by

Rubens Carlos de Souza Gomes
B.S.Co.E., The University of Kansas, 1993

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science.

Dr. David Petr

Professor in Charge

Dr. Douglas Niehaus

Dr. Susan Gauch

Committee Members

Date Defended: December 06, 1996

This paper presents a simulation study of an input buffered Asynchronous Transfer Mode (ATM) switch running the Request-Grant-Status (RGS) iterative scheduling algorithm. The RGS algorithm can reduce the effects of head-of-line blocking on the throughput and delay performance of an input buffered ATM switch. During a cell slot interval, the RGS algorithm performs a fixed number (n) of iterations. At each iteration step, the RGS algorithm determines a match between buffered cells and corresponding idle output ports. After the last iteration step, all matched cells are routed through the switch fabric.

The simulation results show that the RGS algorithm can significantly improve the throughput and delay performance of an input buffered ATM switch under a high load. When independent and identical Bernoulli traffic sources are used, the input queues of a 16x16 ATM switch, using a strict first-in first-out queueing discipline, saturate at an offered load of approximately 60%. When the RGS algorithm is used with 5 iterations, the input queues of a 16x16 input buffered switch, under the same traffic conditions, do not saturate until the offered load is approximately 98%.

Switch throughput and average queueing delays are the performance parameters studied in this paper. These parameters are analyzed in terms of the traffic source model (Bernoulli or On-Off Markov), offered load, number of iterations, switch sizes, and average burst length for correlated traffic source (On-Off source). The input queues are assumed to be of infinite length; therefore, no cell loss performance is collected in this simulation study.

I would like to express my sincere appreciation to Dr. David W. Petr, my committee chairman, for his guidance and support throughout the course of this research project. In addition, I would like to thank Dr. Shusaburo Motoyama of the University of Campinas, Brazil. Dr. Motoyama not only developed the Request-Grant-Status scheduling algorithm, but his guidance and help in the earlier stages of this project was invaluable. Finally, I would like to thank Dr. Douglas Niehaus and Dr. Susan Gauch for serving on my committee.

I should also thank my father, Galdino Ribeiro Gomes, and my mother, Quelusa Maria de Souza Gomes, who supported me in every aspect of my education. I want to acknowledge their love and hard work to provide me with the opportunity to have the best education possible.

Furthermore, I would like to take this opportunity to thank the Institute of International Education and the University of Kansas Foreign Student Scholarship Program for providing me with international airline tickets and a five-year full-tuition scholarship to pursue undergraduate studies in the United States. Without this scholarship I would not have been able to be where I am today.

The average cell delay and throughput performance of an input buffered ATM switch can be substantially degraded due to head-of-line blocking. Head-of-line blocking is a common problem with FIFO (First-In First-Out) input buffered switches where conflicting cells located at the heads of the queues, destined for a certain output, block other cells from being routed to free output ports. The Request-Grant-Status (RGS) iterative scheduling algorithm proposed in [1] can reduce the effects of head-of-line blocking on an input buffered ATM switch. Input buffered switches are attractive, because they permit the switch fabric network to run at the speed of the input links. If in a given cell time slot, cells destined to the same output are not to be discarded, output queued switches require the switch fabric to operate n times faster than the input lines. The Knockout [2] is a an output queued switch where the fabric operates at the same speed as the input ports. But in the Knockout switch, cells destined to the same output can be discarded (or knocked out) as they compete for a specific output port. Therefore, if no cells are discarded, and the input links operate at high speeds, the fabric of an output queued switch is required to run at a very fast rate. As the size of the switch increases, the clock rate reaches very high speed, which may complicate the switch hardware design.

The RGS scheduling algorithm performs a fixed number, n, of iterations. At each iteration, the algorithm attempts to determine a match between waiting cells, stored in input buffers, and their corresponding output ports. After the last iteration, the cells that have been matched are routed through a NxN contentionless switch fabric. The time the ATM switch takes to perform n iterations is assumed to be less than one cell slot. For an input port capacity of 155 Mbits/sec (~366,000 cells/sec), a cell slot lasts approximately 2.74 μseconds.

The purpose of this simulation study is to evaluate the performance of the RGS scheduling algorithm in terms of the traffic source model (Bernoulli, On-Off), the number of iterations, switch size, offered load, and average burst length (On-Off source). Average cell delay and throughput performance metrics are derived by running a computer simulation program. The switch throughput is defined to be the ratio of the total number of cells routed by the total number of cells arrived during a simulation run. The switch cell delay is an average of all queueing delays in a simulation run. Cell delay statitistics are collected after the initial 10% (transient time) of the total simulation time. A theoretical throughput performance analysis of an ATM input buffered switch running the RGS algorithm was done for 1(one) iteration. An attempt was made to extend the theoretical performance analysis for 2(two) or more iterations. Due to the complexity involved in the theoretical throughput analysis of the ATM switch at higher number of iterations, the simulation has become the most viable alternative to predict the performance of the RGS algorithm. Saturated throughput results based on a 'simplified' theoretical analysis of the RGS algorithm, published in [1], have been compared with the simulation results. The results of this comparison can be seen in chapter 5 (Simulation Resutls). In addition, the saturated throughput simulation results have been validated for 1(one) iteration. The simulation validation can also be found in chapter 5.

Assumptions for the simulation, diagrams, algorithms, and simulation object oriented models are provided throughout the paper. The simulation program was implemented in C++, and the source code listing can be found in the Appendix. The code has been

compiled, linked and tested in the following platforms: HP UNIX, SunOS, DEC OS/F UNIX, and Windows95. The source code and executable files can also be found in the EECS (Electrical Engineering and Computer Science) UNIX domain of the University of Kansas at the following path: ~rgomes/MS_Proj/Simulation. The executable filename is called "rgs". Further information on how to run the simulation program can be obtained in the Appendix, "Simulation Program User's Guide", or by consulting the README file in the above file path.

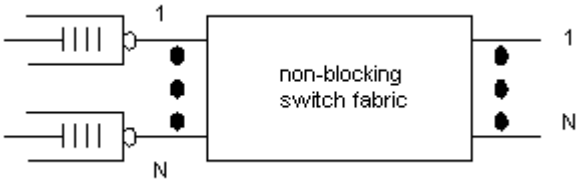

Fig. 1 Input Queueing

In the type of queueing architecture displayed in Fig. 1, a separate buffer is placed on each input port of the switch. The input buffers may operate in a First In First Out (FIFO) or First In Random Out (FIRO) fashion. If no scheduling algorithm is used to select which cells to transmit at the beginning of the routing cycle, head-of-line blocking may occur with more frequency. When 'k' cells at the heads compete for the same output, only one is allowed to pass through, and 'k-1' cells must wait for the next routing cycle. In the meantime, while one of the 'k-1' cells waits for its turn, other cells are queued in the FIFO, and blocked from reaching possibly idle output ports in the switch. Fig. 2 shows head-of-line blocking for a 4 by 4 switch.

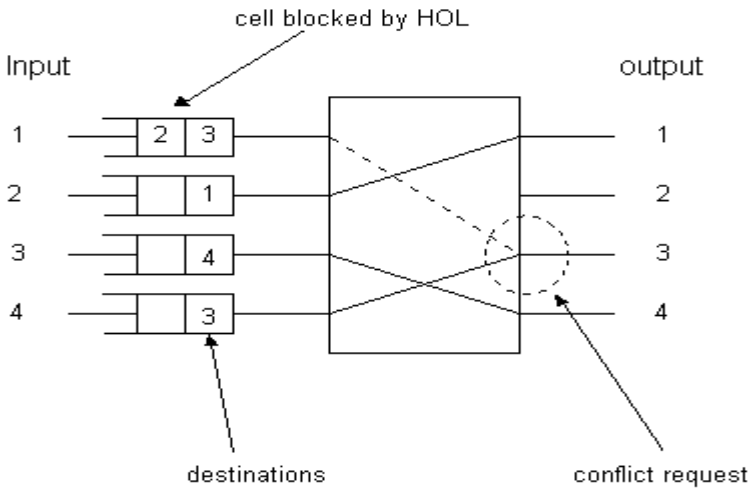

Fig. 2 Head of Line Blocking

A theoretical performance analysis study of a strictly input queued space-division packet switch, without arbitration, was conducted in [3]. For independent and identical Bernoulli traffic sources, a large number of input ports, and incoming packets uniformly distributed among all outputs, it was determined from the analysis [3] that the maximum saturated throughput is approximately 58.6%. With the scheduling algorithm presented in this paper, the throughput of a 64 x 64 ATM switch may reach 96% for five iterations under the same input traffic assumptions.

The proposed RGS scheduling algorithm to prevent head-of-line blocking uses an iterative process. At each iteration step, the RGS scheduling algorithm checks the cells that are buffered in input queues, and attempts to match those cells with their corresponding output ports. The iteration step consists of three phases: request, grant and status. After the last iteration, the algorithm removes the matched cells from their respective buffers, and routes them across the switch fabric.

A diagram of a 3x3 input buffered ATM switch running the RGS algorithm is shown in Fig. 3. The RGS scheduling algorithm runs on the input and output control blocks, which are highlighted in Fig. 3. Incoming cells are placed in RAM (Random Access Memory) buffers which contain address pointers to each cell. The non-blocking switch fabric can be a crossbar. The Cell Header Reading block maps VCI/VPI to respective output port numbers.

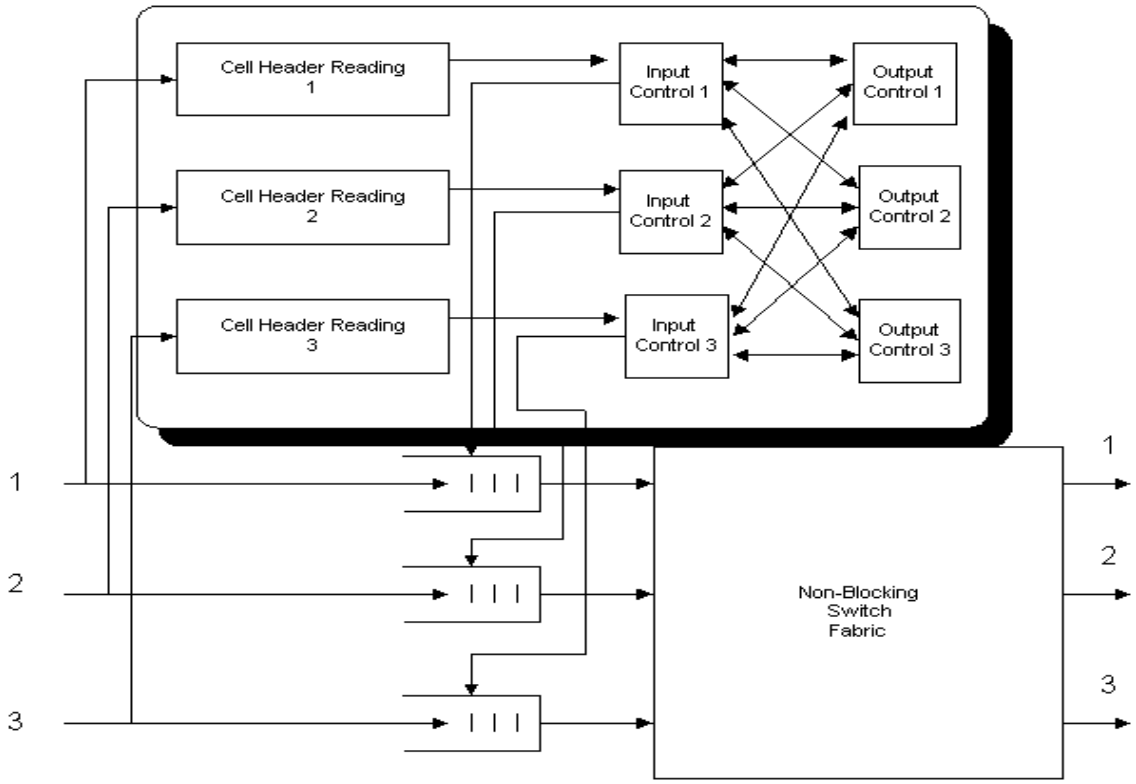

Fig. 3 3 X 3 Switch Architecture

Fig. 4 shows the input and output control modules in detail. Each input control unit has an output table (OT) containing the output port numbers of incoming cells, a cell address table (CAT) for the cell locations, cell selected position pointer, status information registers (SIR) to keep track of matched outputs, and an input reservation register (IRR). Each output control unit has a decision and reservation control (DRC), and an output reservation register (ORR). The IRR and ORR registers keep the status of the control unit. When the IRR or ORR is set to "1" the control unit is reserved; otherwise, the control unit is idle.

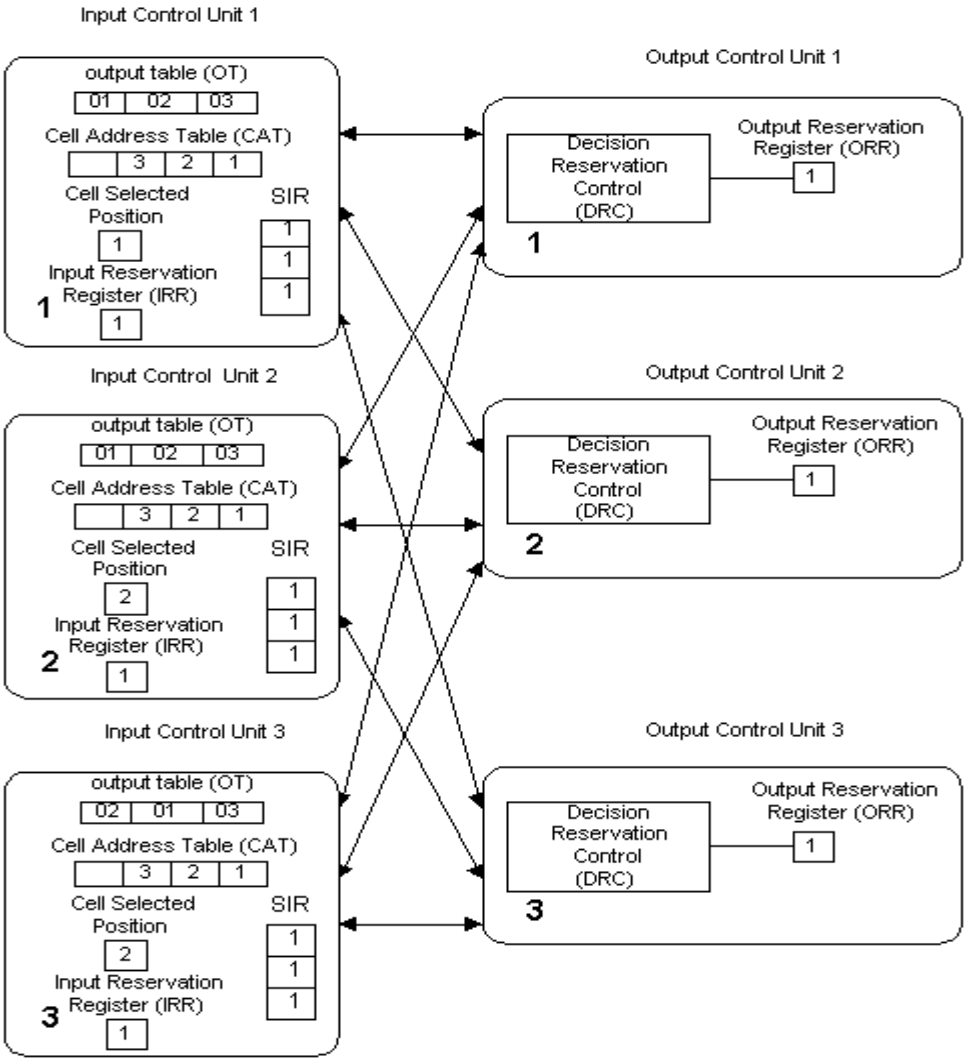

Fig. 4 Structure of Scheduling Control Module

The operation of the algorithm relies on three phases as follows. During the request phase, each ICU checks its output table (OT) to determine the destination output port of the cell stored in the head of the queue. Then, the ICU checks the status of that destination output port by consulting the respective output port status information register (SIR). If the corresponding output port is reserved, which is indicated by the corresponding SIR register being marked as reserved "1", then the ICU reads the destination of the next cell in the OT table. This procedure is repeated until a free output port is found or no more cells are found in the OT (output table). If a free output port is encountered, the ICU sends a request signal to the respective OCU unit. If no free output

ports are found or the end of the queue is reached, no requests are sent for that particular ICU in the current iteration.

Notice that in the first iteration all the SIRs (Status Information Registers) are set to "FREE", and requests are sent for all cells located at the heads of the input queues. If 'k' requests are received in a particular OCU (output control unit), one among 'k' requests is uniformly selected, and a grant signal is sent to the corresponding ICU (input control unit). Then, the status signal of the matched output control unit is broadcasted to all ICUs. This status signal sets the corresponding SIR to "BUSY". Therefore, if only one iteration is used in the RGS algorithm, the ATM input buffered switch behaves like an input queued switch with strict First-In First-Out (FIFO) buffers. The performance of an input buffered ATM switch can only be improved when 2(two) or more iterations are used in the RGS algorithm. In the second or higher iteration step, the status of an SIR can be BUSY, which forces the RGS algorithm to search for the next cell in the queue before

unit that sent the status signal. By setting the SIR (status information register) to "BUSY", no request can be sent to the corresponding OCU unit in the next iteration step. Only one bit is required for the request, grant, and status signals, which allows quick communication between ICUs and OCUs.

At the end of the last iteration, those ICUs (input control units) which have their IRR (input reservation register) marked as reserved "1", route their respective selected cells through the fabric. The address of the selected cell is determined by consulting the cell selected position pointer. Then, all the registers are re-initialized to their initial state (IRR=0, ORR=0, SIRs=0), and the cell selection position pointer is set to NIL.

An input buffered ATM switch using a similar type of algorithm, called Parallel Iterative Matching (P.I.M.) was proposed in [4]. According to [4], a 16x16 switch based on the PIM algorithm can reach a throughput of 98% for four iterations, regardless of the offered load, when independent and identical Bernoulli traffic sources are used. The PIM algorithm has three phases in each iteration: request, grant, and accept. In the request phase, an unmatched input sends requests to all outputs for which it has buffered cells. If an output receives any request, it chooses one randomly to grant and notifies the chosen input. Since multiple requests are sent per input port, that input port may receive more than one grant signal (refer to Fig. 5 below). In the accept phase, if an input receives any grants, it chooses one to accept and notifies that output. The P.I.M. scheduling algorithm has been implemented in the DEC (Digital Equipment Corporation) AN2 ATM input buffered switch [10].

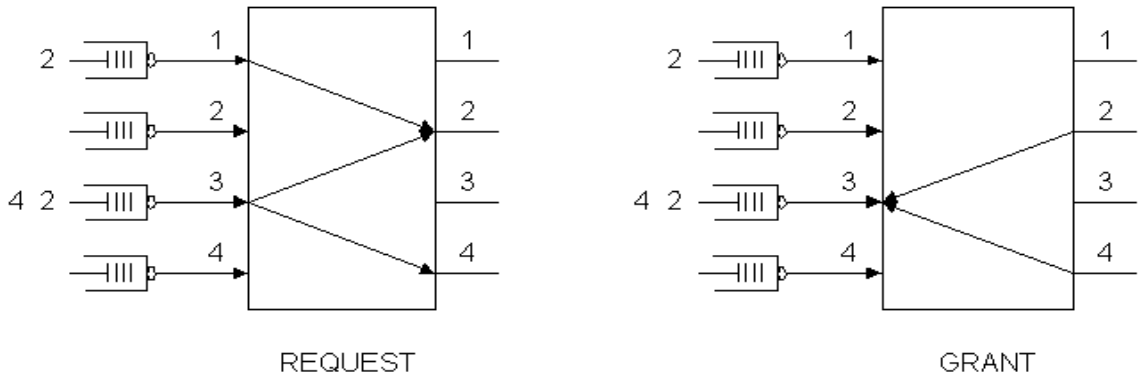

Fig. 5 P.I.M. Algorithm Request and Grant Phases

The main difference between the RGS and the PIM algorithm is that in the RGS algorithm at most one request per input port is sent during an iteration step. Because only one request is sent per input port, the OCU (output control unit) in the RGS algorithm does not become too overloaded with requests. In addition, after the first iteration, the RGS algorithm only sends requests to FREE output control unit, which can substantially reduce the traffic between ICUs and OCUs. Furthermore, the P.I.M. algorithm requires two uniform selections, one at the outputs to select one among multiple requests, and another selection at the inputs to select one among multiple grants (see Fig. 6 below). The RGS algorithm requires only one uniform selection at the outputs. On the other hand, if only 1(one) iteration is used, the P.I.M. can have a better performance than the RGS algorithm because more than one request can be sent per input port in the PIM algorithm.

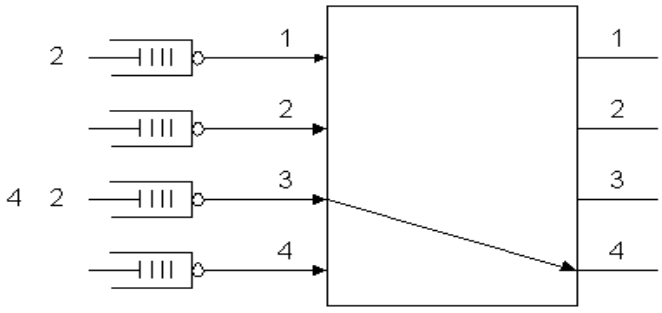

Fig. 6 P.I.M. Algorithm Accept Phase

Computer simulation is a valuable tool used by engineers and scientists to better understand certain aspects of a system, and to predict its performance under a set of assumptions. A computer simulation involves the design of a model of an actual or theoretical system, the execution of the model in a digital computer, and the analysis of the simulation results or execution outputs [5]. A performance analysis study based on computer simulation results can be the most viable alternative to analyze a system whose theoretical performance analysis is otherwise very complex.

In this paper the system under study is a theoretical ATM input buffered switch running the RGS (Request-Grant-Status) iterative scheduling algorithm. Only certain aspects of the ATM switch, such as traffic source, input queues, and switch controllers have been modeled and implemented in the simulation program. One of the main goals of this simulation study is to estimate the average queueing cell delay and throughput performance of an ATM input buffered switch running the RGS algorithm. The queueing delay and throughput performance metrics are determined as a function of number of iterations, number of input/output ports, type of traffic source (Bernoulli and OnOff), offered load, and average cells burst length (OnOff source).

Two types of traffic sources are modeled and implemented in the simulation program: Bernoulli and On-Off. The Bernoulli is a non-correlated cells source; i.e., the random event that decides on an output port for a new cell is independent of the previous event that selected the output port of a previously created cell. The offered load for the Bernoulli traffic source is specified by 'p', probability of an arrival in a given slot, which is also independent of previous slots. The second type of traffic source is the On-Off traffic source. The On-Off source operates as a two state Markov Chain. At the ON state, the source generates a cell destined for a specific output port. At the OFF state, no cells are generated. Two parameters are required for the On-Off traffic source: offered load, and average burst length.

In order to better understand the performance of the RGS algorithm under various conditions, several parameters are specified for each simulation run. The following parameters are to be specified by the user at the start of a simulation run:

- type of traffic source: Bernoulli or OnOff
- initial seed and multiplier to be used by first traffic source
- number of iterations: 1, 2, 3, 4, 5, 6
- number of ports: 16, 32, and 64.
- offered Load (between 0 and 1)
- average burst length (only available for correlated On-Off source)
- simulation time in number of cell slots.

The following simplifications and assumptions are made for the simulation:

1. The mapping of input VCI/VPI (Virtual Circuit Identifier/Virtual Path Identifier) and input port number to output VCI/VPI and output port number is not included in the simulation program.
2. An ATM cell consists only of fields for the input and output port numbers and arrival and departure time stamps.
3. The arrival of a cell in an input queue is modeled after either a discrete Bernoulli process, or a discrete 2-state (ON or OFF) Markov process. For the On-Off source, a burst of cells destined to a single output is generated in the ON state.
4. The output port for a cell (Bernoulli traffic) and burst of cells (on-off traffic) is a uniform random number between 1 and N, where N is the number of output ports.
5. The input and output port capacity of the ATM switch is 155Mbits/s (OC-3). Therefore, the cell slot cycle is assumed to be approximately 2.74 μseconds.
6. The input buffers are infinite. No cells are lost during the simulation.
7. The switch fabric is non-blocking, or contentionless.
8. Performance metrics are measured when cells enter and leave the input queues.
9. Cell delays are collected after the initial transient state, which is assumed to be 10% of the total simulation time.
10. Switch throughput is defined to be the ratio of total number of cells routed by the total number of cells arrived during a simulation run. This calculation includes cells created and arrived during transient state. Therefore, the simulation should run long enough (50,000 or greater) to minimize the effects of the transient state on throughput results.

Modeling serves as language to describe a system at some level of abstraction or at multiple levels of abstraction [5]. To model a system can be thought as way to abstract from reality a description of a system. Therefore, models not only allow us to have a better understanding of a system under study, but also serve as the foundations to build a good system architecture. The development of a model consists of reasoning about a system, and describing in a textual or graphical language what the system does, and how it performs tasks in its domain. Texts, graphs, diagrams, and equations are some of the tools which are used for modeling purposes. In the development of a software system, such as a computer simulation, the models can be categorized as requirements, analysis, design, implementation, and testing models [6]. The models developed at each stage provide the basis for developing or testing subsequent models.

In this paper, the system being modeled is a simulator for an ATM input buffered switch running the RGS algorithm. An object-oriented (OO) modeling approach has been chosen to analyze, design, and implement the simulator program. By using an object oriented approach to model the simulator, the construction of our models focus on the identification, and exposure of objects and object interactions. One of the key

contributions of the object oriented modeling approach to build simulation systems is the mapping between real world and digital world entities. Physical objects encountered in the ATM switch, such as traffic sources, input queues, and input and output control units, are easily mapped to programming objects in the simulator. These programming objects are run in a digital computer by the simulation program. Objects in the simulation program stimulate other objects by sending message to one another. In addition, responsibilities are assigned to objects in the simulator in a similar way those responsibilities are assigned to their corresponding physical entities in the real world.

The development of the simulator was divided into four phases: analysis, design, implementation, and testing. During the analysis modeling phase, scenarios, scenario diagrams and an analysis object model were constructed to describe the main functionality of "what" the simulator does. Fundamental objects in the simulator domain were identified during the analysis phase. The models constructed in the design phase describe the dynamics of the system; i.e., "how" the system does the tasks identified in the analysis phase. The design models were finally implemented using an object oriented programming language (C++). During the testing phase, the results of the simulator were compared to theoretical results. In the course of the development of the simulator program, these models were iterated and modified several times. Therefore, the models presented in this paper are the final ones.

