages. **srs** -- number of sorting stages. **mo** --maximum number of outputs in all the switch boxes, needed to determine what values will be used for valid bits and what value will be the separator.

<uq, [qs]>

uq: 1 -- all switching elements have the queue of the same size, 0 -- queue size will have to be provided for each element separately. **qs** -- common queue size, only if **uq** = 1.

nel(0), <opt(0, 0), [qsz(0, 0)], opt(0, nel(0)-1), [qsz(0, nel(0)-1)]>

nel(1), <opt(1, 0), [qsz(1, 0)], opt(1, nel(1)-1), [qsz(1, nel(1)-1)]>

.

.

.

..

.

nel(st+srs-1), <opt(st+srs-1, 0), [qsz(st+srs-1, 0)], opt(st+srs-1, nel(st+srs-1)-1), [qsz(st+srs-1, nel(st+srs-1)-1)]>

nel(m) -- number of switching elements in stage m. **opt(m, n)** -- functional option of element (m, n), explanation of options is in the thesis report. **qsz(m, n)** - queue size in element (m, n), needed only if **uq=0**.

mnp, in, out

mnp -- maximum number of switching pins (input or output) in all the elements. **in** --number of InSockets. **out** - number of OutSockets.

tma, [argument line for the traffic manager, only if **tma = 1**]

tma = 1 -- traffic manager available, 0 -- traffic manager not available.

tr(0)

.

tr(in-1)

tr(n) -- input option for the InSockets n, 0 - from traces, 1 -- from traffic generator.

qmr, qst

qmr -- maximum number of separate objects of class Queue in any stage of queues. **qst** -- number of stages of queues.

nq(0), ques(0, 0), ... , ques(0, nq(0)-1)

nq(1), ques(1, 0), ... , ques(1, nq(1)-1)

.

nq(qst-1), ques(qst-1, 0), ... , ques(qst-1, nq(qst-1)-1)

nq(n) -- number of Queue objects in each stage of queues, shared queue is considered one object, **ques(m, n)** -- size of the queue (m, n).

supp

supp -- suppressor applied to the last stage of queues.

swp(0), ... swp(st),

[swpq(0), ... swpq(stq-1)]

swp(n) -- maximum number of output pins that any of the elements has in a particular stage n. **swpq(n)** -- maximum number of output pins that any of the Queue objects has in a particular stage n of queue stages, **swpq** is needed only if any queues are available. These parameters make all the connections of objects on the output side. Connections on the input side are made manually or with special functions.

The implementation of the two constructors significantly differs after this point; therefore, separate description will be provided for each of them.

```

/*-----S witch-----*/
Constructor Switch is flexible and allows connecting the elements in any arbitrary order.
-----

"s", 0, 0,
"i", num_conn(0, 0),
s_col(0, 0, 0), s_row(0, 0, 0)
.
.
.
s_col(0, 0, num_conn(0, 0)-1), s_row(0, 0, num_conn(0, 0)-1)
.
.
.
.
"s", st-1, mr-1,
"i", num_conn(st-1, mr-1),
s_col(st-1, mr-1, 0), s_row(st-1, mr-1, 0)
.
.
.
s_col(st-1, mr-1, num_conn(st-1, mr-1)-1), s_row(st-1, mr-1, num_conn(st-1, mr-1)-1)

```

This part connects the switching elements. Each element is marked with two numbers denoting its horizontal and vertical position. **num_conn**(m, n) - number of connections that element (m, n) has on the input side. **s_col** (m, n, k) - column of the Wire object, to which element (m,n) is to be connected. **s_row** (m, n, k)-row of the

Wire object, to which element (m, n) is to be connected. Index k shows the connection number.

```
"o", 0,
o_col(0), o_row(0)
.
.
.
.
"o", out-1,
o_col(out-1), o_row(out-1)
```

This part connects the OutSocket's. Each OutSocket is marked with a number denoting its horizontal position. **o_row** (m) - row of the Wire object, to which OutSocket (m) is to be connected. **o_col** (m) - column of the Wire object, to which OutSocket (m) is to be connected.

```
"q", 0, 0,
"i", q_num_conn(0, 0),
q_col(0, 0, 0), q_row(0, 0, 0)
.
.
.
q_col(0, 0, q_num_conn(0, 0)-1), q_row(0, 0, q_num_conn(0, 0)-1)
.
.
.
.
"q", qst-1, qmr-1,
"i", q_num_conn(qst-1, qmr-1),
q_col(qst-1, qmr-1, 0), q_row(qst-1, qmr-1, 0)
.
```


q_col(qst-1, qmr-1, q_num_conn(st-1, mr-1)-1), q_row(st-1, mr-1, q_num_conn(st-1, mr-1)-1)

This part connects the queues. Each queue is marked with two numbers denoting its horizontal and vertical position. **q_num_conn**(m, n) - number of connections that queue (m, n) has on the input side. **q_col** (m, n, k) - column of the Wire object, to which queue (m, n) is to be connected. **q_row** (m, n, k) - row of the Wire object, to which element (m, n) is to be connected, k is the connection number.

It is recommended to look at the examples in Figures 1 and 2 explaining how the objects are interconnected in the baseline and crossbar architectures.

```
/*-----S w itch1-----*/
```

Constructor Switch1 is intended to construct the networks exactly in the way they are shown in Figures 1 and 2.

"C", option

option: 1 -- this is a multistage banyan network, 2 -- this is a crossbar network.

```
/*-----C = 1-----*/
```

"S", this_stage(0), contig_stage(0), clusters(0), in_cluster(0), pattern(0),
num_con(0)

.