A diagram of a 2 X 2 input buffered ATM switch is presented in Fig. 7 below. ICU, OCU, queue, ATM cell, and traffic sources are some of the objects easily identified in the diagram.

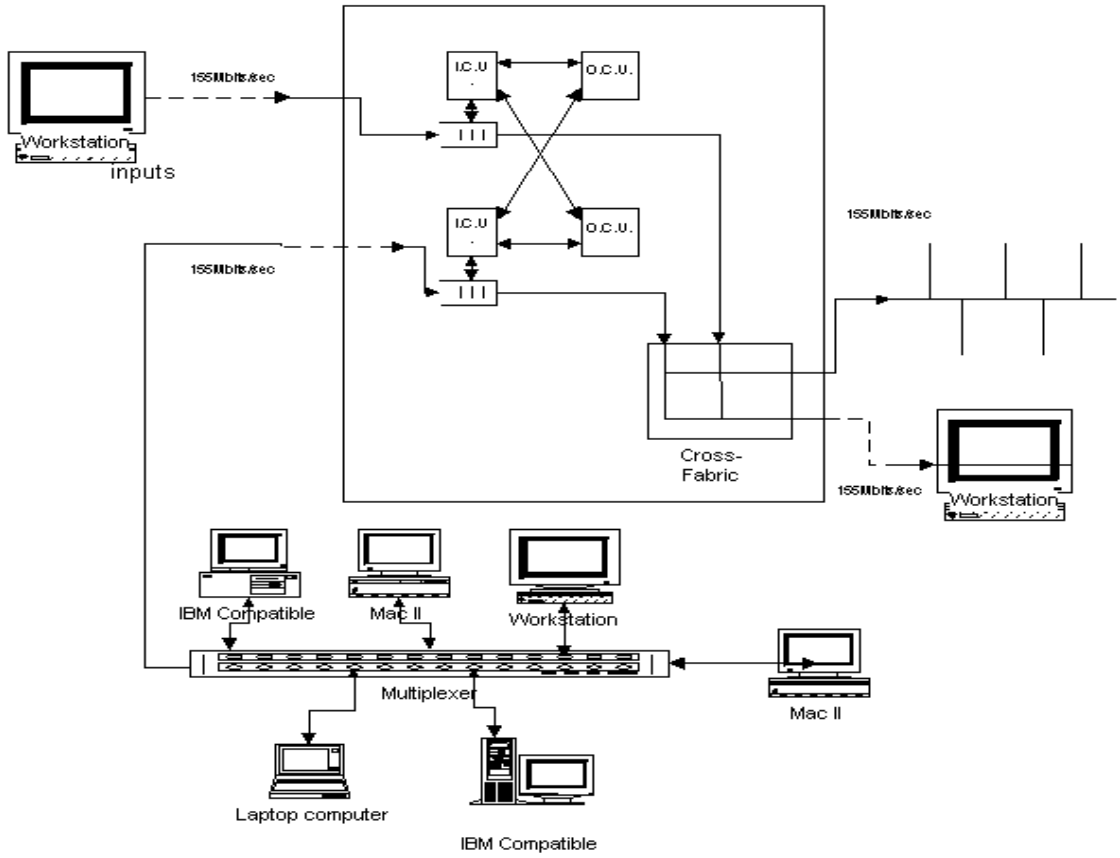

Fig. 7 2 X 2 ATM RGS Switch on a local network

Scenario modeling is usually one of the first steps in the object-oriented analysis of a computer software system. In this stage a set of possible scenarios is constructed to describe different processes within the domain of the system under study. A scenario describes an outline of activities which correspond to some system behavior. Therefore, scenarios help clarify the understanding of what the system does. Also, fundamental objects in the system can be uncovered during the scenario modeling phase. A picture containing the various components participating in a given scenario, and the messages between these components over time is presented in the scenario diagrams.

The scenario models presented in this paper consist of a textual and diagram description of several operations in a ATM switch running the RGS algorithm. Each scenario is described by scenario name, classes, assumptions (or preconditions), and basic course of events. The classes describe a list of objects which are likely to participate in that scenario.

Scenario Name: Starting Simulation Program

Classes:

- Simulation
- User Interface

Preconditions: none

Basic Course of Events:

- this scenario begins when the user starts simulation program by entering "rgs" in the system prompt
- user selects type of traffic source (Bernoulli or OnOff)
- user enters the initial seed for random nr generator
- user enters the initial multiplier for random nr generator
- user enter number of cell slots to run simulation
- user enters switch size: 16, 32, 64. (maximum is 64)
- user enters number of iterations: 1, 2, 3, 4, 5 or 6
- user enters offered load ($0 < load < 1$)
- user enters average burst length (OnOff source)
- all the data entered are stored inside Parameter object
- simulation program runs until number of cell slots to run simulation

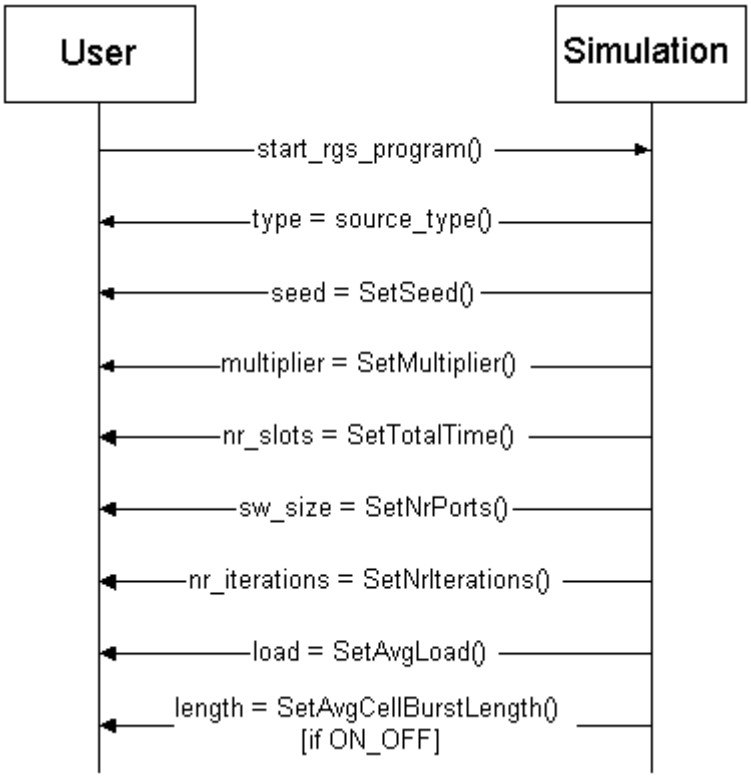

Fig. 8 Scenario Diagram for starting simulation

Scenario Name: Generating an ATM cell

Classes:

- Traffic Source
- Input Queue
- Stats Probe
- Timer
- Simulation

Preconditions: a cell is generated

Basic Course of Events:

- this scenario begins when the cell_slot timer is triggered, and a time signal is sent to the simulation to indicate beginning of a cell slot.
- traffic source generates an ATM cell

- ATM cell is stored in corresponding input queue
- increment number of cells stored in input queue
- increment number of cells arrived in the statistical probe

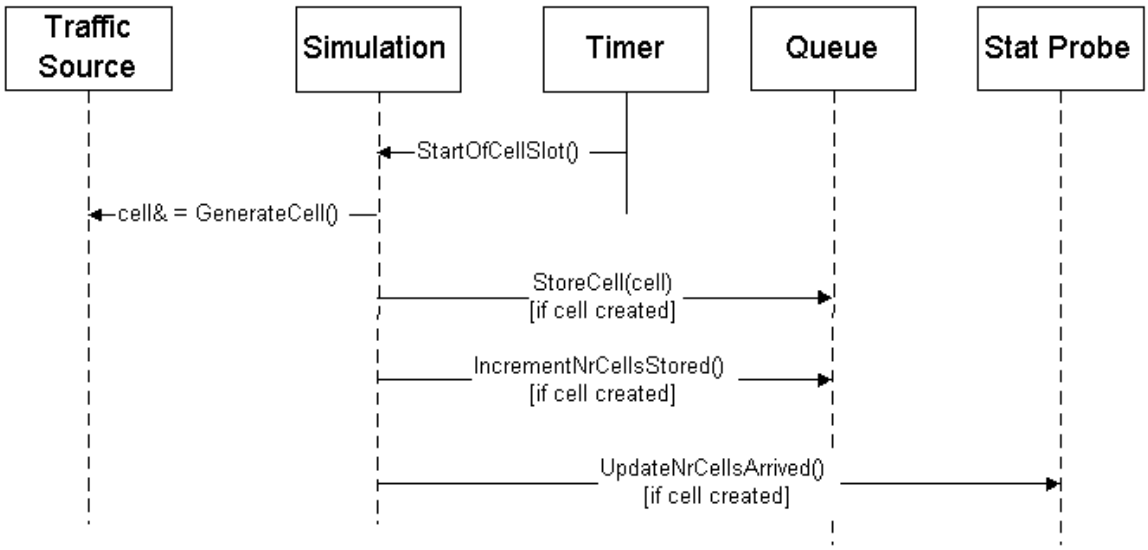

Fig. 9 Scenario Diagram for generating ATM cell

The scenario "Generating an ATM cell" is divided into "Generating an ATM cell with Bernoulli Source" and "Generating an ATM cell with OnOff Source".

Scenario Name: Generating an ATM cell with Bernoulli Source

Classes:

- Bernoulli Traffic Source
- Input Queue
- Stats Probe
- Random Number Generator

Preconditions: Traffic source is Bernoulli

Basic Course of Events:

- generate a uniform random number, and compare to the probability of arrival (offered load.)
- if the uniform random number generated is greater than the probability of arrival, then a cell is created
    - generate a uniform random number between 1 and N, where N is the size of the switch. This uniform number is used for the output port of the ATM cell created
    - stamp the ATM cell with respective input port number, arrival time, and output port number
    - store the ATM cell in its respective Input Queue
    - add number of cells arrived in the Statistical Probe
- otherwise, if the uniform random number generated is less than the probability of arrival, nothing is done by the traffic source.

Scenario Name: Generating an ATM cell with OnOff Source

Classes:

- OnOff Traffic Source
- Input Queue
- Stats Probe
- Random Number Generator

Preconditions: source was in the ON state prior to current cell slot

Basic Course of Events:

- cell is generated. A cell is always generated when the source is in the ON state prior to the current cell slot.
- stamp the ATM cell with respective input port number, current arrival time, and output port number. The output port number should be the same used in the previous cell slot.
- After the cell is generated, the next state of the traffic source is determined. In order to determine the next state of the traffic source, a uniform random

number (between 0 and 1) is generated. This uniform random number is compared to the probability of On-to-On.

- if the uniform random number is less than the probability of On-to-On, source changes to OFF state
- otherwise, if the uniform random number generated is greater than the probability of On-to-On, the source stays in the current state (On state).

Scenario Name: Generating an ATM cell with OnOff Source

Classes:

- OnOff Traffic Source
- Input Queue
- Stats Probe
- Random Number Generator

Preconditions: source was in the OFF state prior to current cell slot

Basic Course of Events:

- cell is generated. A cell is always generated when the source transitions from the OFF to the ON state.
- Since the source has just transitioned from the OFF state to the ON state, an output port needs to be generated for the new cell created. In order to determine an output port number, a uniform random number between 1 and N (N is size of switch) is generated. This uniform number is used as the output port for the newly created cell.
- stamp the ATM cell with respective input port number, current arrival time, and output port number.
- After the cell is generated, the next state of the traffic source is determined. In order to determine the next state of the traffic source, a uniform random number (between 0 and 1) is generated. This uniform random number is compared to the probability of On-to-On.
  - if the uniform random number is less than the probability of On-to-On, source changes to OFF state

- otherwise, if the uniform random number generated is greater than the probability of On-to-On, the source stays in the current state (On state).

Scenario Name: Sending requests (Request Phase)

Classes:

- Simulation
- RGS_Protocol
- InputQueue
- ICU
- OCU

Preconditions: queue is not empty, and IRR not busy

Basic Course of Events:

- this scenario begins at the beginning of each iteration step
- read the output port number of the cell at the head of the queue
- if output port number is destined to free OCU
  - send a request to that OCU.
- otherwise, skip the current cell and read the output port number for the next cell in the queue
- repeat previous step until the end of the queue is reached or a free OCU is found.
- send a request and stop process; or if no free OCU is found, or end of queue is reached, stop the process.

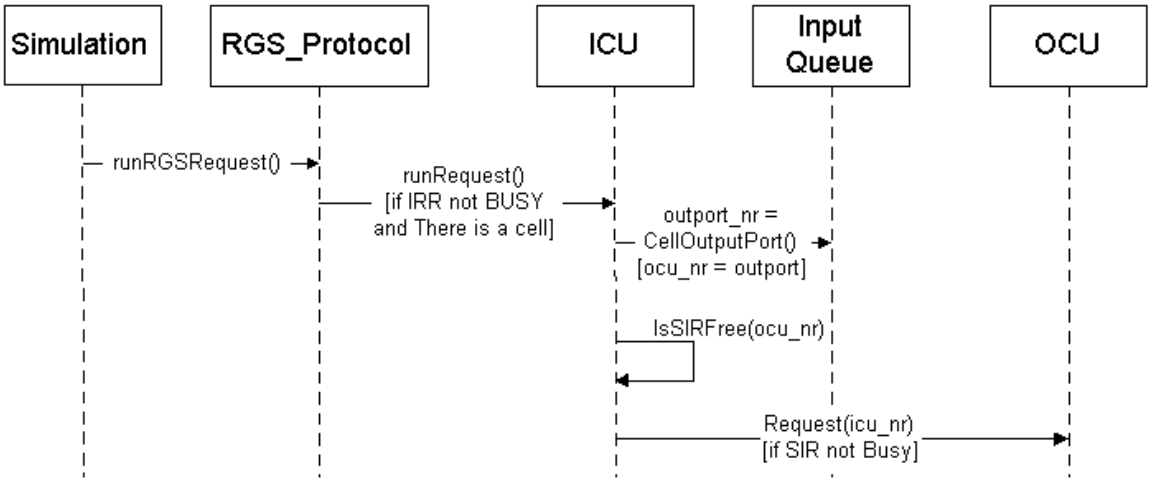

Fig. 10 Scenario Diagram for sending request (Request Phase)

Scenario Name: Selecting a request (Grant Phase)

Classes:

- Simulation
- RGS_Protocol
- ICU
- OCU

Preconditions: OCU has at least one request

Basic Course of Events:

- this scenario begins after all requests have been received by an OCU
- generate a uniform random number between 1 and N, where N is the number of requests received
- use the generated uniform random number as the index of an array that holds ICU number. therefore, the value stored in that array index represents the ICU whose request was granted.
- send a grant signal to the selected ICU unit.

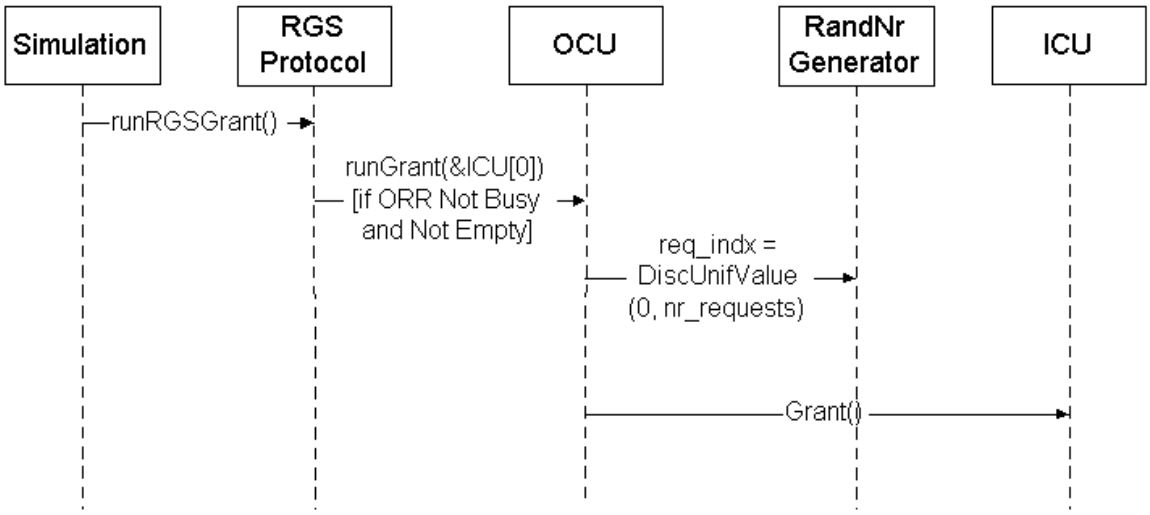

Fig. 11 Scenario Diagram for selecting request (Grant Phase)

Scenario Name: Sending status signal (Status Phase)

Classes:

- Simulation
- RGS_Protocol
- OCU
- ICU

Preconditions: none

Basic Course of Events:

- this scenario begins after the grant phase
- if the ORR of the current OCU is busy
  - notify all the other ICU units that the current OCU is busy.
- else do nothing

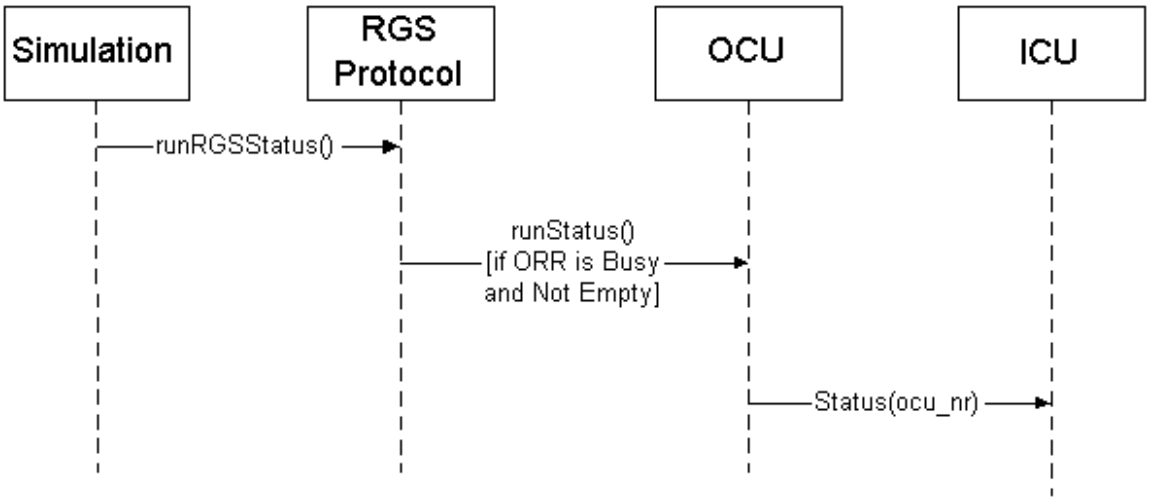

Fig. 12 Scenario Diagram for the Status Phase

Scenario Name: Routing ATM cells

Classes:

- Simulation
- RGS_Protocol
- ICU
- Input Queue
- Statistical Probe

Preconditions: at least one cell was chosen to be routed in the grant phase

Basic Course of Events:

- this scenario begins at the end of the last iteration
- remove selected cell from respective input queue.
- if current time is greater than transient time,
  - subtract current time from arrival time in the cell, and keep track of the result in the delay attribute of the Statistical Probe
- increment number of cells routed in the Statistical Probe

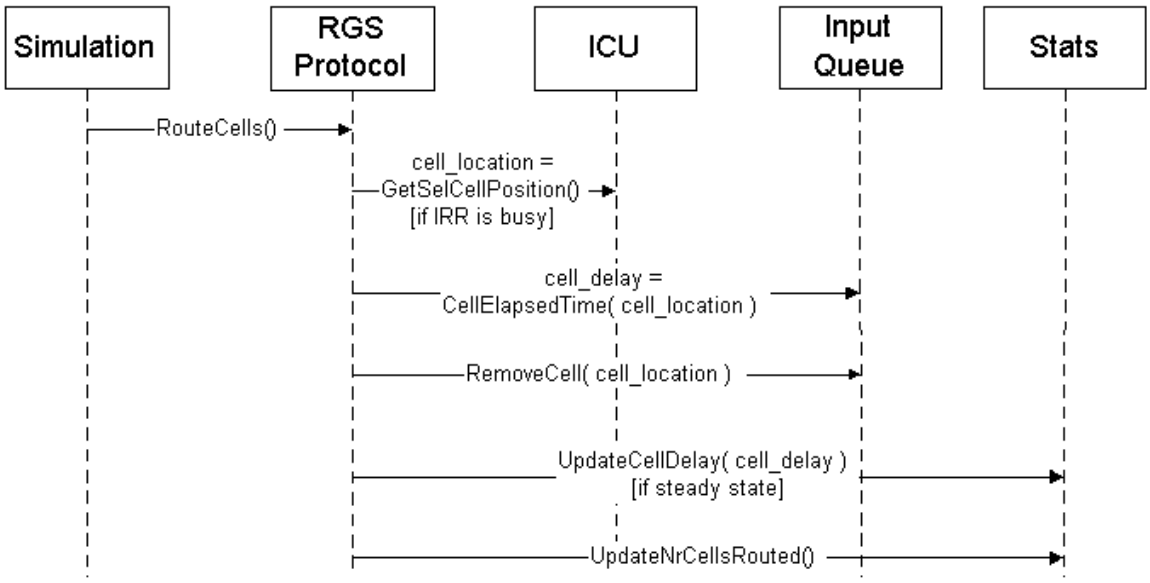

Fig. 13 Scenario Diagram for routing cell

Scenario Name: Resetting ICUs and OCUs (Maintenance Phase)

Classes:

- Simulation
- RGS_Protocol
- ICU
- OCU

Preconditions: none

Basic Course of Events:

- this scenario begins after the routing cycle
- reset all the registers: IRR, ORR, SIR
- reset memory storage in the DRC (Decision Reservation Control) of each OCU unit

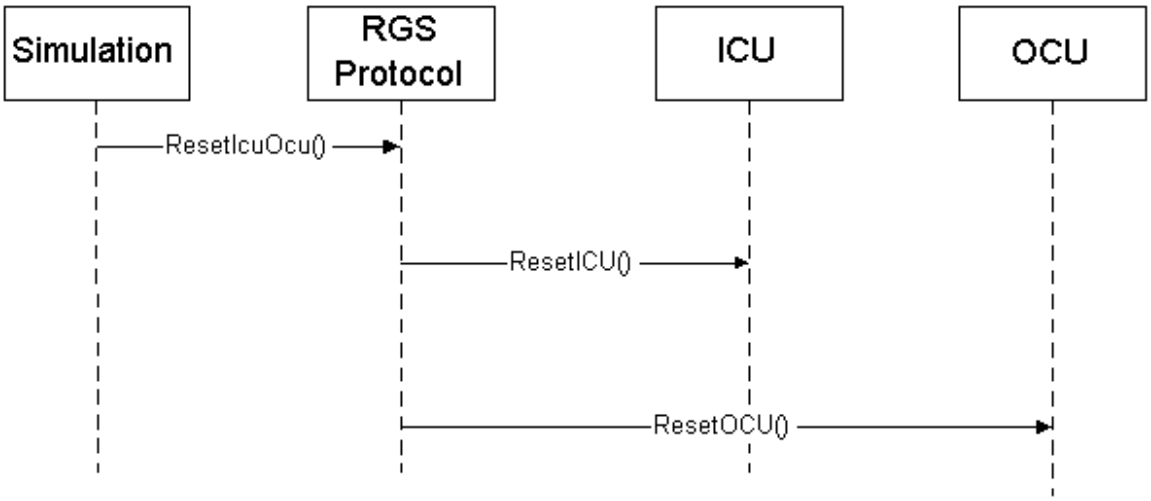

Fig. 14 Resetting all ICU and OCU registers and memory storage

The following AOM (Analysis Object Model) diagram illustrates the fundamental classes and relationships in the simulation system. The cardinalities of some relationships are illustrated, where n means 1 or more, and 1 means exactly one. If a cardinality is not shown, it is assumed to be 1.

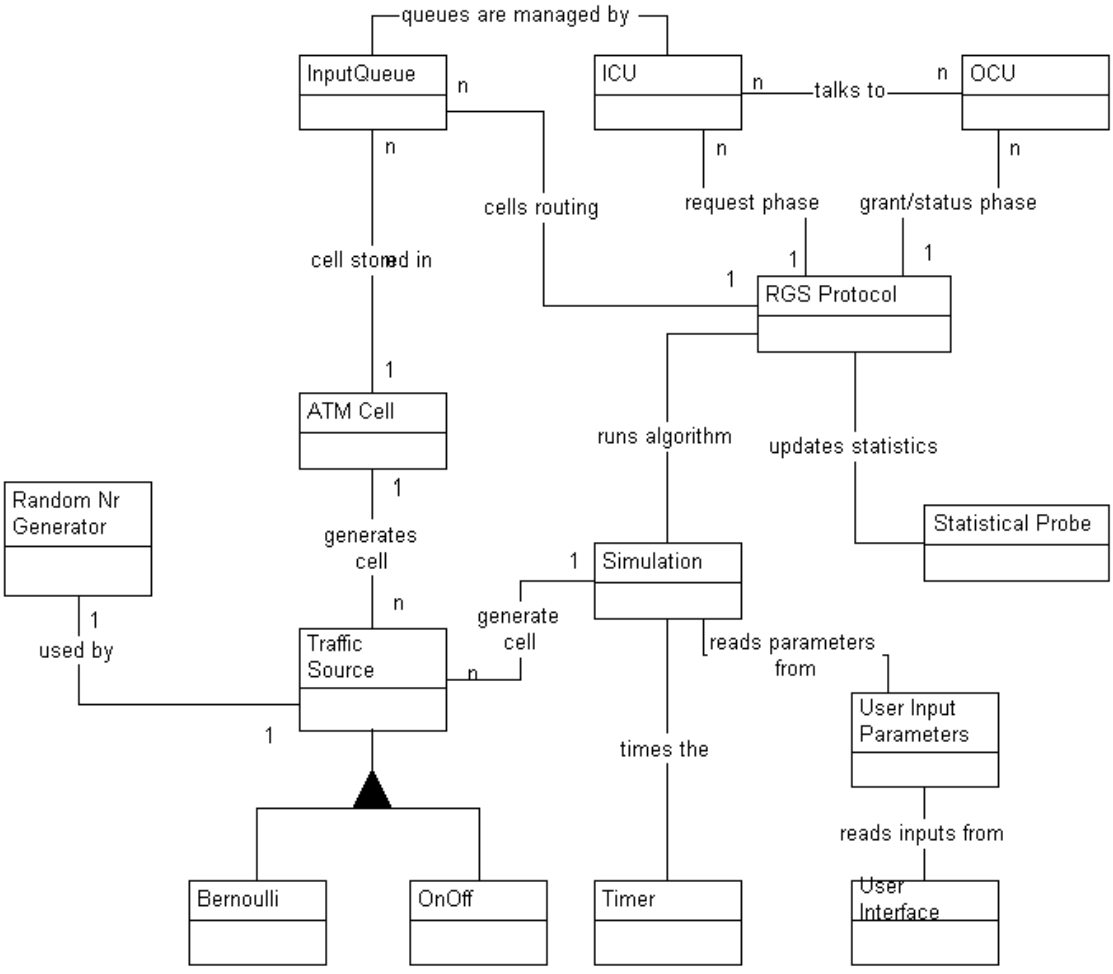

Fig. 15 Analysis Object Model for Simulation

Fig. 16 is a conceptual object model diagram of the ATM RGS switch. This diagram emphasizes the main objects to be implemented by the simulation program, and the message between objects. Objects in the model are represented by circles, and the interaction among them by arrows. A double arrow indicates that messages are sent both ways, as in the interaction between ICUs and OCUs. A brief description of the message operations are shown next to the arrows.

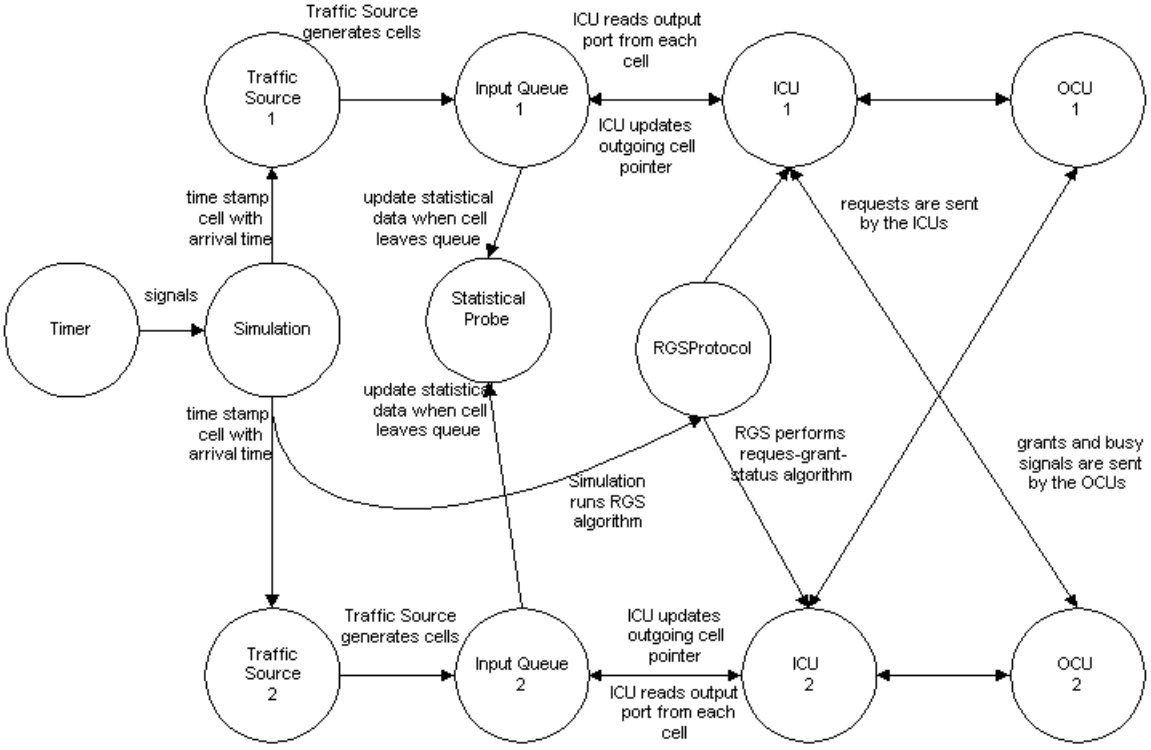

Fig. 16 Conceptual Model of a 2X2 ATM switch

The following object interaction graphs show the dynamics of the analysis object model. Messages between objects are displayed, along with the object collaborations required to fulfill a specific operation. These Object Interaction Graphs (OIGs) have been mapped to C++ code. The numbers in parenthesis mean the order that the operation is executed, and the asterisk represents a repetitive operation. A collection of objects is shown by dashed boxes. The Fusion graphical notation [7] has been used to construct the object interaction graphs, and C++ language notation has been used to name the interactions.

Fig. 17 and Fig. 18 show the operations that the simulator must execute when it is first started. As can be seen from the figures below, several objects are created at the system start-up.

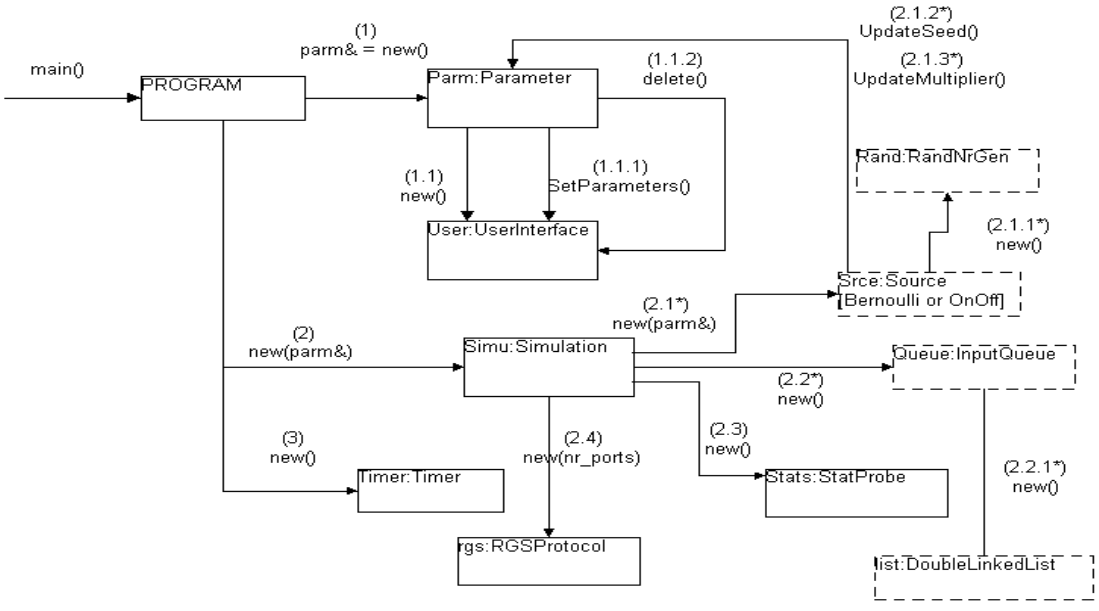

Fig. 17 Interaction Graph for Initializing Simulation

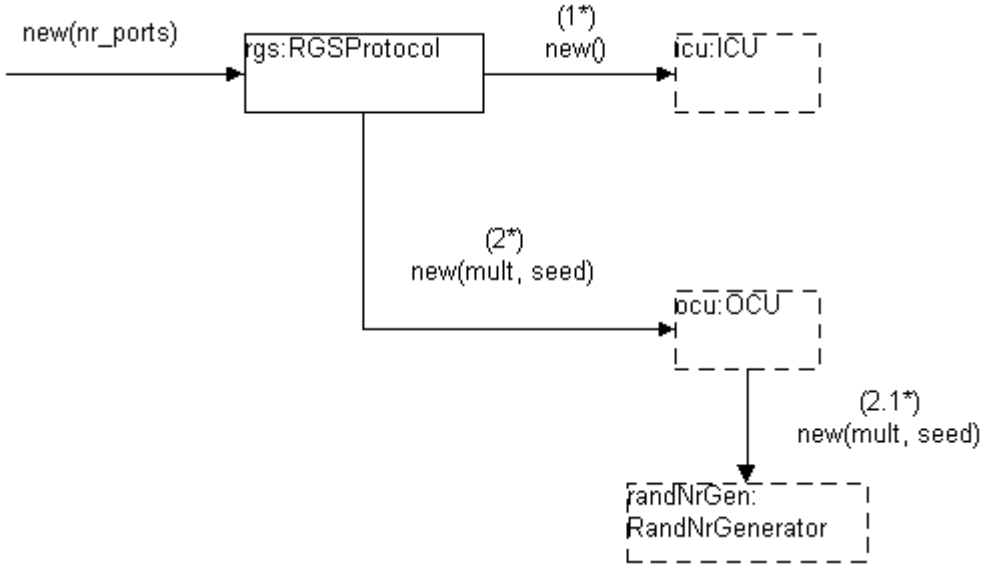

Fig. 18 Interaction Graph for Initializing the RGS protocol

The following object interaction diagram shows the interaction between the timer and the simulation object at the start of each cell slot. The timer acts as a clock circuitry for the ATM switch simulator.

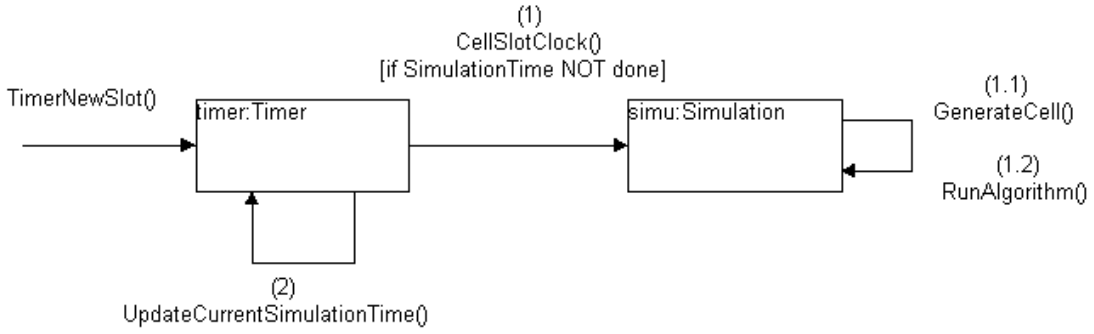

Fig. 19 Object Interaction Graph for Timer Clock

A cell is generated at the beginning of each cell slot cycle. The next interaction graph displays the collaborations among objects to generate an ATM cell in the simulation system.

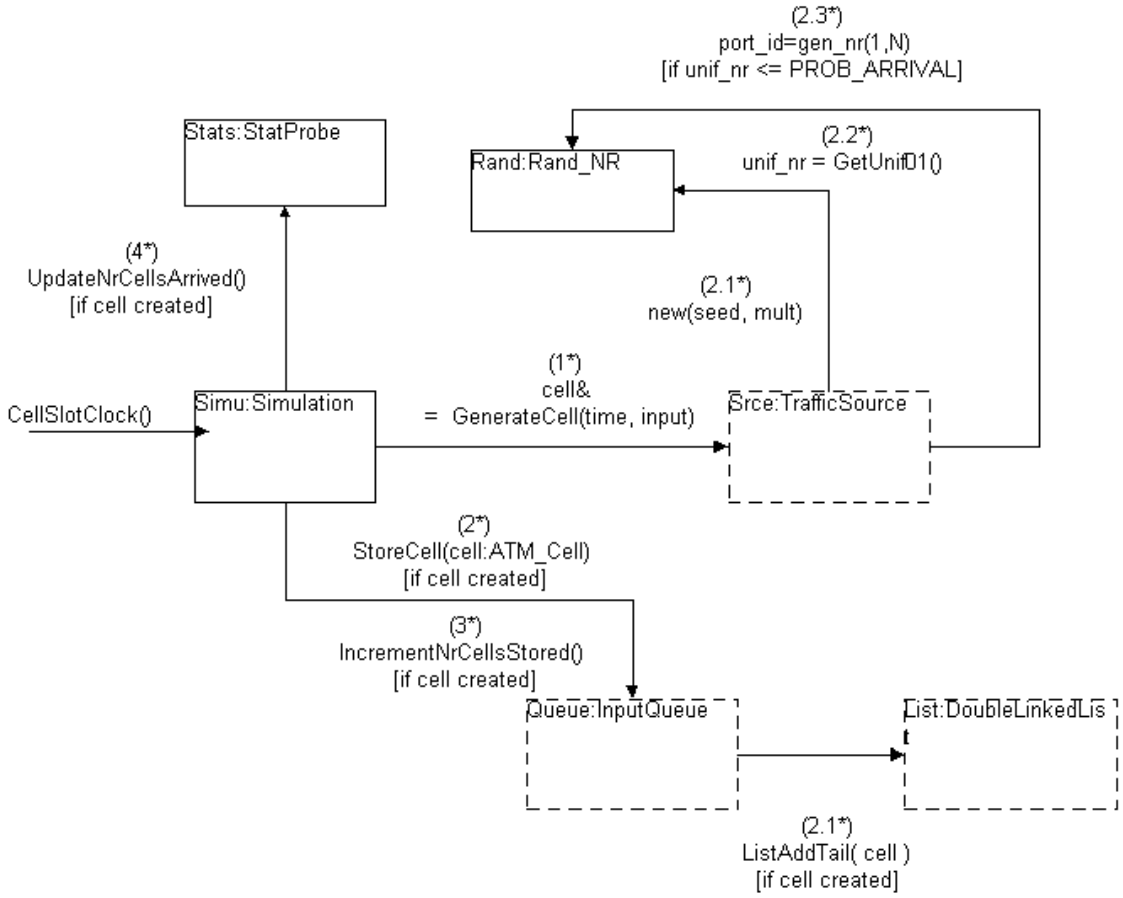

Fig. 20 Interaction Graph for Generating ATM Cell

During the Request phase, ICUs send requests to OCUs units. Requests are only sent for those cells whose output port (OCU) Status Information Register is not set (not Busy). The following interaction graph illustrates the dynamics of the Request phase.

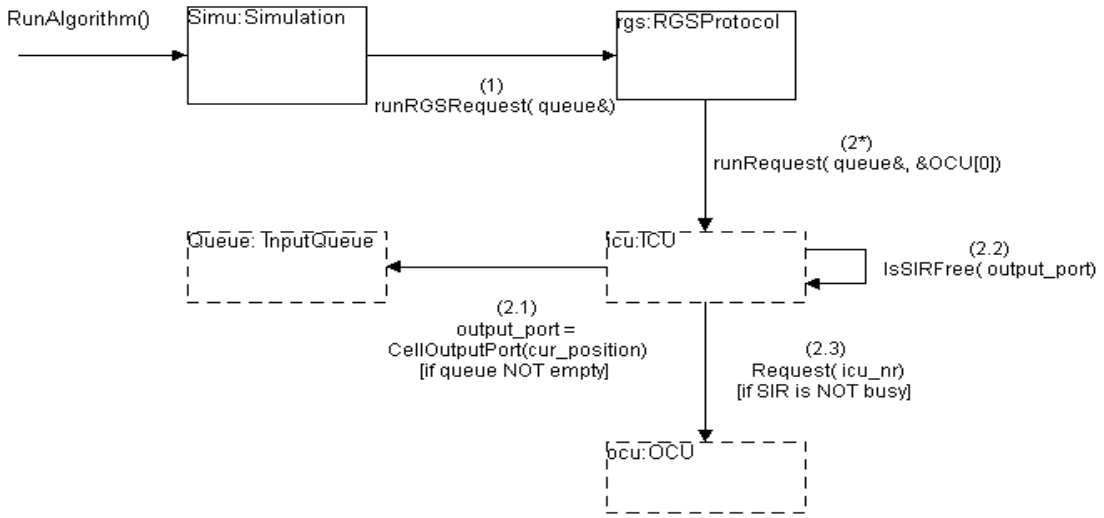

Fig. 21 Interaction Graph for the Request Phase

In the Grant phase, those OCUs which have pending requests, randomly select one among n requests. Then, a grant signal is sent by that OCU to the corresponding selected ICU unit.

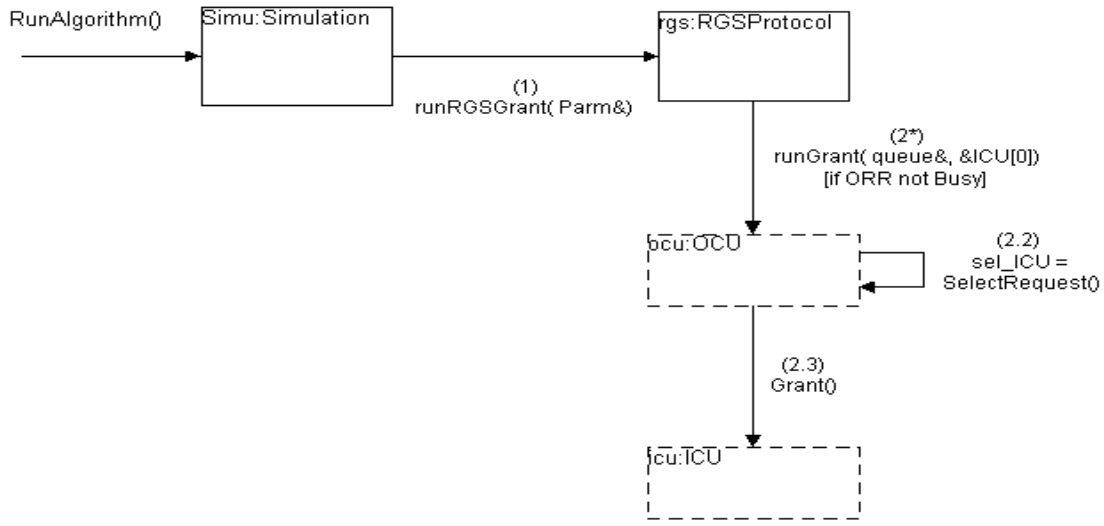

Fig. 22 Interaction Graph for the Grant Phase

In the status phase, a matched OCU notifies all ICUs of its current status by sending a busy status signal to each ICU. Therefore, a status signal is only sent if the ORR register of an OCU unit is set.

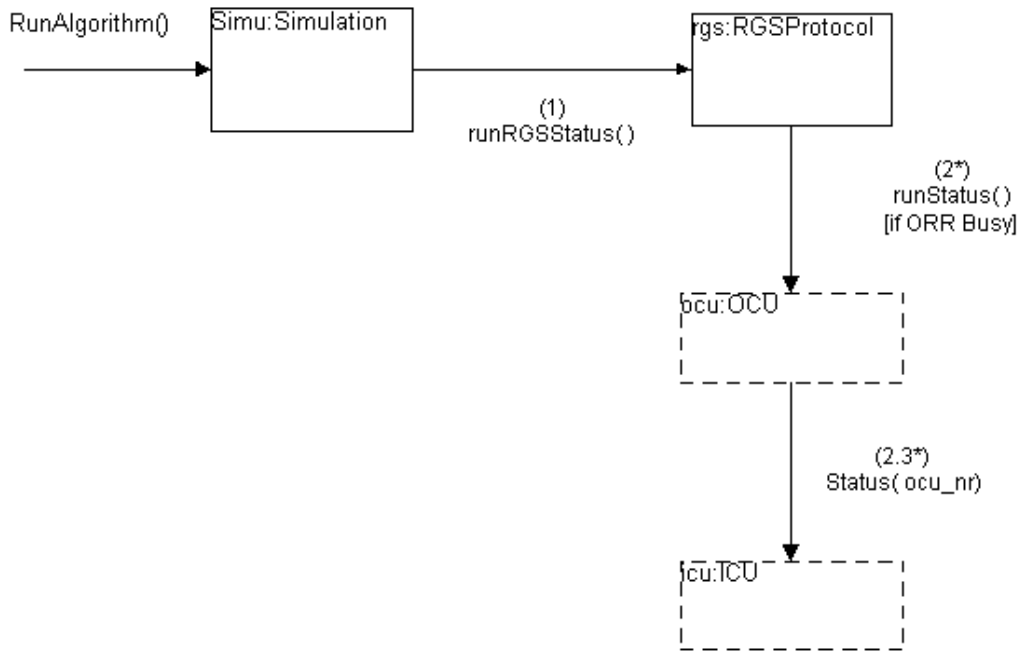

Fig. 23 Interaction Graph for the Grant Phase

A responsibility driven design approach helped delegate messages to appropriate objects in the simulation system. Each software object in the ATM switch simulation program is responsible for a specific task. The traffic source is in charge of generating

cells at each cell slot. The input queues are responsible for storing ATM cells. The statistical probes keep track of performance data information. And the ICU and OCU are the main objects which drive the scheduling algorithm. Following is a detailed explanation of each class used in the Simulation program.

The main responsibility of the Timer object is to keep track of the current simulation time, and to send a signal to the simulation time at the start of each cell slot. This signal tells the simulation object that it is time to process a cell slot cycle: generate cells, and run the RGS algorithm.

The ATM cell object contains four attributes: input_port, output_port, arrival_time, and departure_time. The first three attributes are set when a cell is generated. The last attribute is set when a cell leaves the input queue. The difference between the departure_time and arrival_time is used to keep track of the switch average cell delay.

The simulation is a central object which distributes responsibilities in the simulation system. The simulation object controls the RGS protocol algorithm. At the beginning of each cell slot, the simulation delegates responsibilities for several objects. Firstly, the simulation object requests the traffic source that a cell be generated. Then, the Simulation object stores the cell in the right input queue, and runs the RGS algorithm. Three messages are sent to the RGS object respectively, runRGSRequest, runRGSGrant, runRGSStatus. The simulation object is activated by the Timer object at the beginning of a cell slot.

The User Interface object is responsible for providing an textual interface to a user to enter simulation parameter values. At the start of the simulation program, a user interface object is created. This object displays textual messages in the screen, and collects input values entered by the user. The User Interface object collaborates with the Parameter objects. All the parameters entered by the program user are passed to the Parameter Object.

The statistical probe class contains the data and methods which are used to calculate the switch throughput and average cell delay. Every time a cell is routed the Statistical Probe object updates the number of cells routed, and the cell delays. At the end of the simulation, this object is called to calculate the switch throughput and switch average cell delay. The following formulas are used by the Statistical Probe object to calculate the throughput and average cell delay:

$$\rho = \frac{\sum cells\_routed}{\sum cells\_arrived}$$

Eq. 1 Throughput Formula

$$\Delta T = \frac{\sum cell\_delays}{\sum cells\_routed}$$

Eq. 2 Switch Average Cell Delay Formula

At the start of each simulation run, the Simulation Parameter object collaborates with the User Interface object to collect simulation parameters. Some of the parameters which are entered by the user are:

- type of traffic source: Bernoulli or On-Off
- switch size: the size of the switch can be any value from 2 to 100. This value represents the number of input and output ports for the ATM switch.
- number of iterations: this parameter is used to determine how many iterations the RGS scheduling algorithm performs before cells are routed through the switch fabric.
- offered load: a number between 0 and 1.
- average burst length (OnOff source only)

The input queue object is responsible for storing cells in the ATM input buffered switch. The queues are implemented by a doubly linked list. A new node is added to the end of the linked list every time a new cell is generated. Nodes are deleted as the cells leave the input buffers. Each input port in the switch has its own input queue. Therefore, a total of N (N, switch size) input queue objects are created at the start of the simulation.

The ATM RGS switch consists of N independent traffic sources, where N is the size of the switch. There is one traffic source per input port, and its type can be either a Bernoulli or a Discrete On_Off. The traffic source type to be used for a simulation run is selected by the user at the start of the program.

The Bernoulli traffic source generates cells according to the Bernoulli discrete random process. A single parameter, p (probability of arrival), is required for the source. The offered load per input port is equal to the probability of arrival, p.

$$\text{Prob (1 arrival)} = p$$

Eq. 3 Probability of Arrival

$$\text{Prob(0 arrivals)} = 1 - p$$

Eq. 4 Probability of no arrival

The Bernoulli source is a memoryless process, which means that knowledge of a past arrival does not help predicting the current or future arrivals [9].

The On-Off source is a bursty ATM source model of a discrete-time, 2-state (ON or OFF) Markov process. When the traffic source is in the ON state an ATM cell is generated at the beginning of a cell slot. If the source was in the ON state in the previous cell slot, the same output port number is used. If the source was in the OFF state in the previous state, then an output port is uniformly selected with a probability of 1/N. When the source is in the OFF state, no cell is generated. The On or OFF state probabilities are derived from the average cell burst length and offered load. The state diagram for the traffic source and the equations used to derive the average burst length and load are presented next.

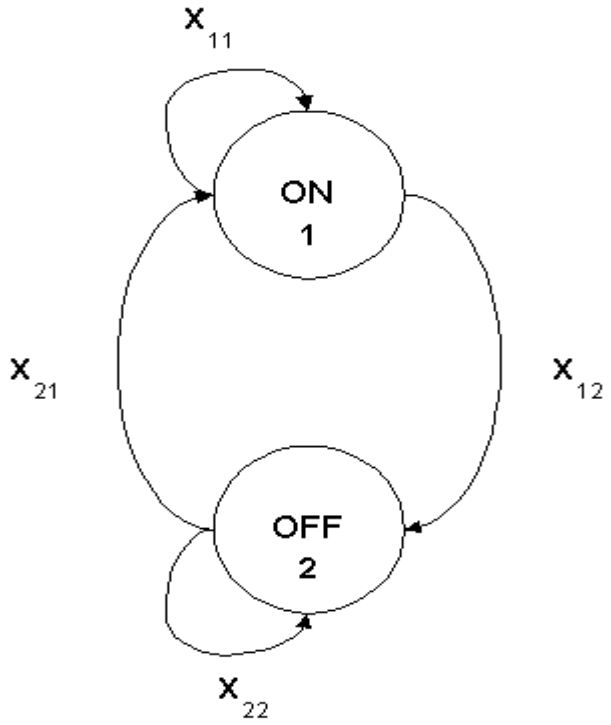

Fig. 24 State Diagram for the On-Off Traffic Source

The Markov Chain shown in Fig. 24 can be represented by the following probability matrix:

$$P(1) = \begin{vmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{vmatrix}$$

Eq. 5 Initial State Probability Matrix

where,

X_{11}: Transition Probability from ON to ON

X_{12}: Transition Probability from ON to OFF

X_{21}: Transition Probability from ON to OFF

X_{22}: Transition Probability from OFF to OFF

Assuming a homogeneous Markov chain, the state probabilities after many transitions become:

$$\pi^{T} = \pi^{T} \times P(1)$$

Eq. 6 Limiting State Probability

where,

$$\pi = \begin{bmatrix} \pi_1 \\ \pi_2 \end{bmatrix}$$

Substituting in Eq. 5, follows:

$$\begin{bmatrix} \pi_1 \pi_2 \end{bmatrix} = \begin{bmatrix} \pi_1 \pi_2 \end{bmatrix} X \begin{bmatrix} x_{11} x_{12} \\ x_{21} x_{22} \end{bmatrix}$$

Eq. 7 Limiting state probabilities matrix multiplication

where,

π_1 = Prob. On State when n ~ ∞

π_2 = Prob. Off State when n ~ ∞

After performing the above matrix multiplication, the steady state (or Limiting) probabilities are:

$$\pi_1 = \pi_1 * X_{11} + \pi_2 * X_{21}$$

Eq. 8 Steady On State probability

$$\pi_2 = \pi_1 * X_{12} + \pi_2 * X_{22}$$

Eq. 9 Off Steady State probability

Using the property of sum of probabilities, we have:

$$\pi_1 + \pi_2 = 1 \qquad (i)$$

or,

$$\pi_2 = 1 - \pi_1 \qquad (ii)$$

Substituting (ii) in Eq. 8, we have

$$\pi_1 = \pi_1 \,.\, X_{11} + X_{21} - \pi_1 \,.\, X_{21}$$

or

$$\pi_1 = \pi_1 \,.\, (1 - X_{12}) + X_{21} - \pi_1 \,.\, X_{21}$$

Eq. 10 On Steady State probability

For steady state operation, the average offered load is the probability that the traffic source is in the ON state:

$$\text{Offered Load} = \gamma = \pi_1 = \text{Limiting On State Probability}$$

Eq. 11 Offered Load

The average burst duration (length) is the expected number or mean of the geometric distribution that describes the duration of a burst in the slot “i”. This geometric distribution is determined as follows:

Assume that we are in the ON state, then the question becomes "what is the probability that at slot 'i' the system changes to the OFF state?" That is,

Prob(changes to OFF state at slot "i") = Prob (changes back to ON state at "i-1" previous slots) AND Prob (changes from ON to OFF state at slot i).