"S", **this_stage**(st+srs-1), **contig_stage**(st+srs-1), **clusters**(st+srs-1),
in_cluster(st+srs-1), **pattern**(st+srs-1), **num_con**(st+srs-1)

this_stage - the stage of elements, whose inputs are to be connected to the stage of elements **contig_stage**. **clusters** - number of clusters of Wires between the two stages being connected. **in_cluster** - number of Wires in one cluster, for example in Fig 1 between stages 1 and 2 there are 2 clusters with 4 Wires in each. **pattern** - shuffle pattern, according to which the stages are interconnected, the patterns are defined in the book "Switching Theory", Achille Pattavina p66 and partially implemented in Shuffle.C. **num_con** - number of connections that a box has on the output side, needed to assign proper connection ID's, for example if **num_con** = 2, the connections will be assigned ID's of 0 and 1.

"o", **o_contig_stage**

o_contig_stage -- the stage, to which the output sockets are going to be connected.

"Q" **q_this_stage**(0), **q_contig_stage**(0), **q_clusters**(0), **q_in_cluster**(0)

.
.
.

"Q" **q_this_stage**(qst-1), **q_contig_stage**(qst-1), **q_clusters**(qst-1),
q_in_cluster(qst-1)

q_this_stage -- the stage of queues, whose inputs are to be connected to the stage of elements **q_contig_stage**. **q_clusters** -- number of clusters of Wires between the two stages being connected. **in_cluster** - number of Wires in one cluster, for example, connection

of the first stage of queues to InSockets has 8 clusters with 1 Wire, connection of the last stage output queue has 1 cluster with 8 Wires.

```
/*-----C = 2-----*/
```

```
"F", cr_contig_stage_f
```

```
"S", cr_this_stage(0), cr_contig_stage(0)
```

```
.
```

```
.
```

```
.
```

```
"S", cr_this_stage(st+srs-1), cr_contig_stage(st+srs-1)
```

```
"o", cr_o_conn_type, [cr_o_contig_stage]
```

```
"Q", cr_q_this_stage(0), cr_q_contig_stage(0),
```

```
cr_q_clusters(0), cr_q_in_cluster(0), cr_q_conn_type(0)
```

```
.
```

```
.
```

```
.
```

```
"Q", cr_q_this_stage(qst-1), cr_q_contig_stage(qst-1),
```

```
cr_q_clusters(qst-1), cr_q_in_cluster(qst-1), cr_q_conn_type(qst-1)
```

cr_contig_stage_f - the stage of Wire's that the first stage of elements of a crossbar connects to. **cr_this_stage(n)** and **cr_contig_stage(n)**: inputs of elements of **cr_this_stage** connect to elements of **cr_contig_stage**. **cr_o_conn_type** - type of connection of the stage of OutSockets: 0 -- directly to the stage of switching elements, 1 -- to a queue stage (Fig. 2). **cr_o_contig_stage** - stage, to which OutSockets connect if **cr_o_conn_type** = 1. Inputs of queues of **cr_q_this_stage** connect to Wires of **cr_q_contig_stage**. **cr_q_clusters** - number of clusters of Wires between the two stages.

cr_q_in_cluster - number of Wires in one cluster. cr_q_conn_type
- type of connection of the Queues, 0 -- at an intermediate stage, 1 --
at the output stage, in Fig 2 the queues of the first stage are of type 0,
the queues of the last stage are of type 1, if cr_q_conn_type = 1,
cr_q_contig_stage is discarded.

Figure 1. Example of object interconnection and marking in a baseline network.

Figure 2. Example of object interconnection and marking in a crossbar network.

/*-----I N IT/q_baseline3x4.des-----*/

4 3 2
4 4 4
2 8 8

1 -poisson -lambda 0.7 -traces 4

1
1
1
1
1
1
1
1
1

8 2
8 8 8 8 8 8 8 8 1 1 0
2 2 2 2

1 8

s 0 0
i 2
4 0
4 1

s 0 1
i 2
4 2
4 3

s 0 2
i 2
4 4
4 5

s 0 3
i 2
4 6
4 7

s 1 0
i 2
1 0
1 2

s 1 1
i 2
1 4
1 6

s 1 2
i 2
1 1
1 3

s 1 3
i 2
1 5
1 7

s 2 0
i 2
2 0
2 2

s 2 1
i 2
2 1
2 3

s 2 2
i 2
2 4
2 6

s 2 3
i 2
2 5
2 7

o 0
5 0

o 1
5 1

o 2
5 2

o 3
5 3

o 4
5 4

o 5
5 5

o 6
5 6

o7
57

q00
i1
00

q01
i1
01

q02
i1
02

q03
i1
03

q04
i1
04

q05
i1
05

q06
i1
06

q07
i1
07

q10
i8
30
31
32
33
34
35
36
37

/*-----INIT1/q_baseline3x4.des-----*/

4 3 0 2
1 3 5

4 1 1 1 1
4 1 1 1 1
4 1 1 1 1

2 8 8

1 -poisson -lambda 0.9 -traces 40

1
1
1
1
1
1
1
1
1
1

8 2
8 8 8 8 8 8 8 8 1 4 0
2 2 2 2

1 8

C 1

S 0 4 1 8 0 2
S 1 1 1 8 2 2
S 2 2 2 4 1 2

o 5

Q 0 0 8 1
Q 1 3 1 8