Assuming independence between the events "changes back to ON state", and "changes to OFF state", we have:

Prob(burst ends in slot i) = Prob(changes back to ON in slot '1') * Prob(changes back to ON in slot '2') * ... * Prob(changes back to ON in slot 'i-1') * Prob(changes to OFF state in slot 'i) = $[X_{11}\text{^}(i-1)] * X_{12}$. The '^' means "to the power of".

but $X_{11} = 1 - X_{12}$, therefore:

Prob(burst ends in 'i' slot) = $[1 - X_{12}\text{^}(i-1)] * X_{12}$

The mean of the above geometric distribution is "$1/X_{12}$". Then:

$$\delta = \frac{1}{X_{12}} = \frac{1}{(1 - X_{11})}$$

Eq. 12 Average Burst Length

From equations Eq. 10, Eq. 11 and Eq. 12 the transition probabilities (X_{11}, X_{12}, X_{21}, X_{22}) are determined as a function of the offered load and average burst length.

δ: Average Burst Length

γ: Average Offered Load

X_{12}: Prob. of On-to-Off

X_{11}: Prob. of On-to-On

X_{21}: Prob. of Off-to-On

X_{22}: Prob. of Off-to-Off

The following transition probabilities X_{12} and X_{11} can be derived from Eq. 12:

$$X_{12} = \frac{1}{\delta}$$

Eq. 13 On-to-Off Transition Probability

Then, since $X_{11} = 1 - X_{12}$, we have:

$$X_{11} = 1 - \frac{1}{\delta}$$

Eq. 14 On-to-On Transition Probability

The Off-to-On transition probability, X_{21}, is determined by substituting Eq. 13 or Eq. 14 in Eq. 10:

$$X_{21} = \frac{\gamma}{\delta(1-\gamma)}$$

Eq. 15 Off-to-On Transition Probability

Similarly,

$$X_{22} = 1 - \frac{\gamma}{\delta(1-\gamma)}$$

Eq. 16 Off-to-Off Transition Probability

The range of acceptable values for the offered load is:

$$0 < \gamma < 1$$

Eq. 17 Acceptable Range for the Offered Load

The minimum acceptable value for the average burst length is determined from equation Eq. 16, as follows:

$$0 < \{X_{22} = 1 - \frac{\gamma}{\delta(1-\gamma)}\} < 1$$

Then,

$$\delta > \frac{\gamma}{(1-\gamma)}$$

Eq. 18 Minimum Value for Avg. Burst Length as a function of Avg. Load

The random number generator class used by the RGS simulator was originally developed in [12]. The object instantiated from the RandomGenerator class is responsible for generating a uniform random number between "lower_bound" and "upper_bound. This uniform random number is used to determine if a cell is generated, and to select an output port for a new cell or burst of cells. In addition, the Random Number Generator is used to determine the next state of an OnOff source. In the Bernoulli traffic source, a cell is generated when a uniform random number between 0 and 1 is less than the probability of success. In the OnOff traffic source, a new cell is generated if the source is in the ON

state or if the source just transitions from the Off to the On state. Therefore, a uniform number between 0 and 1 is used to determine the current state of the source. If this uniform number is greater than the prob.ON_to_ON, and the source is in the ON state, the state is changed to the OFF state. If the source is in the OFF state, and the uniform random number is greater than the prob.OFF_to_OFF, the source changes to the ON state. Otherwise, the source remains in its current state.

The next seed for the random number generator is determined by the following equation.

$$seed(i+1) = (multiplier * seed(i)) \bmod (2^{31} - 1)$$

Eq. 19 seed formula

The above seed and multiplier are internal parameters defined in the simulation program. Each traffic source starts out with its own SEED and MULTIPLIER. The user of the RGS simulation program should select an initial seed, and a corresponding entry for a multiplier as shown below. This seed and multiplier are used as the initial parameters of the first traffic source object created in the simulation. Subsequent traffic sources have their seed and multiplier updated as explained in the Simulator User's Guide (please, see Appendix).

The entry for a multiplier selected by the user has the following correspondences:

Selected Entry	Multiplier
1	950706376
2	742938285
3	1226874159
4	62089911
5	1343714438
6	630360016

7	397204094
8	16807

To generate a new uniform random number from the new seed derived in the above equation, the seed is normalized. The following formula describes how a uniform random number between 0 and 1 is generated by the random number generator:

$$U(0,1) = \frac{(seed(i+1))}{(2^{31}-1)}$$

Eq. 20 uniform random number

The selection of the initial seed is quite arbitrary when using just one generator because every seed value starts a stream which does not repeat for approximately two billion numbers [12]. In the simulation of the RGS ATM switch, each traffic source starts out with its own seed, and multiplier.

A uniform random number between "lower_bound" and "upper_bound" is determined by the following algorithm:

```
LargestInteger  =  ("upper_bound" - "lower_bound") + 1
LargestInteger  =  (int) [LargestInteger * U(0, 1)]
```

Then, the random number between "lower_bound" and "upper_bound" becomes:

```
RandNr    =  LargestInteger + lower_bound
```

The RGS Protocol object is responsible for performing the Request, Grant, and Status phases of the RGS algorithm. In order to perform this task, the RGS protocol collaborates with the ICU, and OCU objects. The RGS_Protocol object also creates the ICU and OCU objects during start-up.

The ICU is one of the main objects in the simulation system. Its responsibilities include sending requests to OCUs, and checking grant and status signals. In addition, the ICU is responsible for telling the RGS_Protocol object the location of the cells which are to be routed in the end of the iteration cycles.

The ICU object can be in one of the following states:

All ICUs in the ATM switch are in idle state at the start of each cell slot. An ICU, Input Control Unit, can only leave the idle state to another state if one of the following conditions is met when an iteration event occurs:

- the IRR, Input Reservation Register, is NOT busy, and there is at least one cell in the ICU input buffer. In this case, the ICU changes to the REQUEST OUTPUT state.
- the IRR is busy, and the ATM switch has just processed the last iteration. Then, the ICU changes to the ROUTING STATE.
- the IRR is NOT busy, and the ATM switch has just processed the last iteration. Then, the ICU changes to the MAINTENANCE STATE.

The IRR, Input Reservation Register, indicates whether a cell belonging to the current input port buffer had its request granted. If the IRR register is set to "1" (busy), the cell whose address is stored in the cell selection pointer is routed after the last iteration cycle (ROUTING STATE). All ICUs are forced to pass through a Maintenance state in the end of the last iteration. During the Maintenance state all the pointers, IRRs (Input Reservation Registers) and SIRs (Status Information Registers) are initialized, and the state of the ICU is reset to idle.

The diagram of Fig. 25 below describes the functional process to be performed by an ICU in idle state when an iteration event occurs. The ICU uses the SIR (Status Information Register) table of N elements, where N is the number of ports, to store the status of the ORRs, Output Reservation Register. If an entry in the SIR is "1", the ORR corresponding to that entry is busy, and no requests may be sent to that OCU.

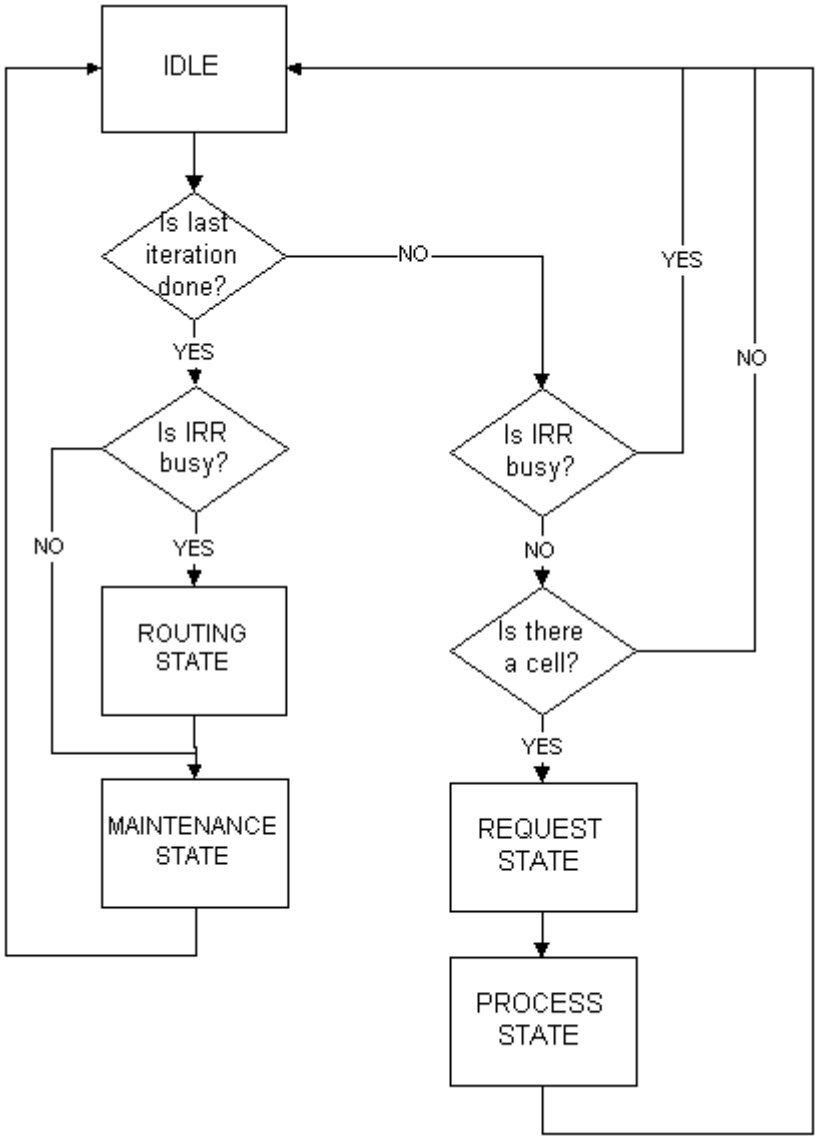

Fig. 25 ICU Functional Diagram in the Idle State

In this state, the ICU determines the status of the SIR, Status Information Register, corresponding to the output port NR of the cell in the head of the queue. If the SIR is free, a request signal is sent by the ICU to the corresponding OCU, Output Control Unit. Otherwise, the next cell in the queue is searched, until a free output port is found or the end of the queue is reached. In the simulation, a request message sent by an ICU to an

OCU corresponds to writing the input port NR (ICU_nr) to an OCU input processing table.

A functional diagram for the ICU in the request state is shown in Fig. 26.

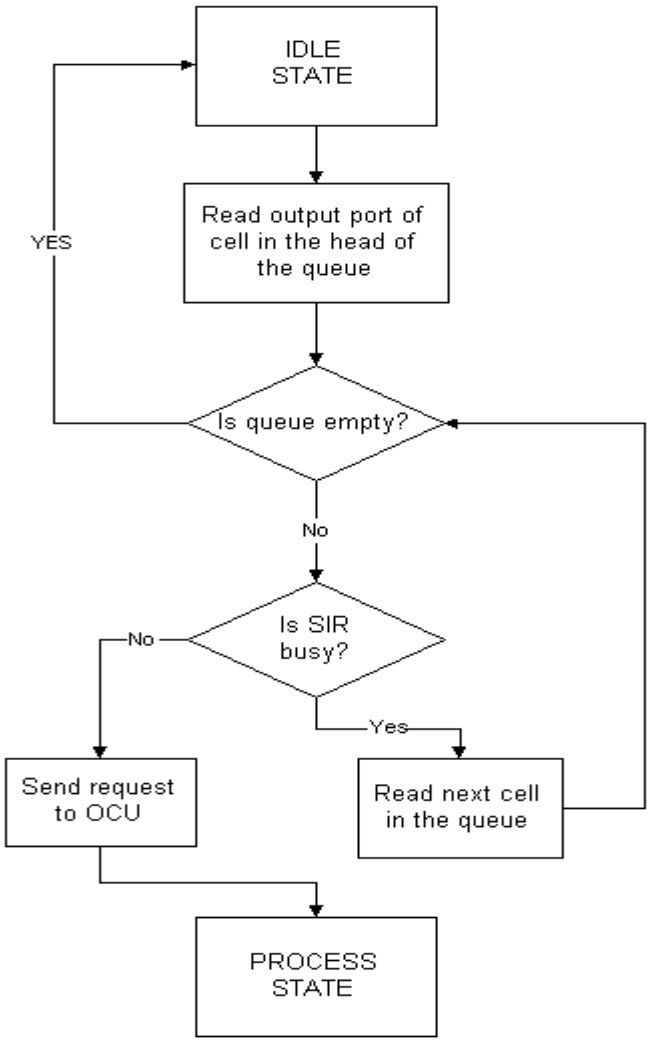

Fig. 26 ICU Request State

In the Process Response State, each ICU processes the grant signal sent by an OCU unit. This signal indicates whether a request sent by a particular ICU was accepted or not. If the grant signal is “1”, the IRR is set to BUSY (1), and the address of the selected cell is stored in the cell selection position pointer. If no grant signal is received by an ICU during the Grant phase, it is implied that the request was not granted, and nothing is done in the ICU unit. The next signal processed by the ICU is the Status signal. The status tells the ICU which OCU unit has been matched. After receiving the status signal, the ICU unit sets the corresponding Status Information Register.

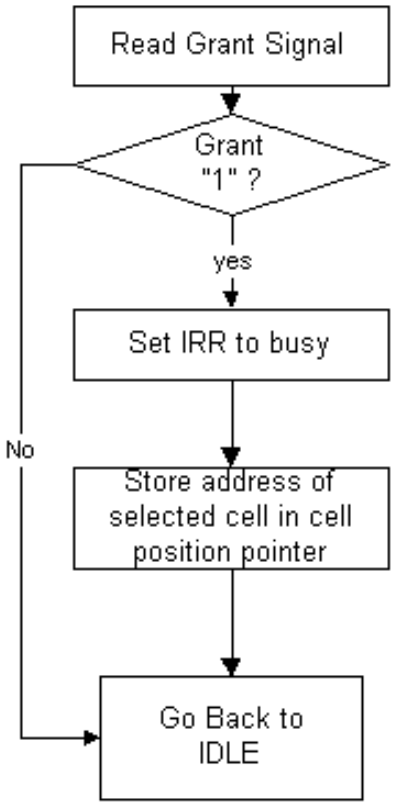

Fig. 27 ICU Process State

In this sate, those cells whose requests have been granted are routed through the switch. The address of the cell is determined by the cell selection position pointer. The routing state takes place right after the last iteration cycle. As the cell is removed from its input queue, a counter in the statistical probe is updated to reflect the total number of cells routed. Also, the current simulation time is subtracted from the cell arriving time in order to determine the delay for that cell. Then, the total delay is updated in the Statistical Probe object. After all the previous steps are done, the ICU unit changes to the maintenance state.

Fig. 28 shows a diagram of the ICU Routing State.

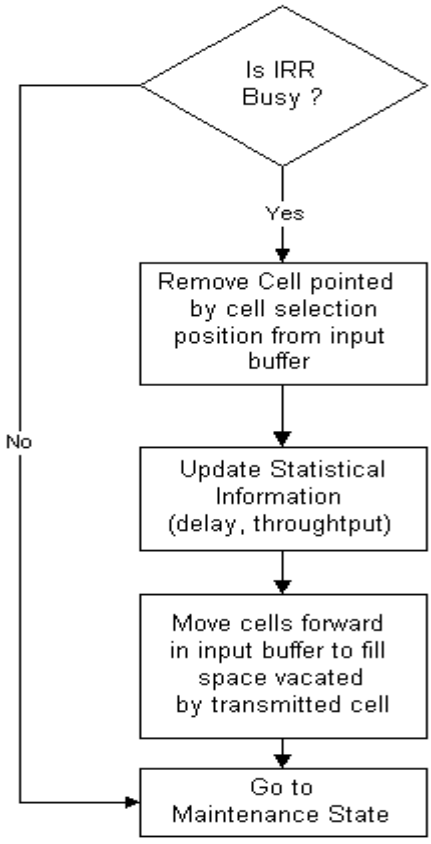

Fig. 28 ICU Routing State

As the name implies, each ICU should perform re-initializations in the Maintenance State. The following are the required resetting: reset IRR to ZERO (free), and reset the cell selection position pointer to NIL.

Fig. 29 shows the functional diagram for the Maintenance state of an Input Control Unit.

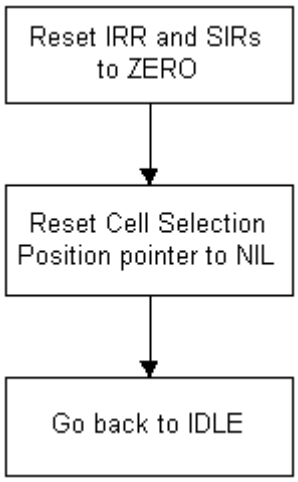

Fig. 29 ICU Maintenance State

The Output control unit is in charge of accepting and selecting requests, and sending grant and busy status signals to ICUs. The OCU performs a uniform random selection to choose one among N received requests received.

The OCU can be in one of the following states:

The OCU is in idle state until the beginning of the request phase. An OCU only changes to the Store/Process state if its ORR is not marked as reserved "1". In addition, the OCU does not receive requests in the Idle state if its ORR is reserved.

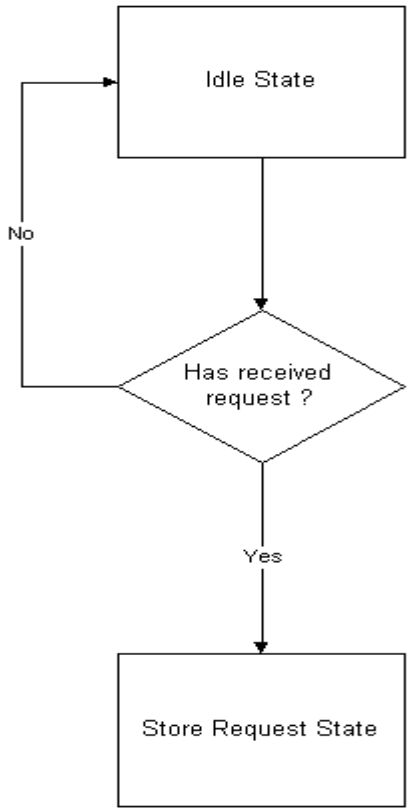

Fig. 30 OCU Idle state

In this state the OCU reads, and stores any request sent by an ICU unit. There must be at least one request because the OCU unit can only get to this state if a request was received in the IDLE state.

The OCU unit uses a uniform random generator to select one among N requests. The limit for N is the number of input ports.

After a request is granted, the OCU sets its ORR to “1”, or reserved, and sends a grant signal to the respective ICU unit. After the grant phase, those OCU which had their ORRs marked, send a busy status signal to all the ICUs in the ATM switch.

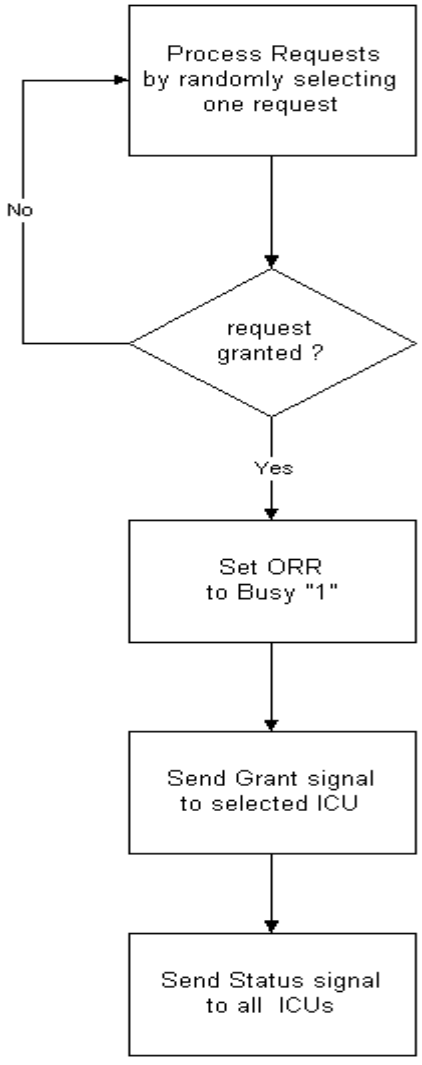

Fig. 31 OCU Greant/Status Phase

In the maintenance state, the Output Reservation Register (ORR) of each Output Control Unit is reset to “0”, or FREE.

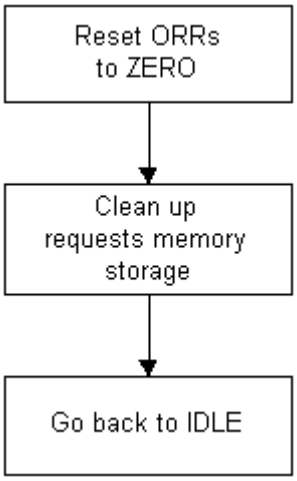

Fig. 32 OCU Maintenance State

Using the 'RGS" C++ simulator program developed in this study, simulation runs of an input buffered ATM switch were performed for loads of 50, 75, 90 and 100%. The traffic sources used were: a Bernoulli non-correlated source, and a bursty OnOff Markov source. The target of each simulation run was to determine the switch throughput and queueuing delay performance of the ATM switch. These performance parameters were determined in terms of the number of iterations used in the RGS algorithm, traffic source type, switch size, offered load, and average burst length (OnOff source). Simulation runs were conducted for switch sizes of 16x16, 32x32, 64x64. The number of iterations tested were 1, 2, 3, 4, and 5. The simulation results using a Bernoulli source, and saturated queues (100% load), were compared against the results of a theoretical performance analysis of the RGS algorithm published in [1].

The results presented in the next section were collected by running the "rgs" simulator program in the following computer architecture platforms: DEC Starion 915 Pentium-75 PC running Windows95, HP 9000/712 workstation running HP-UX UNIX operating system version 10.01, and a SUN4c SparcStation running SunOS release 4.1.3_U1. Queueing delay results were shown only when the switch throughput was greater than 98%. The input queues start to build up if throughput is less than approximately 98%,

causing cell delays to increase with time, which makes the cell delay results unstable. For an offered load of 50%, the switch throughput was essentially 100% for all iterations, and the corresponding throughput plots were not displayed. The total simulation time for each run was 50,000 cell slots.

The initial seed and multiplier values used for each simulation run were 1,000 and '630360016'. This multiplier value corresponds to entering '6' for multiplier at the start of program. (for further information on mapping between entry and actual multiplier used, please refer to the Random_Number_Generator class on Chapter 4) These seed and multiplier values are used for the random number generator of the first traffic source object created at simulation start-up. Subsequent traffic sources objects created by the simulator have their seed and multiplier values automatically updated as follows:

```
Seed of Traffic_Source[ n ] = Seed of Traffic_Source [ n - 1] + 1,000,000.

if ( Multiplier of Traffic_Source[ n-1] < 8 )
      Multiplier of Traffic_Source [ n ] = Multiplier of Traffic_Source [ n -1 ] + 1
else
    Mutiplier of Traffic_Source [ n ] = 0
end
```

The following figures Fig. 33 and Fig. 34 present the delay performance of three ATM input buffered switches running the RGS algorithm for a load of 50%. With this offered load, the simulation switch throughput is essentially 100% for either the Bernoulli or OnOff traffic sources. Therefore, the switch throughput plots are not shown for an offered load of 50%.

As it can be seen from the following two plots, the ATM switches have a better delay performance when the traffic is non-correlated (Bernoulli). For 1 iteration, a 16x16 switch has an average queueing delay of 7.88 usec for a Bernoulli source. When the traffic source used is bursty (OnOff), the same switch has a queueing average delay of 94.4 useconds for 1 iteration. Also, the size of the switch does not appear to have much impact on the delay performance the ATM switches with a Bernoulli source. For 1 iteration, a 64x64 switch has an average delay of 8.67 usec vs 7.88 for a 16x16 switch. When a correlated cell source (OnOff) is used, a 64x64 switch has an average queueuing delay of 141.6 usec versus 94.4 usec for a 16x16 switch.

The delay performance of the ATM switch has a significant improvement when the number of iterations in the RGS algorithm increases from 1 to 2 iterations. With the Bernoulli source, the average queueuing delay of a 16x16 switch is 4.78 useconds for 2 or more iterations. With the OnOff correlated source, the average delay is approximately 16.3 useconds for 2 or more iterations.

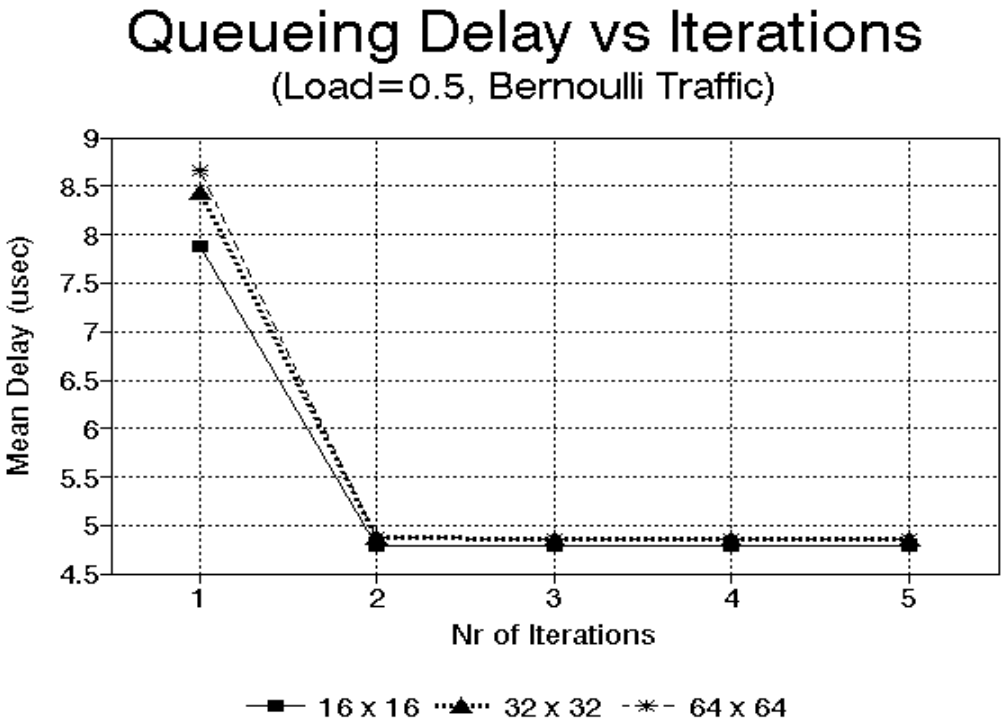

Fig. 33 Bernoulli: Queueuing Delay vs Iterations (load=0.5, switches: 16, 32, 64)

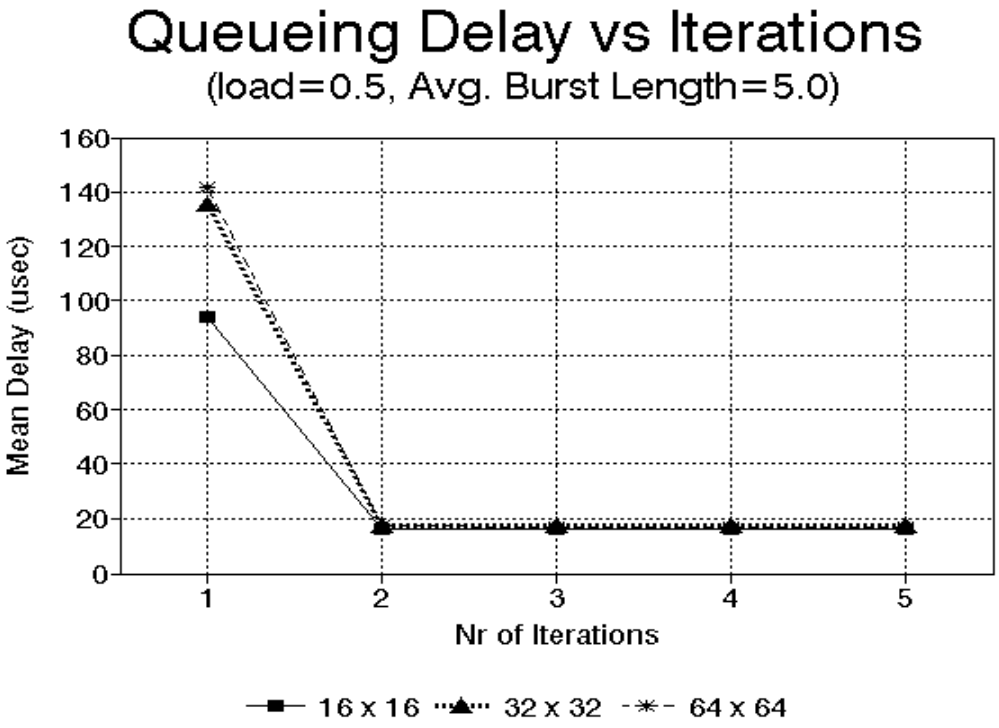

Fig. 34 OnOff: Queueuing Delay vs Iterations (load=0.5, length=5, switches: 16,32,64)

The figures Fig. 35 and Fig. 36 below show the switch throughput simulation results when the average offered load is 75%. As it can be seen from these figures, when the load is increased from 50% to 75%, the switches no longer have a throughput of 100% at 1 iteration. The switch throughput only achieves 100% throughput if 2 or more iterations are used in the RGS algorithm. When a Bernoulli source is used, the switches have a better throughput performance than with the OnOff correlated source. At 1(one) iteration, a 16x16 switch has 80.2% throughput performance with a Bernoulli source, versus 69.9% with the OnOff source. The size of the switch does not appear to have too much impact on the throughput performance. For a 16x16, 32x32, and 64x64 switch, Bernoulli source, and 1 iteration, the throughput results are 80.2%, 79.2%, and 78% respectively. If the OnOff source is used, the throughput simulation results are 69.9%, 68.9%, and 67.8% respectively. Therefore, the ATM switch has a slight throughput improvement of about 2% when the size changes from 64 x 64 to 16 x 16.

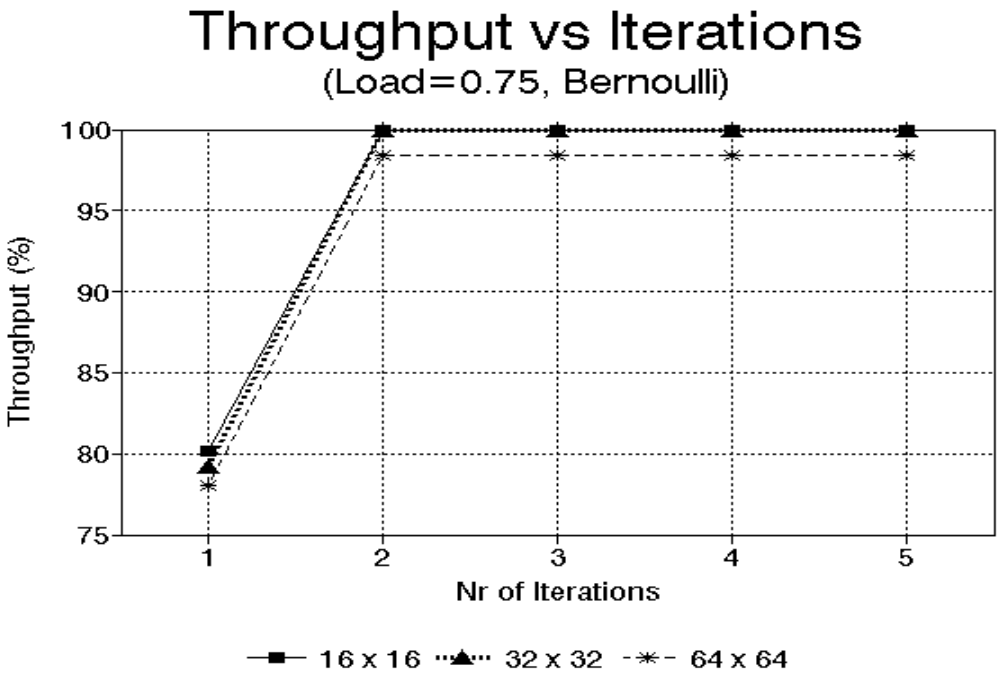

Fig. 35 Bernoulli: Throughput vs Iterations (load=0.75, switches: 16, 32, 64)

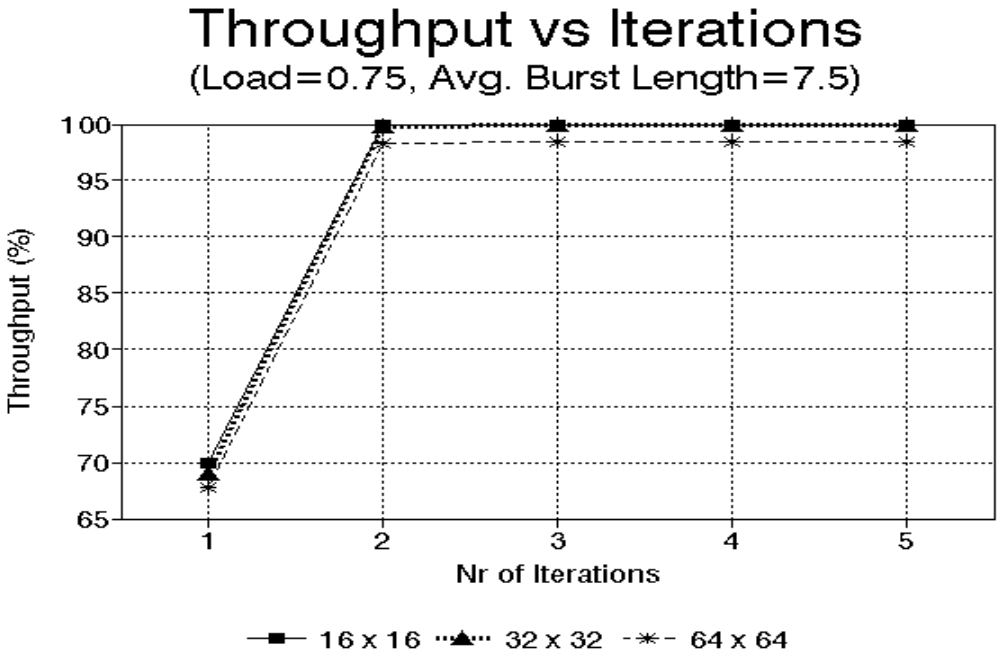

Fig. 36 OnOff : Throughput vs Iterations (load=0.75, length=7.5, switches: 16, 32, 64)

The next two plots (Fig. 37 and) show the delay performance of three ATM switches (16x16, 32x32, and 64x64) for an offered load of 75%. Notice that the delay results are only shown for 2 or more iterations. The reason for not showing delay results at 1 iterations is because the switch throughput at 1 iteration is less than 98%, which causes the delay to be unstable. With a Bernoulli source the average queueing delay increases almost twice as the offered load changes from 50% to 75%. When the OnOff source is used, the delay performance degrades much more rapidly. For a 16x16 switch, a burst source (OnOff), and 75% load, the average queueing delay at 2 iterations is 164.5 usec vs. 16.3 usec for a load of 50%. At 3 iterations, the delay performance of the same switch with a correlated source is approximately 84 usec vs. 16.3 usec with a load of 50%. Also, as it can be seen from the Fig. 38, the size of the switch has an impact on the delay performance at 2 iterations. The 64x64 switch has an average queueuing delay of 325 usec vs. 164.5 for a 16x16 switch at 2 iterations. Overall, the impact of the total offered load on the delay performance of the switches is more noticeable when the traffic source is correlated.

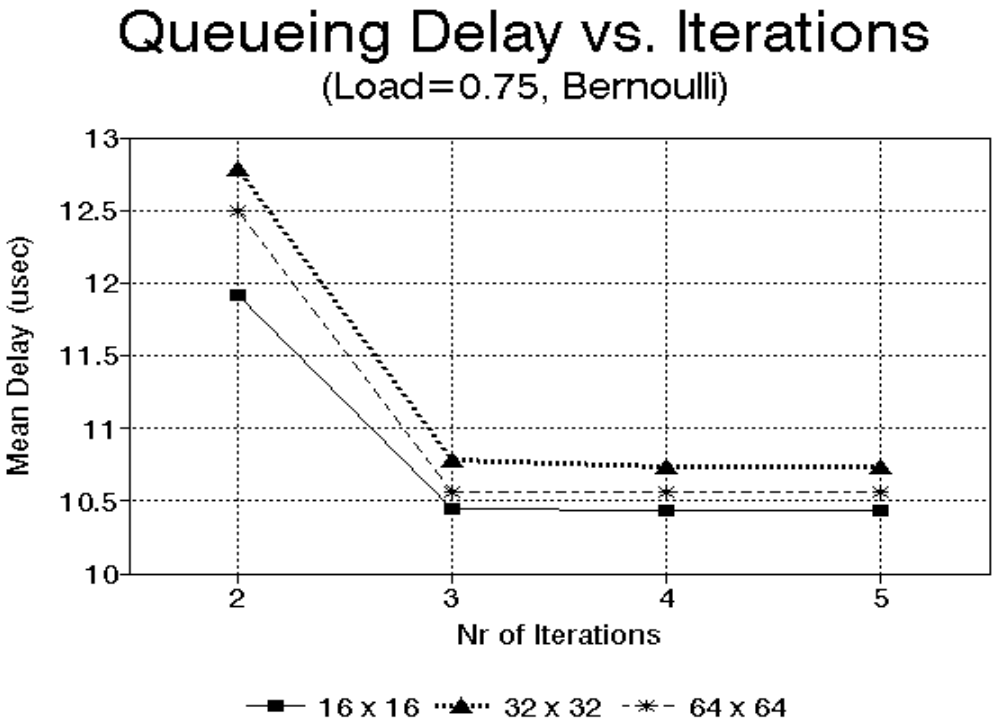

Fig. 37 Bernoulli: Queueuing Delay vs Iterations (load=7.5, switches: 16, 32, 64)

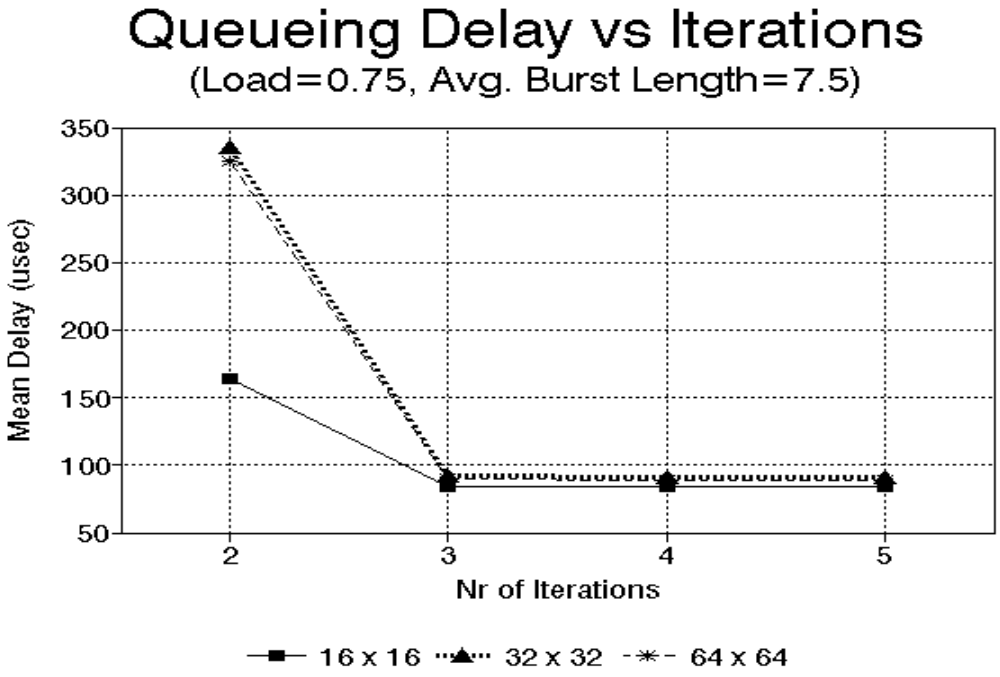

Fig. 38 OnOff: Queueuing Delay vs Iterations (load=7.5, length=7.5, switches: 16, 32,64)

The plots of figures and Fig. 40 show the switch throughput simulation results when the offered load is 90%. As it can be seen from the next two figures, a high offered load can have a significant impact on the switch throughputs for both the Bernoulli and OnOff traffics if the RGS algorithm uses 1(one) iteration. The simulation results show that by increasing the number of iterations used by the RGS algorithm the throughput performance of the ATM switches can be significantly improved. At 1(one) iteration, which means the RGS does not schedule any cells, the throughput of all switches with an OnOff source is about 57%, and 65% for the Bernoulli source. At 2 iterations the same switches have a throughput of 84% for OnOff, and 90% for Bernoulli source. At 4 iterations, the switches perform at a throughput of 98% for either traffic sources. With the number of iterations being less than 3 (three), the correlated source (OnOff) appears to have a higher impact on the throughput performance of all three ATM switches.

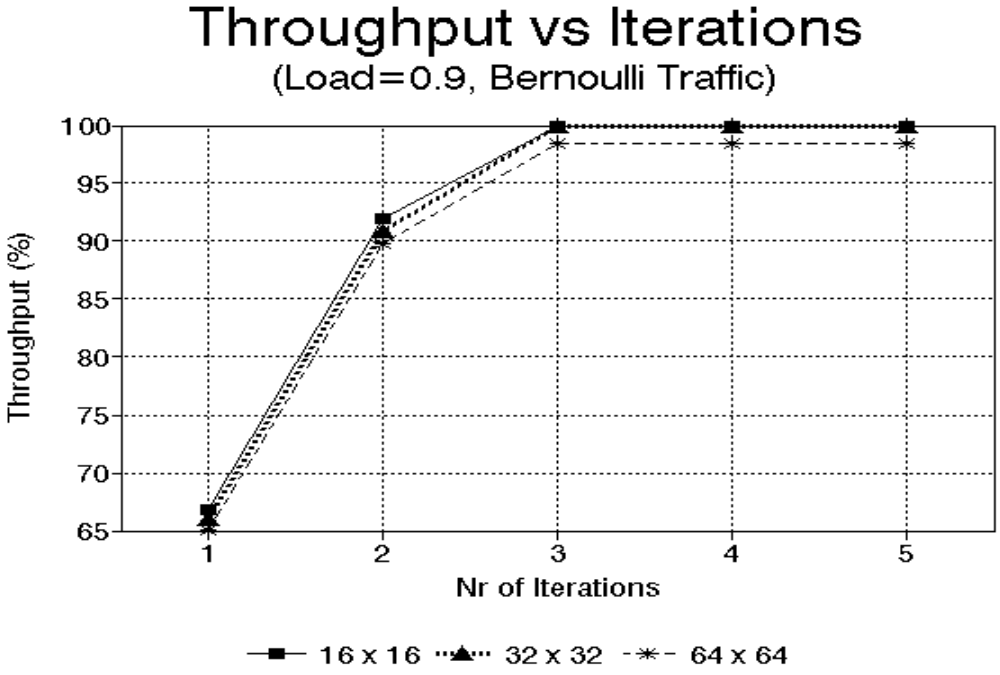

Fig. 39 Bernoulli: Throughput vs Iterations (load=0.9, switches: 16, 32, 64)

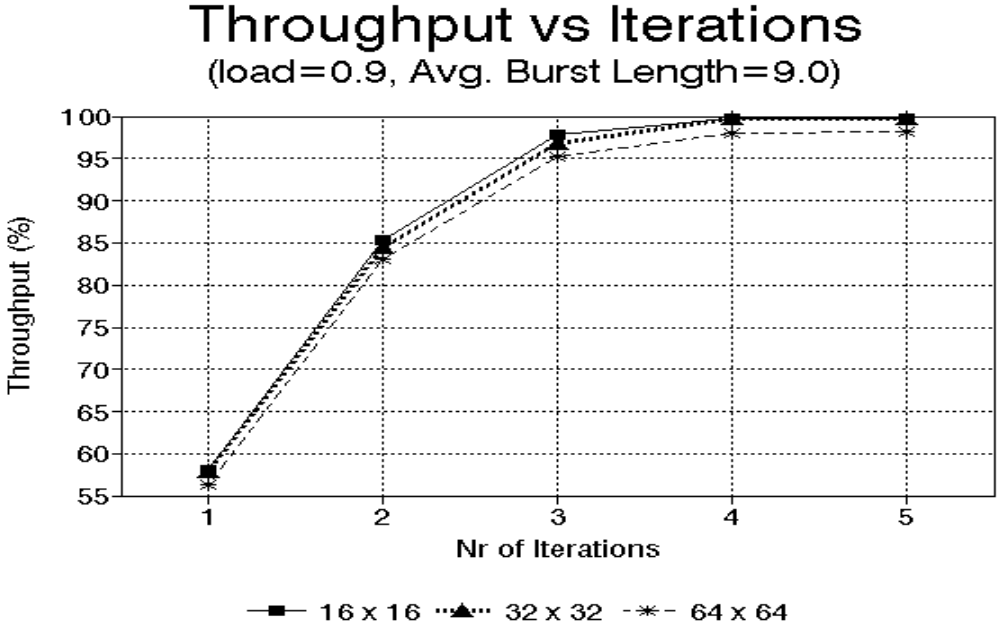

Fig. 40 OnOff : Throughput vs Iterations (load=0.9, length=9, switches: 16, 32, 64)

The last queueing delay plots presented in this simulation study are shown in Fig. 41 and . Delay results are only displayed for more than 3 itreations with a Bernoulli arrival process, and 4 iterations with the OnOff source. These number of iterations provide a switch throughput of 98% or above. As the offered load is increased from 75% to 90%, the average queueing delay becomes approximately three times as large with the Bernoulli source, and 4.5 larger with the OnOff source. With a 90% load, a 16x16 switch has an average queueing delay of approximately 38.7 usec at 3 iterations when the source is Bernoulli. At 4 iterations, the 16x16 switch under the same traffic conditions has an average delay of 29.8 usec, which shows an improvement of almost 10 usec. With the OnOff source, the delay performance is about 4.5 times worse as the load is increased from 75% to 90%. A 16x16 switch, having a correlated source with a load of 90% and an average busty length of 9 cells, has an average queueing delay of 364 usec at 4 iterations. At 75% load, and an average bursty length of 7.5 cells, the same switch has an average queueing delay of 83.6 usec. Interestingly enough, as it can be seen from Fig. 42, the delay performance of a 64x64 switch is slightly better than a 32x32 switch. At 4 iterations, the 64x64 switch with the OnOff source has an average delay of 422.3 usec vs. 435 usec for a 32x32 switch. At 5 iterations, the 64x64 switch has a delay of 372 usec versus 390 usec for the 32x32 switch. A second simulation run was conducted using the same load and average bursty length parameters, but with initial seed and multiplier values equal to 2 and '742938285'. This multiplier corresponds to entering "2" for multiplier at the start of simulation. The results of the second simulation run show a 64x64 switch having an average delay of 369 usec at 4 iterations versus 453 usec for a 32x32 switch. At 5 iterations, a 64x64 switch has a delay of 351 usec versus 422 usec for a 32x32 switch. Therefore, from the two simulation runs, a 64x64 switch appears to have a slightly better delay performance than a 32x32 switch when the traffic is correlated, and the offered load is 90%.

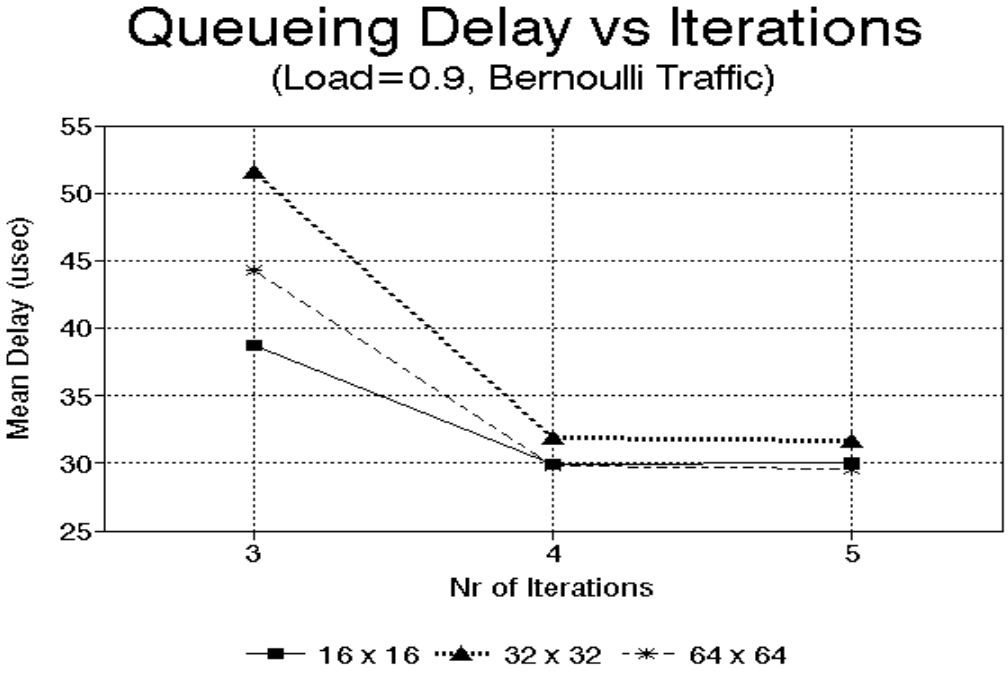

Fig. 41 Bernoulli: Queueuing Delay vs Iterations (load=9, switches: 16, 32, 64)

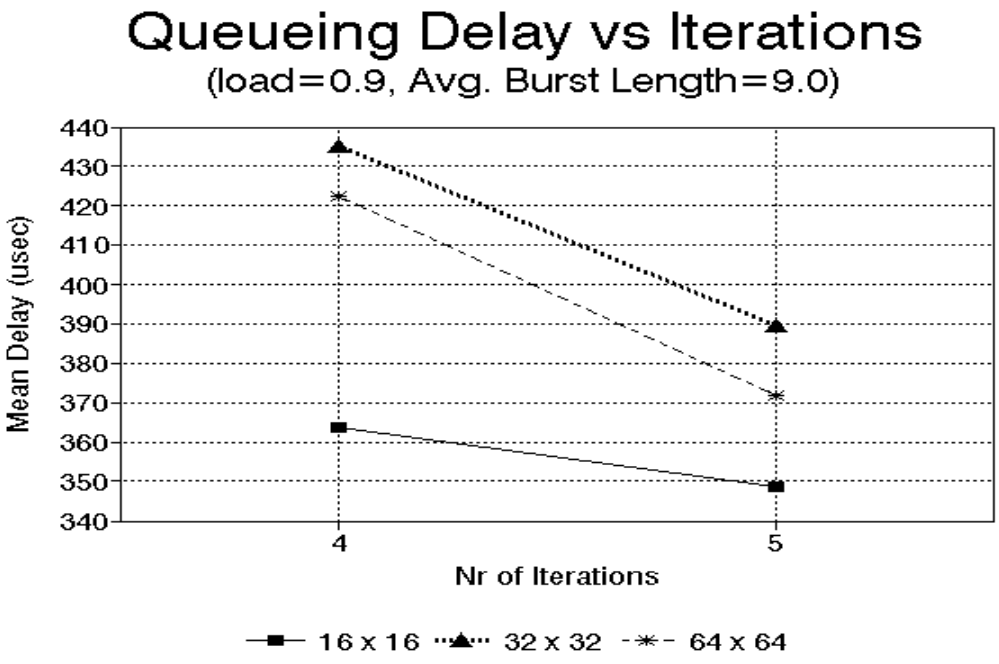

Fig. 42 OnOff: Queueuing Delay vs Iterations (load=9, length=9, switches: 16, 32,64)

The throughput simulation results, collected by running the 'rgs' simulator, have been validated for 1 iteration, identical and independent Bernoulli sources, and an offered load of 100%. For 1(one) iteration, the ATM switch studied in this paper behaves exactly like an input queued packet switch with strict FIFO (first-in first-out) buffers. In the first iteration, all the SIRs (Status Information Registers) are set to FREE, and requests are sent for all the cells located at the heads of the input queue. (for further information, refer to section2.1 in this paper). With one iteration only the cells at the heads of the queues can be routed during the current routing cycle. Therefore, the theoretical performance analysis results of an input queued switch with FIFO buffers, published in [3], are used next to validate the throughput 'rgs' simulation results.

The following Table 1 presents a comparison between theoretical and simulation results. The saturated throughput simulation results were obtained by running the 'RGS' simulator program for 50,000 cell slots, with an initial seed of 1000, a multiplier equal to 6, an offered load of 100%, and a Bernoulli source. The theoretical results can also be found in [3].

N (switch size)	Theoretical Saturation Throughput	Simulation Saturation Throughput
1	1.0000	1.0
2	0.7500	0.7475
3	0.6825	0.68133
4	0.6553	0.65525
5	0.6399	0.6484
6	0.6302	0.643667
7	0.6234	0.628
8	0.6184	0.62525

Table 1 Maximum Throughput Achievable with Input Queueing

In addition to Table 1 above, a second paper written by Karol and Hluchyj, [8], presents a table of simulation throughput results of input queued switches having various window sizes. At 1(one) iteration, the ATM switch running the RGS algorithm behaves exactly like an input queued switch having a window size of '1'. Therefore, the simulation throughput results of those switches with a window size of '1', published in [8], are compared against the simulation results collected running the 'RGS' simulator at 1(one) iteration. The RGS simulation results were collected using the same traffic assumptions described in [8]; i.e., Bernoulli independent and identical sources, and offered load of 100%. An initial seed and multiplier equal to 2 has been used in the 'RGS' simulation runs.

N (switch size)	Window Size	Simulation Saturated Throughput Results from [8]	Simulation Saturated Throughput Results using RGS simulator
16	1	0.60	0.601811
32	1	0.59	0.593331
64	1	0.59	0.585224

Table 2 Comparison between two Simulation Throughput Results

From above comparisons, the saturated throughput results produced by the 'RGS' simulator program appear to be very consistent with expected theoretical results published in [3], and simulation results published in [8].

The following figures, Fig. 43 and Fig. 44, show a comparison between theoretical and simulation saturated throughput results of three ATM switches running the RGS algorithm. A theoretical throughput performance analysis of the ATM switches running

the Request-Grant-Status algorithm was conducted by Professor Motoyama of the University of Campinas, Brasil. Motoyama's theoretical performance analysis can be found in [1], and the throughput results are replicated in Fig. 43 below. The traffic characteristics of both the theoretical and simulation results are the same; i.e., Bernoulli independent and identical sources, and an offered load of 100%. In both cases, the probability of an output port being selected, during a cell generation, is 1/N, where N is the size of the switch (16, 32, or 64). The simulation results were collected by running the simulator program for 50,000 cell slots, using an initial seed and multiplier equal to 2, an offered load of 100%, and a Bernoulli source. The results of Motoyama's theoretical analysis were produced by running the MatLab program found in the Appendix.

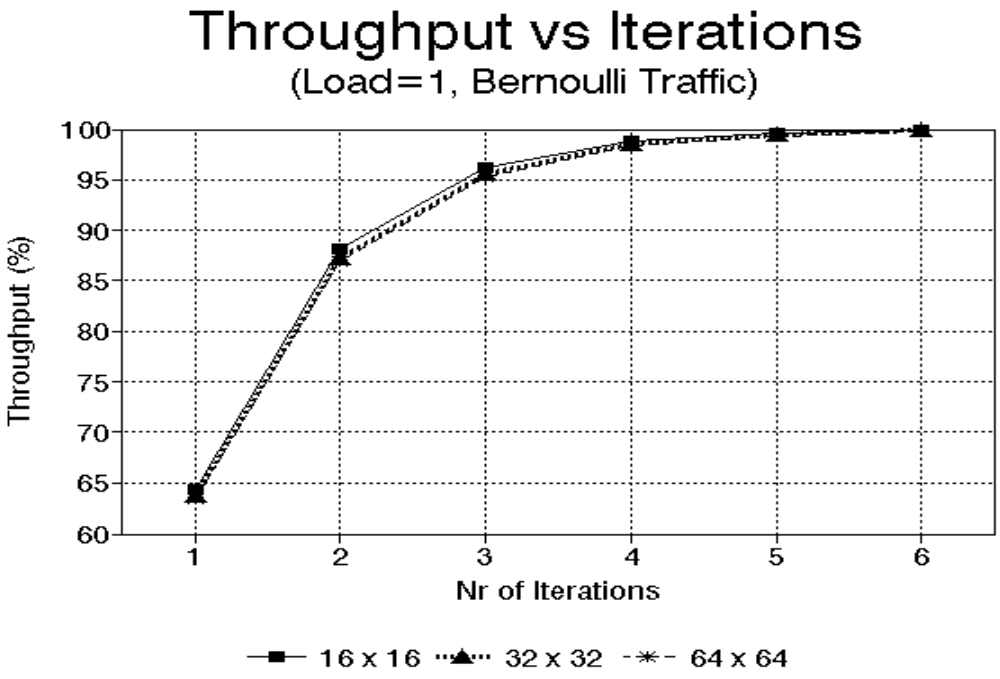

Fig. 43 Motoyama's Theoretical Saturated Throughput Result (Load=1, Bernoulli)

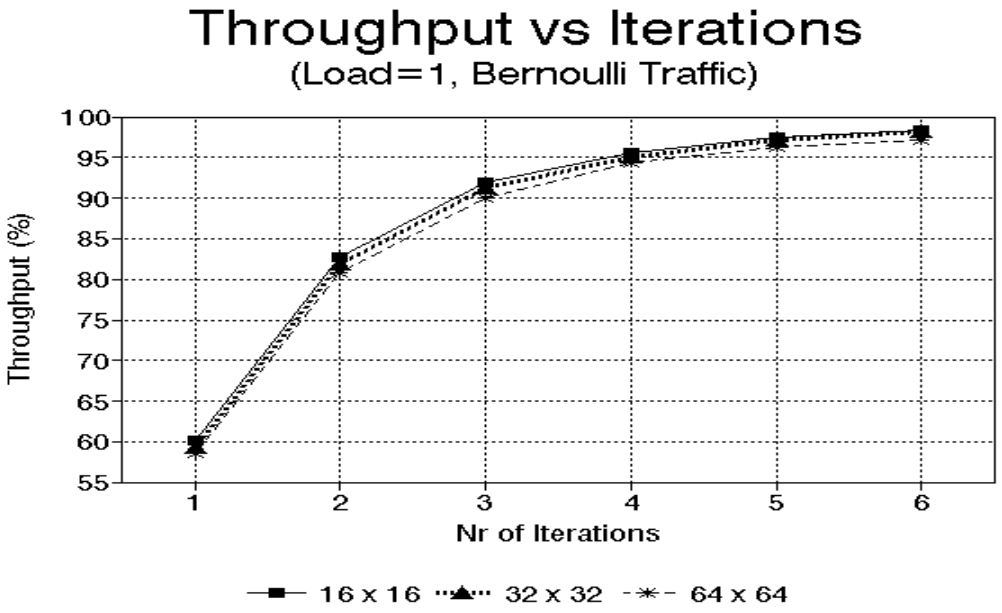

Fig. 44 Simulation Saturated Throughput Result (Load=1, Bernoulli)

As it can be verified from the plots (Fig. 43 and) , the theoretical throughput performance results appear to be slightly better than the results obtained from simulation. For a 16x16 switch, Motoyama's theoretical throughput result for 1(one) iteration is 64.39%. The simulation throughput result is 60.18%, which gives a difference of 4.21% between theoretical and simulation. When 3(three) iterations are used, Motoyama's theoretical analysis gives a throughput of 96.19%. The simulation provides a throughput of 91.87%, which results in a difference of 4.32%. For 5(five) iterations, the difference between the theoretical (Motoyama) and the simulation is 2.09%. And for 6(six) iterations, the difference is 1.49%.

For a 32x32 switch, the throughput difference for 1(one) iteration is 4.45%. And for 3(three) iterations, Motoyama's throughput result is 4.49% greater than the simulation throughput result. If 5(five) iterations are used, the difference between theoretical and simulation throughputs is 2.46%. When the number of iterations is 6, the difference is 1.67%.

For a 64x64 switch, the differences are 4.97% for 1(one) iteration, 5.25 for 3(three) iterations, 3.23% for 5(five) iterations, and 2.73% for 6(six) iterations.

The theoretical throughput performance analysis in [1] does not show the dependence between the number of cells destined for a particular output 'i' in the current and previous slots when the switch operates in steady-state. If no cells are discarded, the number of header cells destined to a given output 'i' in the current slot is the sum of the number of header cells destined to that output but not selected in a previous slot, plus the number of header cells that arrive which are destined to output 'i' in the current slot. Assuming a Bernoulli arrival process with a probability of arrival equal to 'p', and that a cell in a given slot can be evenly sent to any output port, the probability that a specific output i receives a cell from an input in the first iteration is given by Prob[A1] (according to Motoyama's analysis):

Prob[A1] = ({Prob[cell_1 arrives] * Prob[cell_1 is for output port i]} + {Prob{cell_2 arrives] * Prob[cell_2 is for output port i]} + ... + {Prob[cell_k arrives] * Prob[cell_k is for output port i] }) * Prob[one cell among K cells is selected].

Therefore, according to Motoyama's analysis , Prob[A1] is :

$$Prob[A1] = K *(p * 1/N) * 1/K = p * 1/N$$

Eq. 21 Prob Cell Arrival at port i at first iteration

The above expression is correct under the assumption that the ATM switch discards all the conflicting cells destined to an output port 'i', and not selected during the current slot. In the simulation of the RGS algorithm, conflicting cells are not discarded, and can impact the throughput of the ATM switches. Therefore, throughput results produced by the simulation are slightly smaller than the theoretical results from [1].

The theoretical analysis of the RGS algorithm for 1(one) iteration is similar to the performance analysis of an input queued switch published in [3]. Therefore, the analysis presented in this paper follows [3] very closely. Since header cells blocked from an output 'i' are not discarded, the number of cells destined to output 'i' in the current time slot depends on the number of header cells blocked from output 'i' in the previous time slot. For saturated inputs, let's define 'B(m, i)' as the number of cells at the heads of the queues that are blocked for output i at the end of the 'mth' slot. In other words, 'B(m, i)' is the number of requests sent to output 'i' in the mth time slots, but none of these requests are granted in the first iteration. Let's also define 'A(m+1, i)' as the number of cells arriving at the head of the queues in the beginning of slot '(m+1)th', and destined to output 'i'. Therefore, 'B(m+1, i)' and 'B(m, i)' are given by:

$$B(m+1, i) = \max[\, 0,\ B(m, i) + A(m-1, i) - 1]$$

or

$$B(m, i) = \max[\, 0, B(m-1, i) + A(m, i) - 1]$$

Eq. 22 number of blocked cells destined to output ‘i’ at the ‘mth’ slot

‘A(m, i)‘ is the number of cells moving to the heads of input queues during the slot ‘mth’ and destined to an output ‘i’. ‘A(m, i)’ has the following binomial probability, where ‘1/N’ is the probability that a cell is destined to a particular output ‘i’ when the queues are saturated (probability of arrival, ‘p’, is 1).

$$\Pr ob[A(m,i) = K] = \binom{F(m-1)}{K} * (1/N)^{K} * (1-1/N)^{F(m-1)-K}$$

Eq. 23 Binomial Probability for ‘A(m, i)’ for saturated input queues

F(m-1) is the total number of header cells transmitted at the end of the (m-1)th slot. Therefore, F(m-1) can be defined as the total number of input queues subtracted by the total number of queues whose header cells were blocked in the ‘(m-1)th’ slot:

$$F(m-1) \equiv N - \sum_{i}^{N} B(m-1,i)$$

Eq. 24 Definition of F(m-1)

F(m-1) is also the total number of input queues with new cells at their heads in the ‘mth’ time slot.

$$F(m-1) = \sum_{j=1}^{N} A(m,j)$$

Eq. 25 Total number of queues with new cells at the heads

For steady-state operation, the mean value E[F], divided by the size of the switch, N, gives the utilization of the output ports; i.e., the probability that an output port receives a

cell in a given time slot. For an offered load of 100%, or saturated queues, the output port utilization is also the switch throughput, $\rho = E[F]/N$. As the size of the switch becomes too large ($N \sim \infty$), the number of cells destined to output 'i', and moving to the heads of the queue at the time slot 'mth' becomes Poisson with a rate 'ρ' (see [3] for further details). Therefore, the average value of 'B(i)' is derived from the output queueing analysis also done in [3]. From [3], the mean value of 'B(i)' is:

$$E[B(i)] = \frac{\rho^2}{2(1-\rho)}$$

Eq. 26 Steady-state mean value of B(i) derived from output queueing analysis

Dividing both sides of Eq. 24 by 'N':

$$\frac{F(m-1)}{N} \equiv 1 - \frac{\sum_{i}^{N} B(m-1,i)}{N}$$

Eq. 27 F(m-1) divided by 'N'

For the system in equilibrium, the Eq. 27 becomes:

$$\rho = 1 - E[B(i)]$$

Eq. 28 steady-state mean value of B(i)

Substituting Eq. 26 in Eq. 28, and solving the roots of the quadratic equation we have:

$$\rho \approx 58.58\%$$

Eq. 29 RGS throughput performance for a large switch ($N\sim\infty$) and 1(one) iteration

This paper has presented a simulation study of an input buffered ATM switch running the RGS (Request-Grant-Status) scheduling algorithm. The switch normalized throughput and average queueuing delay simulation results have been plotted, and discussed in the paper. As it was demonstrated by the simulation results, the throughput and delay performance of an input buffered ATM switch can be substantially improved by using the RGS algorithm with 2(two) or more iterations. For a Bernoulli traffic source with 100% offered load, an input OC-3 rate (155Mbits/sec), and infinite buffers, the ATM switches (16x16, 32x32, and 64x64) had a throughput of 59% for 1(one) iteration, versus 98% for 6(six) iterations, which represented an improvement of almost 40%. The average cell queueing delay for saturated queues and a Bernoulli arrival process determined by simulation was 1.76 msec for 5(five) iterations, versus about 1.1 msec for 6(six) iterations. For a correlated source (OnOff) with an offered load of 90%, average burst length of 9 cells, the average throughput at 1(one) iteration was close to 57%. For 5(five) iterations, the throughput was close to 100%, resulting in a throughput improvement of 42%. The average queueuing delay was 407usec for 4(four) iterations, versus about 370 usec for 5(five) iterations, which equates to about 37usec in delay improvement as the number of iterations is increased.

In addition to the simulation performance analysis of the RGS algorithm, a comparison between theoretical throughput analysis results published in [1] and simulation results was presented. As it can be seen in Chapter 5 (section 5.4), the difference between Motoyama's and simulation results were between 5% and 1.5% for different iterations. Furthermore, the saturated throughput simulation results of the RGS algorithm for 1(one) iteration were validated against theroretical results. The throughput results obtained from the RGS simulator were consistent with expected theroretical results. Finally, a theoretical throughput analysis of the RGS algorithm for 1(one) iteration, and saturated queues was presented in Chapter 6. This analysis is very similar to the theoretical performance analysis of an input queued switch published in [3].

An attempt was made to extend the throughput theoretical analysis of an ATM input buffered switch running the RGS algorithm for 2 or more iterations. Due to the complex in such analysis and time constraint to conclude this project, this analysis has been left for further studies. A simulation study to determine the cell loss performance for the input buffered switches may be done in future studies. In order to determine the switch cell loss performance, the size of the input buffer would need to be previously determined, and hard coded in the simulation program (InputQueue class). Since the arrivals of cells to each queue is random, the number of cells in the buffers is also random, and is a function of time. The size of the buffer may be determined by running the simulation with different traffic sources and then tracking the maximum number of cells in each buffer during the simulation run. The load offered by these traffic sources should be high, but not large enough to saturate the queues.

It also would be advisable to check the maximum buffer window size that the RGS algorithm. The window size can vary depending on the timing constraint imposed by the switch control clock and memory read/write cycles.

Switch Size	Iterations	Load	Throughput (%)	Delay (usec)
16	1	0.5	99.994	7.8782
16	2	0.5	99.9988	4.7821
16	3	0.5	99.9988	4.7794
16	4	0.5	99.9988	4.7794
16	5	0.5	99.9988	4.7794
32	1	0.5	99.9969	8.4407
32	2	0.5	99.9986	4.8687
32	3	0.5	99.9988	4.8662
32	4	0.5	99.9988	4.8662
32	5	0.5	99.9988	4.8662
64	1	0.5	98.4299	8.6655
64	2	0.5	98.4329	4.8756
64	3	0.5	98.4329	4.8654
64	4	0.5	98.4329	4.8654
64	5	0.5	98.4329	4.8654

Table 3 RGS Simulation Results for Bernoulli Source with Load =50%

Switch Size	Iterations	Load	Avg Length	Throughput (%)	Delay (usec)
16	1	0.5	5.0	99.9058	94.4012
16	2	0.5	5.0	99.9915	16.3392
16	3	0.5	5.0	99.9922	16.3273
16	4	0.5	5.0	99.9922	16.3273
16	5	0.5	5.0	99.9922	16.3273
32	1	0.5	5.0	99.8987	135.873
32	2	0.5	5.0	99.9942	17.5370
32	3	0.5	5.0	99.9941	17.4166
32	4	0.5	5.0	99.9938	17.4029
32	5	0.5	5.0	99.9938	17.4029
64	1	0.5	5.0	98.3164	141.650
64	2	0.5	5.0	98.4173	17.4862
64	3	0.5	5.0	98.4174	17.3603
64	4	0.5	5.0	98.4177	17.3654
64	5	0.5	5.0	98.4177	17.3654

Table 4 RGS Simulation Results for OnOff Source with Load = 50%

Switch Size	Iterations	Load	Throughput (%)	Delay (usec)
16	1	0.75	80.1625	14942.0000
16	2	0.75	99.9938	11.9158
16	3	0.75	99.9945	10.4384
16	4	0.75	99.9948	10.4370
16	5	0.75	99.9952	10.4360
32	1	0.75	79.1696	15734.3000
32	2	0.75	99.9927	12.7917
32	3	0.75	99.9942	10.7837
32	4	0.75	99.9948	10.7400
32	5	0.75	99.9948	10.7400
64	1	0.75	78.0085	15650.2000
64	2	0.75	98.4233	12.5030
64	3	0.75	98.4248	10.5628
64	4	0.75	98.4247	10.5565
64	5	0.75	98.4247	10.5565

Table 5 RGS Simulation Results for Bernoulli Source with Load =75%

Switch Size	Iterations	Load	Avg Length	Throughput (%)	Delay (usec)
16	1	0.75	7.5	69.9656	22633.80
16	2	0.75	7.5	99.8298	164.5260
16	3	0.75	7.5	99.9481	84.0322
16	4	0.75	7.5	99.9496	83.6076
16	5	0.75	7.5	99.9496	83.6076
32	1	0.75	7.5	68.9636	23443.00
32	2	0.75	7.5	99.7866	335.7880
32	3	0.75	7.5	99.9248	92.1901
32	4	0.75	7.5	99.9275	91.0312
32	5	0.75	7.5	99.9277	91.1272
64	1	0.75	7.5	67.8001	23531.40
64	2	0.75	7.5	98.242	325.0860
64	3	0.75	7.5	98.3719	89.3677
64	4	0.75	7.5	98.3699	88.5189
64	5	0.75	7.5	98.3733	88.6990

Table 6 RGS Simulation Results for OnOff Source with Load = 75%

Switch Size	Iterations	Load	Throughput (%)	Delay (usec)
16	1	0.9	66.8825	24970.70000
16	2	0.9	91.9347	6114.17000
16	3	0.9	99.9725	38.71950
16	4	0.9	99.9779	29.89430
16	5	0.9	99.9776	29.98760
32	1	0.9	66.0187	25584.50000
32	2	0.9	90.9093	6874.31000
32	3	0.9	99.9557	51.59220
32	4	0.9	99.9773	31.91460
32	5	0.9	99.9769	31.60240
64	1	0.9	65.0505	25573.30000
64	2	0.9	89.7655	66133.20000
64	3	0.9	98.407	44.23700
64	4	0.9	98.4145	29.81470
64	5	0.9	98.4145	29.58980

Table 7 RGS Simulation Results for Bernoulli Source with Load =90%

Switch Size	Iterations	Load	Avg Length	Throughput (%)	Delay (usec)
16	1	0.9	9.0	58.0463	31824.8
16	2	0.9	9.0	85.2951	11131.6
16	3	0.9	9.0	97.8074	1941.52
16	4	0.9	9.0	99.7501	363.885
16	5	0.9	9.0	99.7682	348.678
32	1	0.9	9.0	57.7982	31742.6
32	2	0.9	9.0	84.3852	11845.7
32	3	0.9	9.0	96.8743	2599.49
32	4	0.9	9.0	99.7014	435.092
32	5	0.9	9.0	99.7258	389.856
64	1	0.9	9.0	56.2684	32374.1
64	2	0.9	9.0	83.0419	11721.7
64	3	0.9	9.0	95.1818	2410.15
64	4	0.9	9.0	97.9608	422.301
64	5	0.9	9.0	98.1112	372.026

Table 8 RGS Simulation Results for OnOff Source with Load = 90%

Switch Size	Iterations	Load	Throughput (%)	Delay (msec)
16	1	1	60.1811	29.96940
16	2	1	82.7482	13.02550
16	3	1	91.865	6.11719
16	4	1	95.5343	3.24477
16	5	1	97.5295	1.75741
16	6	1	98.39	1.08377
32	1	1	59.3331	30.59970
32	2	1	81.8016	13.73310
32	3	1	91.1115	6.73552
32	4	1	95.1063	3.69357
32	5	1	97.0159	2.20692
32	6	1	98.1477	1.36701
64	1	1	58.5224	30.50600
64	2	1	80.7627	13.53700
64	3	1	90.0547	6.40355
64	4	1	94.2321	3.18278
64	5	1	96.165	1.64484
64	6	1	97.052	0.959512

Table 9 RGS Simulation Results for Bernoulli Source with Load =100%

Switch Size	Iterations	Load	Throughput (%)
16	1	1	64.39
16	2	1	88.14
16	3	1	96.19
16	4	1	98.8
16	5	1	99.62
16	6	1	99.88
32	1	1	63.79
32	2	1	87.28
32	3	1	95.6
32	4	1	98.49
32	5	1	99.48
32	6	1	99.82
64	1	1	63.5
64	2	1	86.87
64	3	1	95.31
64	4	1	98.33
64	5	1	99.4
64	6	1	99.79

Table 10 Motoyama Theorectical Analysis Results

This Appendix provides instructions to run the "rgs" simulation program. The "rgs" program was written by Rubens S. Gomes as a partial fullfilment of a master's degree project in Electrical Engineering at the University of Kansas.

The "rgs" program has been compiled, linked, and tested in the following platforms:

* Windows95
* HP UX 10.01
* DEC UNIX version 4.0
* SunOS

To run the program currently installed in the EECS UNIX system of the University of Kansas, follow these steps:

1) Login in a EECS UNIX machine at the Univ. of Kansas

2) Then change to one of the following directories depending on your machine's architecture

HP_UX : /rusers/rgomes/MS_Proj/Simulation/HP_UX
DEC_UNIX: /rusers/rgomes/MS_Proj/Simulation/DEC_UNIX

SunOS : /rusers/rgomes/MS_Proj/Simulation/SunOS

3) At the system prompt '%' type the following command, and press ENTER or RETURN:

./rgs

4) Then follow the instructions on the screen.

Parameters Information

Seed:

The "seed" and "multiplier" are the required to instantiate an object from the Random Number Generator Class. This Random Generator object is unique per Traffic Source. The "seed" can be any integer number. The seed for the others traffic source is calculated based on the entered 'seed' by incrementing it by 1,000,000.

For example, if you enter "2":

Traffic_Source[0] seed = 2
Traffic_Source[1] seed = 1,000,002
Traffic_Source[2] seed = 2,000,002
....

Multiplier:

The multiplier along with the seed are the parameters required to instantiate a Random Nr Generator object. The possible multipliers are '1', '2', '3', '4', ..., '8'. Each Random Nr Generator object is passed a 'seed' and 'multiplier' at the time this object is created. The multiplier for the second, third,.. random number will follow the pattern:

For example, if you enter '1':

Traffic_Source[0] multiplier = 1
Traffic_Source[1] multiplier = 2
Traffic_Source[2] multiplier = 3
....
Traffic_Source[7] multiplier = 8
Traffic_Source[8] multiplier = 1
Traffic_Source[1] multiplier = 2
....

Number of Cells for Simulation:

This is the total number of cell slots the simulation will run. It is suggested not to enter a number bigger than 50,000, since the higher this number the longer the simulation will take to run.

Number of ports for the ATM switch:

This is the size of the switch. The maximum switch size allowed is 64x64. A integer number between 1 and 64 should be entered for this parameter.

Number of Iterations:

The number of iterations is parameter used by the RGS algorithm. As can be seen from the master's project paper, the higher this number the better is the switch performance. The allowable values range from 1 to 6.

Offered Load:

The offered load entered should be a decimal number between 0 and 1 (1 not inclusive for OnOff source).

Average Burst Length:

The average burst length is any meaningful number which represents the average number of cells that will be contained in a burst. Notice that according to the master's project paper this number has a minimum acceptable value, which is determined based on the offered load entered above.

When the simulation is done, the following information will be displayed on the screen. Here is an example of a complete simulation run:

```
/net/chicago/users/home/thesis/code>./rgs
This program has been written to simulate an ATM
input buffered switch running the Request-Grant
Status (RGS) scheduling algorithm.


Simulation Program written by Rubens S. Gomes
to fullfil master's project at the University
of Kansas, Lawrence, KS, USA.


Press <ENTER> or <RETURN> key to continue...

Select Type of Traffic Source
(1) Bernoulli  (2) On_Off:                1
Enter Seed:                            1000
Enter Multiplier (1, 2, ..., 8):          6
Enter nr of cells slots for simulation:  1000
Enter NR of ports for the ATM switch:    16
Enter Nr of Iterations (1, 2, ..., 6) :   1
Enter  offered load (0 < load =< 1):      1
```

Please, Wait ...

Simulation Results

Traffic type:	BERNOULLI
Switch size:	16
Total Number Cell Slots:	1000 slots
Total Nr Steady State Slots:	900 slots
Number of Iterations:	1
Total Nr of cells arrived:	16000 cells
Total Nr of cells routed:	9709 cells
Steady State Nr cells arrived:	14400 cells
Steady State Nr cells routed:	8689 cells
Switch Offered Load:	100 %
Avg Utilization of Output Ports:	60.3403 %
Switch Throughput:	60.6812 %
Queueing Avg Cell Delay:	0.000589154 sec.

Press Return to continue...

/net/chicago/users/home/thesis/code>

```
%
% MatLab program to generate the saturated throughput results published in [1]
%
% Authors:      Shusaburo Motoyama & Rubens Gomes
% Date:      05/01/95.
% EECS Department  - University of Kansas
%
% Note: This program was written based on the formulas published in Motoyama's
% paper [1].
%
p=1;                    % switch is saturated.
N=16;                   % 16 x 16 switch
C1=(1-(p/N)).^N;
%
%  One iteration throughput
%
rho1=1-(1-(p/N)).^N;
A1=N.*rho1;
B1=(1-(1-(p/N)).^(N-A1));
rho1=1-(1-(p/N)).^N;
A1=N.*rho1;
B1=(1-(1-(p/N)).^(N-A1));
```

```
%
%  Two iteration throughput
%
rho2=C1.*B1;
A2=(N-A1).*rho2;
C2=(1-(p/N)).^(N-A1);
B2=1-((1-(p/N)).^(N-A1-A2));
%
%  Three iteration throughput
%
rho3=C1.*C2.*B2;
A3=(N-A1-A2).*rho3;
B3=1-((1-(p/N)).^(N-A1-A2-A3));
C3=(1-(p/N)).^(N-A1-A2);
%
%  Four iteration throughput
%
rho4=C1.*C2.*C3.*B3;
A4=(N-A1-A2-A3).*rho4;
B4=1-((1-(p/N)).^(N-A1-A2-A3-A4));
C4=(1-(p/N)).^(N-A1-A2-A3);
% %  Five iteration throughput
%
rho5=C1.*C2.*C3.*C4.*B4;
A5=(N-A1-A2-A3-A4).*rho5;
B5=1-((1-(p/N)).^(N-A1-A2-A3-A4-A5));
C5=(1-(p/N)).^(N-A1-A2-A3-A4);
%
%  Six iteration throughput
%
```

```
rho6=C1.*C2.*C3.*C4.*C5.*B5;
r1=rho1;
r2=r1+rho2;
r3=r2+rho3;
r4=r3+rho4;
r5=r4+rho5;
r6=r5+rho6;
R=[r1 r2 r3 r4 r5 r6]
```

```
#########################################################################
#                                                                       #
#    Filename:         Makefile                                         #
#                                                                       #
#    Author:           Rubens S. Gomes                                  #
#    Date:             August, 1996.                                    #
#    Site:             Electrical Engineering & Computer Science        #
#                      The University of Kansas                         #
#                      Lawrence, KS, 66045                              #
#    Operating System: UNIX HP OS 10.01, UNIX DEC OS/F, Windows95       #
#    Language:         C++                                              #
#    Node Location:    noehter.eecs.ukans.edu                           #
#    Path:             ~rgomes/MS_Proj/Simulation/Makefile              #
#                                                                       #
#    Purpose:                                                           #
#                                                                       #
#        This Makefile compiles and links the ATM switch                #
#        simulation program developed by Rubens S. Gomes.               #
#        The simulation program is part of fullfilment of               #
#        Master's project at the University of Kansas.                  #
#                                                                       #
#                                                                       #
#                                                                       #
#########################################################################

LIBS= -lm
#
# If you have "purify", and would like to run it with the program,
# uncomment next line.
#
#PURIFY = purify
PURIFY =

CPP = g++
#CPP = CC
```

```
#CPP = gcc
#CPP = c++

CFLAGS =  -c -v  -w
DBGFLAG = -g
PROGRAM = rgs

SOURCES = $(PROGRAM).C atm_par.C atmcell.C bern.C \
         icu_unit.C list.C ocu_unit.C onoff.C     \
         queue.C rand.C rgs_prot.C simu.C         \
         stats.C timer.C traffic.C user_int.C

OBJECTS = $(PROGRAM).o atm_par.o atmcell.o bern.o \
         icu_unit.o list.o ocu_unit.o onoff.o     \
         queue.o rand.o rgs_prot.o simu.o         \
         stats.o timer.o traffic.o user_int.o

HEADERS = $(PROGRAM).h atm_par.h atmcell.h     \
         icu_unit.h list.h ocu_unit.h          \
         queue.h rand.h rgs_prot.h simu.h      \
         stats.h timer.h traffic.h user_int.h

$(PROGRAM):         $(OBJECTS)
  $(PURIFY) $(CPP) $(DBGFLAG) -o $(PROGRAM) $(OBJECTS) $(LIBS)

$(PROGRAM).o: $(PROGRAM).C $(PROGRAM).h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) $(PROGRAM).C

atm_par.o:    atm_par.C atm_par.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) atm_par.C

atmcell.o:    atmcell.C atmcell.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) atmcell.C

bern.o:    bern.C traffic.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) bern.C

icu_unit.o:   icu_unit.C icu_unit.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) icu_unit.C
```

```
list.o:       list.C list.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) list.C

ocu_unit.o:   ocu_unit.C ocu_unit.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) ocu_unit.C

onoff.o:      onoff.C traffic.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) onoff.C

queue.o:      queue.C queue.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) queue.C

rand.o:       rand.C rand.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) rand.C

rgs_prot.o:   rgs_prot.C rgs_prot.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) rgs_prot.C

simu.o:       simu.C simu.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) simu.C

stats.o:      stats.C stats.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG)  stats.C

timer.o:    timer.C timer.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) timer.C

traffic.o:    traffic.C traffic.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) traffic.C

user_int.o:    user_int.C user_int.h
  $(PURIFY) $(CPP) $(CFLAGS) $(DBGFLAG) user_int.C

clean:
  rm -f $(OBJECTS)

clean-all:
  rm -f $(OBJECTS) $(PROGRAM)  *.o
```

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         rgs.C                                         //
//                                                                    //
//    Author:           Rubens S. Gomes                               //
//    Date:             August, 1996.                                 //
//    Site:             Electrical Engineering & Computer Science     //
//                      The University of Kansas                      //
//                      Lawrence, KS, 66045                           //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows95             //
//    Language:         C++                                           //
//    Node Location:    noehter.eecs.ukans.edu                        //
//    Path:             ~rgomes/MS_Proj/Simulation/rgs.C              //
//                                                                    //
//    Purpose:          This is the body of the main program for      //
//                      the simulation of an input buffered ATM       //
//                      switch based on the Request-Grant-Status      //
//                      Scheduling Algorithm.                         //
//                                                                    //
//                      This program was written as partial           //
//                      fullfilment of master's project in            //
//                      Electrical Engineering at the University      //
//                      of Kansas.                                    //
//                                                                    //
//    Limitations:      None.                                         //
//                                                                    //
//                                                                    //
////////////////////////////////////////////////////////////////////////

#include "rgs.h"

int main()
{
  SimulationParameter* parm;
  Simulation* simu;
  Timer* timer;

  parm = new SimulationParameter;
  simu = new Simulation( parm );
  timer= new Timer( parm, simu );
```

```
  while( !timer->SimulationTimeDone() )
  {
    // for each new slot, the timer sends a message
    // to the simulation object which runs the RGS algorithm.
    timer->NewCellSlot();
  }
  simu->printResults();

  delete parm;
  delete simu;
  delete timer;

  return( 0 );
}

////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         rgs.h                                         //
//                                                                    //
//    Author:           Rubens S. Gomes                               //
//    Date:             August, 1996.                                 //
//    Site:             Electrical Engineering & Computer Science     //
//                      The University of Kansas                      //
//                      Lawrence, KS, 66045                           //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows95             //
//    Language:         C++                                           //
//    Node Location:    noehter.eecs.ukans.edu                        //
//    Path:             ~rgomes/MS_Proj/Simulation/rgs.h              //
//                                                                    //
//    Purpose:          Header file for the RGS program.              //
//                                                                    //
//                                                                    //
//    Limitations:      None.                                         //
//                                                                    //
//                                                                    //
////////////////////////////////////////////////////////////////////////

#include "atm_par.h"
#include "simu.h"
#include "timer.h"
#include <memory.h>
#include <malloc.h>
////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         atmcell.h                                     //
```

```
//                                                                      //
//    Author:           Rubens S. Gomes                                 //
//    Date:             August, 1996.                                   //
//    Site:             Electrical Engineering & Computer Science       //
//                      The University of Kansas                        //
//                      Lawrence, KS, 66045                             //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows95               //
//    Language:         C++                                             //
//    Node Location:    noehter.eecs.ukans.edu                          //
//    Path:             ~rgomes/MS_Proj/Simulation/atmcell.h            //
//                                                                      //
//    Purpose:          Header file for the ATM Cell object.  This      //
//                      file defines the data members contained in      //
//                      the object.                                     //
//                                                                      //
//                                                                      //
//    Limitations:      None.                                           //
//                                                                      //
//                                                                      //
//////////////////////////////////////////////////////////////////////////

#if !defined _ATMCell_HPP
#define      _ATMCell_HPP

class ATM_Cell
{
  public:
    ATM_Cell(void);
    ~ATM_Cell(void);
    long   arrival_time;   // slot nr that cell arrived
    long   departure_time; // slot nr that cell departed
    int    output_port;    // output port cell destined
    int    input_port;     // output port cell arrived
};

#endif
//
//  File:      atmcell.C
//

#include "atmcell.h"

/*----------------------------------------------------------------*/
//   Name:        ATM_Cell
//   Parameters:  None
```

```
//   Returns:     nothing
//   Purpose:     constructor method for the ATM_Cell object.
/*-------------------------------------------------------------*/
ATM_Cell::ATM_Cell( void )
{
  arrival_time = 0;
  departure_time = 0;
  output_port = 0;
  input_port = 0;
}

/*-------------------------------------------------------------*/
//   Name:        ~ATM_Cell
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     destructor method for the ATM_Cell object.
/*-------------------------------------------------------------*/
ATM_Cell::~ATM_Cell( void )
{
}
////////////////////////////////////////////////////////////////////
//                                                                //
//    Filename:         atm_par.h                                 //
//                                                                //
//    Author:           Rubens S. Gomes                           //
//    Date:             October, 1996.                            //
//    Site:             Electrical Engineering & Computer Science //
//                      The University of Kansas                  //
//                      Lawrence, KS, 66045                       //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows 95        //
//    Language:         C++                                       //
//    Node Location:    noehter.eecs.ukans.edu                    //
//    Path:             ~rgomes/MS_Proj/Simulation/atm_par.h      //
//                                                                //
//    Purpose:          Header file for the Simulation Parameter  //
//                      object.  This file defines the data member //
//                      and operations performed by the object.   //
//                                                                //
//    Limitations:      None.                                     //
//                                                                //
//                                                                //
////////////////////////////////////////////////////////////////////
```

```
#if !defined _PARAMETERS_HPP
#define      _PARAMETERS_HPP

#include <math.h>
#include "user_int.h"

#define  MAX_PORTS            (    65 )
#define  SEED_INCREMENT       (  1000 )
#define  ITERATION_TIME_DELAY (     1 )
#define  BERNOULLI            (     1 )
#define  ON_OFF               (     2 )

class SimulationParameter
{
  public:
    SimulationParameter( void );
    ~SimulationParameter( void );
    int    GetNrIterations( void );
    int    GetNrPorts( void );
    long   GetTotalTime( void );
    double GetCellBurstLength( void );
    double GetOfferedLoad( void );
    long   GetSeed( void );
    int    GetTransientTime( void );
    int    GetMultiplier( void );
    void   UpdateSeed( void );
    void   UpdateMultiplier( void );
    int    GetTypeOfSource( void );

  private:
    void   SetSourceType( void );
    void   SetParameters( void );
    void   SetBernParameters( void );
    void   SetOnOffParameters( void );
    void   SetSeed( void );
    void   SetMultiplier( void );
    void   SetTotalTime( void );
    void   SetNrPorts( void );
    void   SetNrIterations( void );
    void   SetAvgLoad( void );
    void   SetAvgCellBurstLength( void );
    double CalculateMinAvgLength( void );

    UserInterface *User;
    int     NR_ITERATIONS;
```

```
    long    TOTAL_TIME;
    long    SEED;
    int     MULTIPLIER;
    double  MIN_AVG_LENGTH;
    double  AVG_BURST_LENGTH;
    double  AVG_LOAD;
    int     NUM_PORTS;
    int     TRANSIENT_TIME;
    int     SOURCE_TYPE;
};

#endif
//
//  File:      atm_par.C
//

#include "atm_par.h"

/*-------------------------------------------------------------*/
//   Name:        SimulationParameter
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     constructor function for the SimulationParameter
//                object.  This function initializes all the
//                private member data belonging to this object.
/*-------------------------------------------------------------*/
SimulationParameter::SimulationParameter( void )
{
  SEED = 0;
  MULTIPLIER = 0;
  AVG_LOAD = 0;
  AVG_BURST_LENGTH = 0;
  User = new UserInterface;
  SetParameters();
}

/*-------------------------------------------------------------*/
//   Name:        SimulationParameter
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     destructor function for the SimulationParameter
//                object.  This function initializes all the
//                private member data belonging to this object.
/*-------------------------------------------------------------*/
SimulationParameter::~SimulationParameter( void )
```

```
{
  if( User )
  {
    delete User;
  }
}

/*-------------------------------------------------------------*/
//   Name:        SetParameters
//   Parameters:  None
//   Returns:     void
//   Purpose:     This functions sets all parameters needed for
//                the simulation.
/*-------------------------------------------------------------*/
void SimulationParameter::SetParameters( void )
{
  SetSourceType();
  if( SOURCE_TYPE == BERNOULLI )
  {
    SetBernParameters();
    User->WaitMessage();
    delete User;
  }
  else
  {
    SetOnOffParameters();
    User->WaitMessage();
    delete User;
  }
}

/*-------------------------------------------------------------*/
//   Name:        SetSourceType
//   Parameters:  None
//   Returns:     void
//   Purpose:     This functions asks the user to select the
//                type of Traffic Source, and returns either
//                BERNOULLI or ON_OFF.
/*-------------------------------------------------------------*/
void SimulationParameter::SetSourceType( void )
{
  SOURCE_TYPE = User->EnterTypeOfSource();
}
```

```
/*-------------------------------------------------------------*/
//   Name:        SetBernParameters
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This functions sets all the parameters to be
//                used by the simulation when a Bernoulli Traffic
//                source is used.
/*-------------------------------------------------------------*/
void SimulationParameter::SetBernParameters( void )
{
  SetSeed();
  SetMultiplier();
  SetTotalTime();
  SetNrPorts();
  SetNrIterations();
  SetAvgLoad();
}

/*-------------------------------------------------------------*/
//   Name:        SetOnOffParameters
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This functions sets all the parameters to be
//                used by the simulation when a OnOff Traffic
//                source is used.
/*-------------------------------------------------------------*/
void SimulationParameter::SetOnOffParameters( void )
{
  SetSeed();
  SetMultiplier();
  SetTotalTime();
  SetNrPorts();
  SetNrIterations();
  SetAvgLoad();
  SetAvgCellBurstLength();
}

/*-------------------------------------------------------------*/
//   Name:        SetSeed
//   Parameters:  None
//   Returns:     void
//   Purpose:     This functions asks the user to enter the
//                initial seed to be used by the first traffic
//                source created.   The other traffic sources
//                will have different seeds.
```

```
/*-------------------------------------------------------------*/
void SimulationParameter::SetSeed( void )
{
  SEED = User->EnterSeed();
}

/*-------------------------------------------------------------*/
//   Name:        SetMultiplier
//   Parameters:  None
//   Returns:     void
//   Purpose:     This functions asks the user to enter the
//                initial multiplier to be used by the first traffic
//                source created.   The other traffic sources
//                will use different multiplier.
/*-------------------------------------------------------------*/
void SimulationParameter::SetMultiplier( void )
{
  MULTIPLIER = User->EnterMultiplier();
}

/*-------------------------------------------------------------*/
//   Name:        SetTotalTime
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This functions sets total time in cell slots to
//                run simulation. Then, it calculates the transient
//                time which is 10% of the total simulation time.
/*-------------------------------------------------------------*/
void SimulationParameter::SetTotalTime( void )
{
  TOTAL_TIME = User->EnterTotalTime();
  TRANSIENT_TIME = (int) (0.1 * TOTAL_TIME);
}


/*-------------------------------------------------------------*/
//   Name:        SetNrPorts
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This function sets number of ports for the
//                ATM switch.
/*-------------------------------------------------------------*/
void SimulationParameter::SetNrPorts( void )
{
  char done = FALSE;
```

```
  while( !done )
  {
    NUM_PORTS = User->EnterNrPorts();
    if( NUM_PORTS > MAX_PORTS )
    {
      User->WrongEntry();
    }
    else
    {
      done = TRUE;
    }
  }
}

/*-------------------------------------------------------------*/
//   Name:        SetNrInterations
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This functions sets nr of Iterations for the
//                simulation.
/*-------------------------------------------------------------*/
void SimulationParameter::SetNrIterations( void )
{
  NR_ITERATIONS = User->EnterNrIterations();
}


/*-------------------------------------------------------------*/
//   Name:        SetAvgLoad
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This functions sets the average load offered
//                by the traffic source.  The offered load is
//                the probability of success for the Bernoulli
//                Traffic.  In the case of the OnOff Markov traffic,
//                the probability of the source being in the On
//                state is the offered load.
/*-------------------------------------------------------------*/
void SimulationParameter::SetAvgLoad( void )
{
  char done = FALSE;

  while (!done)
  {
    AVG_LOAD = User->EnterAvgLoad(SOURCE_TYPE);
```

```
    if(   (SOURCE_TYPE == BERNOULLI)
       && ( (AVG_LOAD < 0) || (AVG_LOAD > 1) ) )
    {
      User->WrongEntry();
    }
    else if(   (SOURCE_TYPE == ON_OFF)
            && ( (AVG_LOAD < 0) || (AVG_LOAD >= 1) ) )
    {
      User->WrongEntry();
    }
    else
    {
      done = TRUE;
    }
  }
}
/*-------------------------------------------------------------*/
//   Name:        SetAvgCellBurstLength
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This functions sets average cells burst length.
//                The average cells burst length is the mean of
//                the geometric distribution which describes the
//                arrivals of cells in the queue.  This parameter
//                is only used for the OnOff Traffic Source.
/*-------------------------------------------------------------*/
void SimulationParameter::SetAvgCellBurstLength( void )
{
  char done = FALSE;
  double min_length = 0;

  min_length = CalculateMinAvgLength();
  while( !done )
  {
    AVG_BURST_LENGTH = User->EnterAvgCellBurstLength();
    if ( float(AVG_BURST_LENGTH) < float(min_length) )
    {
      User->WrongEntry( min_length );
    }
    else
    {
      done = TRUE;
    }
  }
}
```

```
/*-------------------------------------------------------------*/
//   Name:        GetTypeOfSource
//   Parameters:  None
//   Returns:     SOURCE_TYPE (int).
//   Purpose:     This functions returns the type of the
//                traffic source being used.
/*-------------------------------------------------------------*/
int SimulationParameter::GetTypeOfSource( void )
{
  return( SOURCE_TYPE );
}

/*-------------------------------------------------------------*/
//   Name:        GetTotalTime
//   Parameters:  None
//   Returns:     Total time for the simulation to run.
//   Purpose:     This functions returns the total time to run
//                simulation.
/*-------------------------------------------------------------*/
long SimulationParameter::GetTotalTime( void )
{
  return( TOTAL_TIME );
}

/*-------------------------------------------------------------*/
//   Name:        GetNrPorts
//   Parameters:  None
//   Returns:     Number of ports in the ATM switch.
//   Purpose:     This function returns the number of ports
//                for the ATM switch to be used during the
//                simulation.
/*-------------------------------------------------------------*/
int SimulationParameter::GetNrPorts( void )
{
  return( NUM_PORTS );
}


/*-------------------------------------------------------------*/
//   Name:        GetNrIterations
//   Parameters:  None
//   Returns:     Number of Iterations
//   Purpose:     This functions returns the number of iterations
```

```
//                to be used during the simulation.
/*-------------------------------------------------------------*/
int SimulationParameter::GetNrIterations( void )
{
  return( NR_ITERATIONS );
}

/*-------------------------------------------------------------*/
//   Name:        GetOfferedLoad
//   Parameters:  None
//   Returns:     The value stored in the AVG_LOAD
//   Purpose:     This functions returns the value currently
//                stored in the private offered load variable.
/*-------------------------------------------------------------*/
double SimulationParameter::GetOfferedLoad( void )
{
  return( AVG_LOAD );
}

/*-------------------------------------------------------------*/
//   Name:        GetCellBurstLength
//   Parameters:  None
//   Returns:     The value stored in the AVG_BURST_LENGTH
//   Purpose:     This functions returns the value currently
//                stored in the private burst length variable.
/*-------------------------------------------------------------*/
double SimulationParameter::GetCellBurstLength( void )
{
  return( AVG_BURST_LENGTH );
}

/*-------------------------------------------------------------*/
//   Name:        GetSeed
//   Parameters:  None
//   Returns:     Seed (long).
//   Purpose:     This functions returns the seed for the Random
//                Nr Generator.
/*-------------------------------------------------------------*/
long SimulationParameter::GetSeed( void )
{
  return( SEED );
}


/*-------------------------------------------------------------*/
```

```
//   Name:        GetMultiplier
//   Parameters:  None
//   Returns:     multiplier (int).
//   Purpose:     This functions returns multiplier for Random
//                Nr Generator.
/*-------------------------------------------------------------*/
int SimulationParameter::GetMultiplier( void )
{
  return( MULTIPLIER );
}


/*-------------------------------------------------------------*/
//   Name:        GetTransientTime
//   Parameters:  None
//   Returns:     Transient Time in cell slots.
//   Purpose:     This functions returns the number of cell slots
//                to run the simulation during the transient state.
/*-------------------------------------------------------------*/
int SimulationParameter::GetTransientTime( void )
{
  return( TRANSIENT_TIME );
}


/*-------------------------------------------------------------*/
//   Name:        UpdateSeed
//   Parameters:  None
//   Returns:     nothing.
//   Purpose:     This functions updates the value of the seed
//                to be used in later random number generations
/*-------------------------------------------------------------*/
void SimulationParameter::UpdateSeed( void )
{
    if ( SEED >= 2145000000 )
    {
       SEED = 0;
    }
    else
   {
       SEED = SEED + 1000000;
    }
}

/*-------------------------------------------------------------*/
```

```
//   Name:        UpdateMultiplier
//   Parameters:  None
//   Returns:     nothing.
//   Purpose:     This functions updates the value of multiplier
//                to be used in later random number generations
/*-------------------------------------------------------------*/
void SimulationParameter::UpdateMultiplier( void )
{

    if ( MULTIPLIER >= 8 )
    {
      MULTIPLIER = 1;
    }
    else
   {
      MULTIPLIER = MULTIPLIER + 1;
    }
}

/*-------------------------------------------------------------*/
//   Name:        CalculateMinAvgLength
//   Parameters:  None
//   Returns:     min_avg_length (double)
//   Purpose:     This function calculates the minimum required
//                average cell burst length for a given offered
//                load (AVG_LOAD).  The Expected number has to
//                greater or equal to LOAD/(1-LOAD) to satisfy
//                the range for the transition probabilities of
//                0 to 1.  For further details, refer to my master's
//                project paper.  This function should only be
//                called after the avg load has been set.
/*-------------------------------------------------------------*/
double SimulationParameter::CalculateMinAvgLength( void )
{
  double min_avg_length = 0;
  min_avg_length = AVG_LOAD / (1.0 - AVG_LOAD);
  if( min_avg_length < 1.0 )
  {
    //
    // the minimum average burst length has to be greater than
    // 1(one) to satisfy the inequality:
    //
    // 0 < {[1 - (1/avg_length)] == Prob_On_to_On} < 1
    min_avg_length = 1.0;
  }
```

```
  return( min_avg_length );
}


////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         user_int.h                                    //
//                                                                    //
//    Author:           Rubens S. Gomes                               //
//    Date:             October, 1996.                                //
//    Site:             Electrical Engineering & Computer Science     //
//                      The University of Kansas                      //
//                      Lawrence, KS, 66045                           //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows95             //
//    Language:         C++                                           //
//    Node Location:    noehter.eecs.ukans.edu                        //
//    Path:             ~rgomes/MS_Proj/Simulation/user_int.h         //
//                                                                    //
//    Purpose:          Header file for the user interface class.     //
//                      This file defines the data member             //
//                      and operations performed by the object.       //
//                                                                    //
//    Limitations:      None.                                         //
//                                                                    //
//                                                                    //
////////////////////////////////////////////////////////////////////////

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>

#if !defined _USERINTERFACE_HPP
#define      _USERINTERFACE_HPP

#define BERNOULLI ( 1 )
#define ON_OFF    ( 2 )
#define FALSE     ( 0 )
#define TRUE      ( 1 )

class UserInterface
{
  public:
    UserInterface( void );
    int EnterTypeOfSource( void );
    int EnterSeed( void );
```

```
    int EnterMultiplier( void );
    int EnterTotalTime( void );
    int EnterNrPorts( void );
    int EnterNrIterations( void );
    double EnterAvgLoad( int source_type );
    double EnterAvgCellBurstLength( void );
    void WrongEntry( void );
    virtual void WrongEntry( double min_length);
    void WaitMessage( void );

  private:
    void DisplayIntro( void );
};

#endif
//
//  File:      user_int.C
//

#include "uIIIEyvK4II5IIIEyNXXuK4II5IIIX  adNPnclude "□EyeK4PP5PPPEy K4II5IIIEy"K4II5IIIEy"K4I
```

```
  cout << "input buffered switch running  the Request-Grant" << endl;
  cout << "Status (RGS) scheduling algorithm.              " << endl;
  cout << endl;
  cout << endl;
  cout << "Simulation Program written  by  Rubens  S. Gomes" << endl;
  cout << "to fullfil master's project  at  the  University" << endl;
  cout << "of Kansas, Lawrence, KS, USA."                    << endl;
  cout << endl;
  cout << endl;
  cout << "Press <ENTER> or <RETURN> key to continue..."  << endl;
  fgets( dummy_buffer, 2, stdin );
}


/*-------------------------------------------------------------*/
//   Name:        EnterTypeOfSource
//   Parameters:  None
//   Returns:     BERNOULLI or ON_OFF (SOURCE_TYPE)
//   Purpose:     This method asks the user to select the
//                type of Traffic Source, and returns either
//                BERNOULLI or ON_OFF.
/*-------------------------------------------------------------*/
int UserInterface::EnterTypeOfSource( void )
{
  char done = FALSE;
  char source = '0';
  int source_type = 0;

  while( !done )
  {
    cout << "Select Type of Traffic Source";
    cout << endl;
    cout << "(1) Bernoulli   (2) On_Off: ";
    cin  >> source;
    if( source == '1' )
    {
      done = TRUE;
    }
    else if( source == '2')
    {
      done = TRUE;
    }
    else
    {
      done = FALSE;
```

```
      cout << "Wrong choice, try again...";
      cout << endl;
    }
  }
  source_type = atoi(&source );

  return( source_type );
}


/*-------------------------------------------------------------*/
//   Name:        EnterTotalTime
//   Parameters:  None
//   Returns:     total_time (int)
//   Purpose:     This functions asks the user to enter the
//                total time in cell slots to run simulation.
/*-------------------------------------------------------------*/
int UserInterface::EnterTotalTime( void )
{
  int total_time = 0;
  cout << "Enter nr of cells slots for simulation: ";
  cin  >> total_time;
  return( total_time );
}

/*-------------------------------------------------------------*/
//   Name:        EnterSeed
//   Parameters:  None
//   Returns:     Seed (int)
//   Purpose:     This functions asks the user to enter the
//                initial seed to be used by first Traffic Source
/*-------------------------------------------------------------*/
int UserInterface::EnterSeed( void )
{
  int seed = 0;
  cout << "Enter Seed:                              ";
  cin  >> seed;
  return( seed );
}

/*-------------------------------------------------------------*/
//   Name:        EnterMultiplier
//   Parameters:  None
//   Returns:     multiplier (int)
//   Purpose:     This functions asks the user to enter the
```

```
//                initial multiplier to be used by first Traffic
//                Source
/*-------------------------------------------------------------*/
int UserInterface::EnterMultiplier( void )
{
  int multiplier = 0;
  char done = FALSE;
  char entry = '\0';

  while( !done )
  {
    cout << "Enter Multiplier (1, 2, ..., 8):        ";
    cin  >> entry;
    if( entry > '8' || entry < '1' )
    {
      done = FALSE;
      cout << "Wrong choice, try again...";
      cout << endl;
    }
    else
    {
      done = TRUE;
    }
  }
  multiplier = atoi( &entry );
  return( multiplier );
}

/*-------------------------------------------------------------*/
//   Name:        EnterNrPorts
//   Parameters:  None
//   Returns:     switch size (int)
//   Purpose:     This function asks the user to enter the number
//                of ports for the ATM switch.
/*-------------------------------------------------------------*/
int UserInterface::EnterNrPorts( void )
{
  int nr_ports = 0;
  cout << "Enter NR of ports for the ATM switch:   ";
  cin  >> nr_ports;
  return( nr_ports );
}

/*-------------------------------------------------------------*/
//   Name:        EnterNrIterations
```

```
//   Parameters:  None
//   Returns:     nr_interations (int)
//   Purpose:     This functions asks the user to enter the
//                Nr of Iterations for the simulation.  Iterations
//                is a parameter specific for the RGS algorithm.
/*-------------------------------------------------------------*/
int UserInterface::EnterNrIterations( void )
{
  char user_input = '\0';
  char done = FALSE;
  int nr_iterations = 0;

  while( !done )
  {
    cout << "Enter Nr of Iterations (1, 2, ..., 6) : ";
    cin  >> user_input;
    if( user_input > '6' || user_input < '1' )
    {
      done = FALSE;
      cout << endl;
      cout << "This is a wrong entry!!!";
      cout << endl;
      cout << "Please, try again...";
      cout << endl;
    }
    else
    {
      done = TRUE;
    }
  }

  nr_iterations = atoi( &user_input );
  return( nr_iterations );
}



/*-------------------------------------------------------------*/
//   Name:        EnterAvgLoad
//   Parameters:  None
//   Returns:     load (float)
//   Purpose:     This functions asks the user to enter the
//                average load offered by the traffic source.
//                The offered load is the probability of success
//                for the Bernoulli Traffic.  In the case of the
```

```
//                OnOff Markov traffic, the probability of the
//                source being in the On state is the offered load.
/*-------------------------------------------------------------*/
double UserInterface::EnterAvgLoad( int source_type )
{
  double avg_load = 0.0;
  if( source_type == BERNOULLI )
  {
    cout << "Enter  offered load (0 < load =< 1):    ";
    cin  >> avg_load;
  }
  else
  {
    cout << "Enter avg offered load (0< load <1):    ";
    cin  >> avg_load;
  }
  return( avg_load );
}

/*-------------------------------------------------------------*/
//   Name:        EnterAvgCellBurstLength
//   Parameters:  None
//   Returns:     avg_length (float)
//   Purpose:     This functions asks the user to enter the
//                average cells burst length.  The average
//                cells burst length is the mean of the geometric
//                distribution which describes the arrivals of
//                cells in the queue.  This parameter is only
//                used for the OnOff Traffic Source.
/*-------------------------------------------------------------*/
double UserInterface::EnterAvgCellBurstLength( void )
{
  double avg_length = 0.0;
  cout << "Enter average cell burst length:        ";
  cin  >> avg_length;
  return( avg_length );
}

/*-------------------------------------------------------------*/
//   Name:        WrongEntry
//   Parameters:  None
//   Returns:     void
//   Purpose:     This method indicates that user entered a wrong
//                input value for a parameter.
/*-------------------------------------------------------------*/
```

```
void UserInterface::WrongEntry( void )
{
  cout << endl;
  cout << "This is a wrong entry!!!";
  cout << endl;
  cout << "Refer to Master's project paper, and try again...";
  cout << endl;
}

/*-------------------------------------------------------------*/
//   Name:        WrongEntry
//   Parameters:  None
//   Returns:     void
//   Purpose:     This method indicates that user entered a wrong
//                input value for a the average length (ON_OFF).
/*-------------------------------------------------------------*/
void UserInterface::WrongEntry( double min_length)
{
  cout << endl;
  cout << "This is a wrong entry!!!";
  cout << endl;
  cout << "Minimum average length for the load entered is: ";
  cout << min_length;
  cout << endl;
  cout << "Refer to Master's project paper for further information.";
  cout << endl;
  cout << "Then, try again...";
  cout << endl;
}


/*-------------------------------------------------------------*/
//   Name:        WaitMessage
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This method displays a waiting message on screen.
/*-------------------------------------------------------------*/
void  UserInterface::WaitMessage( void )
{
  cout << "Please, Wait ..." << endl;
}
////////////////////////////////////////////////////////////////////
//                                                                //
//    Filename:         traffic.h                                 //
//                                                                //
```

```
//    Author:           Rubens S. Gomes                                 //
//    Date:             October, 1996.                                  //
//    Site:             Electrical Engineering & Computer Science       //
//                      The University of Kansas                        //
//                      Lawrence, KS, 66045                             //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows 95              //
//    Language:         C++                                             //
//    Node Location:    noehter.eecs.ukans.edu                          //
//    Path:             ~rgomes/MS_Proj/Simulation/traffic.h            //
//                                                                      //
//    Purpose:          Header file for the Traffic Source              //
//                      object.  This file defines the data member      //
//                      and operations performed by the object.         //
//                                                                      //
//    Limitations:      None.                                           //
//                                                                      //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


#if !defined _TRAFFIC_HPP
#define      _TRAFFIC_HPP

typedef enum
{
  ON,
  OFF
} TRAFFIC_STATES;

class SimulationParameter;
class ATM_Cell;
class RandNrGenerator;


class TrafficSource
{
  public:
    TrafficSource( SimulationParameter* Parm );
    ~TrafficSource( void );
    virtual ATM_Cell* GenerateCell( int input, long time ) = 0;

  private:
    void   SetTrafficParameters( SimulationParameter* Parm );
    double AVG_LOAD;
    double AVG_LENGTH;
```

```
    int    SOURCE_TYPE;
    int    SWITCH_SIZE;

  protected:
    double AvgLoad( void );
    double AvgLength( void );
    int    SwitchSize( void );
};


class BernoulliSource: public TrafficSource
{
  public:
    BernoulliSource( SimulationParameter* Parm );
    ~BernoulliSource( void );
    virtual ATM_Cell* GenerateCell( int input, long time );

  private:
    void SetBernParameters( void );
    RandNrGenerator* rand;
    double PROB_SUCCESS;
};


class OnOffSource: public TrafficSource
{
  public:
    OnOffSource( SimulationParameter* Parm );
    ~OnOffSource( void );
    virtual ATM_Cell* GenerateCell( int input, long time );

  private:
    void  SwitchTrafficState( void );
    void  SetOnOffParameters( void );
    RandNrGenerator* rand;
    int first_time;
    int LastOutputPort;
    double PROB_ON_TO_ON;
    double PROB_OFF_TO_OFF;
    TRAFFIC_STATES traffic_state;
};

#endif
//
// File:      traffic.C
```

```
//

#include "traffic.h"
#include "atm_par.h"

/*-------------------------------------------------------------*/
//   Name:        TrafficSource
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     constructor function for the TrafficSource
/*-------------------------------------------------------------*/
TrafficSource::TrafficSource( SimulationParameter* Parm)
{
  SetTrafficParameters( Parm );
}


/*-------------------------------------------------------------*/
//   Name:        TrafficSource
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     destructor function for the TrafficSource
/*-------------------------------------------------------------*/
TrafficSource::~TrafficSource( void )
{
}


/*-------------------------------------------------------------*/
//   Name:        SetTrafficParameters
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     sets the type of source being used.
/*-------------------------------------------------------------*/
void TrafficSource::SetTrafficParameters( SimulationParameter* Parm)
{
  AVG_LOAD = Parm->GetOfferedLoad();
  AVG_LENGTH = Parm->GetCellBurstLength();
  SWITCH_SIZE = Parm->GetNrPorts();
}

/*-------------------------------------------------------------*/
//   Name:        AvgLoad
//   Parameters:  None
//   Returns:     average load (double)
```

```
//   Purpose:     returns Average Load.
/*-------------------------------------------------------------*/
double TrafficSource::AvgLoad( void )
{
  return( AVG_LOAD);
}


/*-------------------------------------------------------------*/
//   Name:        AvgLength
//   Parameters:  None
//   Returns:     average length (double)
//   Purpose:     returns Average Length.
/*-------------------------------------------------------------*/
double TrafficSource::AvgLength( void )
{
  return( AVG_LENGTH );
}


/*-------------------------------------------------------------*/
//   Name:        SwitchSize
//   Parameters:  None
//   Returns:     switch size (int)
//   Purpose:     returns switch size for rand nr generator.
/*-------------------------------------------------------------*/
int TrafficSource::SwitchSize( void )
{
  return( SWITCH_SIZE );
}

//
// File:      bern.C
//

#include "traffic.h"
#include "atmcell.h"
#include "atm_par.h"
#include "rand.h"



/*-------------------------------------------------------------*/
//   Name:        BernoulliSource
//   Parameters:  None
```

```
//   Returns:     nothing
//   Purpose:     constructor method for the BernoulliSource
/*-------------------------------------------------------------*/
BernoulliSource::BernoulliSource( SimulationParameter* Parm ) :
                     TrafficSource( Parm )
{
  SetBernParameters();
  long seed = Parm->GetSeed();
  int multiplier = Parm->GetMultiplier();
  rand = new RandNrGenerator( multiplier , seed );
  Parm->UpdateSeed();
  Parm->UpdateMultiplier();
}

/*-------------------------------------------------------------*/
//   Name:        ~BernoulliSource
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     destructor method for the BernoulliSource
/*-------------------------------------------------------------*/
BernoulliSource::~BernoulliSource( void )
{
}

/*-------------------------------------------------------------*/
//   Name:        GenerateCell
//   Parameters:  int input, long time
//   Returns:     pointer to an ATM_Cell;  if this pointer points
//                to NIL, the cell was not created.
//   Purpose:     generates a cell.
/*-------------------------------------------------------------*/
ATM_Cell* BernoulliSource::GenerateCell( int input, long time )
{
  ATM_Cell* cell;
  if( rand->GetUnif01() <= PROB_SUCCESS )
  {
    int nr_ports = SwitchSize();
    cell = new ATM_Cell;
    cell->output_port = rand->RandDiscUnifValue(0, (nr_ports - 1) );
    cell->input_port = input;
    cell->arrival_time = time;
  }
  else
  {
    cell = '\0';
```

```
  }
  return( cell );
}


/*-------------------------------------------------------------*/
//   Name:        SetProbSuccess
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     sets probability of success
/*-------------------------------------------------------------*/
void BernoulliSource::SetBernParameters( void )
{
  PROB_SUCCESS = AvgLoad();
}

//
// File:      onoff.C
//

#include "traffic.h"
#include "atmcell.h"
#include "atm_par.h"
#include "rand.h"




/*-------------------------------------------------------------*/
//   Name:        OnOffSource
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     constructor method for the OnOffSource
/*-------------------------------------------------------------*/
OnOffSource::OnOffSource( SimulationParameter* Parm ) :
                        TrafficSource( Parm )
{
  SetOnOffParameters();
  traffic_state = ON;
  first_time = TRUE;
  LastOutputPort = 0;
  long seed = Parm->GetSeed();
  int multiplier = Parm->GetMultiplier();
  rand = new RandNrGenerator( multiplier , seed );
  Parm->UpdateSeed();
```

```
  Parm->UpdateMultiplier();
}

/*-------------------------------------------------------------*/
//   Name:        OnOffSource
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     destructor method for the OnOffSource
/*-------------------------------------------------------------*/
OnOffSource::~OnOffSource( void )
{
}

/*-------------------------------------------------------------*/
//   Name:        SetOnOffParameters
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This function calculates the probabilities
//                of On-to-On and Off-to-Off based on the avg.
//                offered load and avg. cell burst length.
//                For more information on how the equations
//                in this function was determined, please refer
//                to my master's project paper.
/*-------------------------------------------------------------*/
void OnOffSource::SetOnOffParameters( void )
{
  //
  // Avg. Cell Burst Length = 1 / prob. ON_to_OFF
  //  "     "    "      "   = 1 / (1 - prob. ON_to_ON )
  // Avg. Load              = "ON" Steady State Prob.
  //

  PROB_ON_TO_ON = 1.0 - ( 1.0 / AvgLength() );

  // the following formula is determined by solving for
  // the steady state probabilities assuming that the
  // traffic source starts out in the ON state.

  PROB_OFF_TO_OFF = (1.0 - AvgLoad() - ( AvgLoad() / AvgLength() )) /
                    (1.0 - AvgLoad() );
}

/*-------------------------------------------------------------*/
//   Name:        SwitchTrafficState
//   Parameters:  None
```

```
//   Returns:     nothing
//   Purpose:     switches the state of the OnOff source
/*-------------------------------------------------------------*/
void OnOffSource::SwitchTrafficState( void )
{
  if( traffic_state == ON )
  {
    traffic_state = OFF;
  }
  else
  {
    traffic_state = ON;
  }
}

/*-------------------------------------------------------------*/
//   Name:        GenerateCell
//   Parameters:  int input, long time
//   Returns:     pointer to an ATM_Cell;  if this pointer points
//                to NIL, the cell was not created.
//   Purpose:     generates a cell.
/*-------------------------------------------------------------*/
ATM_Cell* OnOffSource::GenerateCell( int input, long time )
{
  ATM_Cell* cell;
  int out_port = 0;
  int nr_ports = SwitchSize();

  if( traffic_state == ON )
  {
    cell = new ATM_Cell;
    if( rand->GetUnif01() <= PROB_ON_TO_ON )
    {
      if( first_time )
      {
        out_port  = rand->RandDiscUnifValue(0, (nr_ports - 1) );
        first_time = FALSE;
        LastOutputPort = out_port;
      }
      else // was in ON state prior to current cell slot
      {
        out_port = LastOutputPort;
      }
      cell->output_port = out_port;
      cell->input_port = input;
```

```
      cell->arrival_time = time;
    }
    else
    {
      if( first_time )
      {
        out_port  = rand->RandDiscUnifValue(0, (nr_ports - 1) );
      }
      else
      {
        out_port = LastOutputPort;
      }
      cell->output_port = out_port;
      cell->input_port = input;
      cell->arrival_time = time;
      first_time = TRUE;
      SwitchTrafficState();
    }
  }
  else // Traffic Source is in the OFF state
  {
    if( !(rand->GetUnif01() <= PROB_OFF_TO_OFF) )
    {
      SwitchTrafficState();
    }
    cell = '\0';
  }
  return( cell );
}

////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         rand.h                                        //
//                                                                    //
//    Author:           Rubens S. Gomes                               //
//    Date:             August, 1996.                                 //
//    Site:             Electrical Engineering & Computer Science     //
//                      The University of Kansas                      //
//                      Lawrence, KS, 66045                           //
//    Operating System: UNIX SunOS                                    //
//    Language:         C++                                           //
//    Node Location:    noehter.eecs.ukans.edu                        //
//    Path:             ~rgomes/MS_Proj/Simulation/rand.h             //
//                                                                    //
//    Purpose:          Header file for the Randon Number Generator   //
```

```
//                      object.  This file defines the data member   //
//                      and operations performed by the object.      //
//                                                                   //
//                                                                   //
//    Limitations:      None.                                        //
//                                                                   //
//                                                                   //
///////////////////////////////////////////////////////////////////////

#include <math.h>

#if !defined _RAND_HPP
#define      _RAND_HPP 1

const int CAPTIONLENGTH = 32;

class RandNrGenerator
{

 public:
    RandNrGenerator( int _multiplier, long _seed );
    double GetUnif01( void );
    long GetMultiplier( void )   { return multiplier; }
    long GetSeed( void )         { return seed; }
    void SetSeed( long _seed );
    int RandDiscUnifValue( int lowerBound, int upperBound );

      //-- Prime modulus multiplicative congruential generator --//
  private:
    static long defaultSeed;         // increments seeds by 1,000,000

    long seed;                       // current RNG seed
    long multiplier;                 // RNG multiplier

    void SetDefaultSeed( void );
    long GetDefaultSeed( void );

    void SelectMultiplierType ( int _multiplier );
    void SelectMultiplierType1( void ) { multiplier =  950706376; }
    void SelectMultiplierType2( void ) { multiplier =  742938285; }
    void SelectMultiplierType3( void ) { multiplier = 1226874159; }
    void SelectMultiplierType4( void ) { multiplier =   62089911; }
    void SelectMultiplierType5( void ) { multiplier = 1343714438; }
    void SelectMultiplierType6( void ) { multiplier =  630360016; }
    void SelectMultiplierType7( void ) { multiplier =  397204094; }
```

```
    void SelectMultiplierType8( void ) { multiplier =      16807; }

};

#endif
//
//  File:     rand.C
//


#include "rand.h"

const long LONG_MAX=2147483647;

long RandNrGenerator::defaultSeed = 0;


/*-------------------------------------------------------------*/
//   Name:        RandNrGenerator
//   Parameters:  multiplier (input, int)
//                seed       (input, long)
//   Returns:     nothing
//   Purpose:     constructor function for the RandNrGenerator
/*-------------------------------------------------------------*/
RandNrGenerator::RandNrGenerator( int _multiplier, long _seed )
{
   SelectMultiplierType( _multiplier );
   SetSeed( _seed );
}

/*-------------------------------------------------------------*/
//   Name:        SelectMultiplier
//   Parameters:  multiplier choice (input, int)
//   Returns:     nothing
//   Purpose:     This function sets the multiplier value to
//                be used in the generation of a uniform random
//                number
/*-------------------------------------------------------------*/
void RandNrGenerator::SelectMultiplierType( int _multiplier )
{
  switch( _multiplier )
  {
     case 2:
    {
        SelectMultiplierType2();
```

```
 }
 break;

  case 3:
 {
     SelectMultiplierType3();
 }
 break;

  case 4:
 {
     SelectMultiplierType4();
 }
 break;

  case 5:
 {
     SelectMultiplierType5();
 }
 break;

  case 6:
 {
     SelectMultiplierType6();
 }
 break;

  case 7:
 {
     SelectMultiplierType7();
 }
 break;

  case 8:
 {
     SelectMultiplierType8();
 }
 break;

  default:
 {
     SelectMultiplierType1();
 }
  break;
}
```

```
}

/*-------------------------------------------------------------*/
//   Name:        SetSeed
//   Parameters:  seed (input, long)
//   Returns:     nothing
//   Purpose:     This function sets the seed value to be used
//                when generating first uniform random number.
/*-------------------------------------------------------------*/
void RandNrGenerator::SetSeed( long _seed )
{
   if ( (_seed <= 0) || (_seed >= LONG_MAX) )
  {
    SetDefaultSeed();
  }
   else
  {
     seed = _seed;
  }
}

/*-------------------------------------------------------------*/
//   Name:        SetDefaultSeed
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     This function sets the default seed in case
//                the user enters a seed out of a pre-specified
//                range.
/*-------------------------------------------------------------*/
void RandNrGenerator::SetDefaultSeed( void )
{
  seed = GetDefaultSeed();
}

/*-------------------------------------------------------------*/
//   Name:        GetDefaultSeed
//   Parameters:  none
//   Returns:     default seed value
//   Purpose:     This function returns the number to be used
//                as the default seed.
/*-------------------------------------------------------------*/
long RandNrGenerator::GetDefaultSeed( void )
{
   if ( defaultSeed >= 2145000000 )
   {
```

```
      defaultSeed = 0;
   }

  defaultSeed= defaultSeed + 1000000;
  return defaultSeed;
}



/*-------------------------------------------------------------*/
//   Name:        GetUniformNr
//   Parameters:  none
//   Returns:     uniform number between 0 and 1
//   Purpose:     This function uses the congruetional mod
//                formula to return a uniform number between
//                0 and 1.
/*-------------------------------------------------------------*/
double RandNrGenerator::GetUnif01( void )
{
   seed = (long) fmod(( (long double) multiplier * (long double) seed ),
                         (long double) LONG_MAX );
   return( (double) seed / (double) LONG_MAX );
}



/*-------------------------------------------------------------*/
//   Name:        RandDiscUnifValue
//   Parameters:  lowerBound ( input, int )
//                upperBound ( input, int )
//   Returns:     uniform nr between lowerBound and upperBound
//   Purpose:     This function returns a uniform random number
//                between lowerBound and upperBound.  This
//                function may be used when selecting the
//                output port nr for a burst of arriving cells..
/*-------------------------------------------------------------*/
int RandNrGenerator::RandDiscUnifValue( int lowerBound,
                                        int upperBound )
{
  int largestInt;

  largestInt = upperBound - lowerBound + 1;
  largestInt = (int) floor( (double) largestInt * GetUnif01() );

  return ( lowerBound + largestInt );
}
```

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         queue.h                                       //
//                                                                    //
//    Author:           Rubens S. Gomes                               //
//    Date:             August, 1996.                                 //
//    Site:             Electrical Engineering & Computer Science     //
//                      The University of Kansas                      //
//                      Lawrence, KS, 66045                           //
//    Operating System: UNIX SunOS                                    //
//    Language:         C++                                           //
//    Node Location:    noehter.eecs.ukans.edu                        //
//    Path:             ~rgomes/MS_Proj/Simulation/queue.h            //
//                                                                    //
//    Purpose:          Header file for the Input Queue object.  This//
//                      file defines the data member and operations   //
//                      performed by the object.                      //
//                                                                    //
//                                                                    //
//    Limitations:      None.                                         //
//                                                                    //
//                                                                    //
////////////////////////////////////////////////////////////////////////


#include <assert.h>
#include <string.h>
#include "list.h"

#if !defined _INPUTQUEUE_HPP
#define      _INPUTQUEUE_HPP

#define      INPUT_BUFFER_CAPACITY  ( 50000 )

class ATM_Cell;

class InputQueue
{
  public:
    InputQueue( void );
    void    StoreCell( ATM_Cell* cell );
     long    CellArrivalTime( int Position );
```

```
     long    CellElapsedTime( int Position,
                              long cur_time );
    int     CellOutputPort( int Position );
    int     IsQueueEmpty( void );
     void    RemoveCell( int Position );
    int     IsThereCellAt( int Position );
    int     RetrieveNrCellsDeparted( void );
    void    IncrementNrCellsStored( void );
    void    DecrementNrCellsStored( void );
    int     RetrieveNrCellsStored( void );
    void    IncrementCellsLost( void );
    int     RetrieveCellsLost( void );

  private:
     LinkedList Queue;
     int LastOutputPort;
     int NrCellsDeparted;
     int NrCellsStored;
     int CellsLost;
};

#endif
//
//  File:     queue.C
//

#include "queue.h"
#include "atmcell.h"
#include "list.h"
#include "atm_par.h"



/*-------------------------------------------------------------*/
//   Name:        InputQueue
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     constructor function for the InputQueue object
/*-------------------------------------------------------------*/
InputQueue::InputQueue( void )
{
  LastOutputPort = 0;
  NrCellsDeparted = 0;
  NrCellsStored = 0;
  CellsLost = 0;
```

```
}


/*--------------------------------------------------------------------*/
```

```
  int Success;
  ATM_Cell cell;

  Queue.ListRetrieve( Position, cell, Success );

  /* use assert to check validility of ListRetrieve */
  assert( Success );

  return( cell.arrival_time );
}

/*-------------------------------------------------------------*/
//   Name:        CellElapsedTime
//   Parameters:  Position  (input, int) position of node.
//                cur_time  (input, long) current cell slot time
//   Returns:     Elapsed time the cell spent in the queue since
//                its arrival. This time is based on the number
//                of cell slots that have elapsed between the
//                arrival of a cell and its departure.
//   Purpose:     This function calculates the cell elapsed time,
//                which is the difference between the current
//                simulation cell slot time and the arrival cell
//                slot time.
/*-------------------------------------------------------------*/
long InputQueue::CellElapsedTime( int Position, long cur_time )
{
   long arrival_time = 0;
  long elapsed_time = 0;

   arrival_time = CellArrivalTime( Position );
  elapsed_time = (cur_time + ITERATION_TIME_DELAY) - arrival_time;

   return( elapsed_time );
}

/*-------------------------------------------------------------*/
//   Name:        CellOutputPort
//   Parameters:  Position  (input, int) position of node.
//   Returns:     Output port number for the position node.
//   Purpose:     This function returns the output port nr of
//                the cell specified in the passed Position
//                field.
/*-------------------------------------------------------------*/
int InputQueue::CellOutputPort( int Position )
{
```

```
  int Success;
  ATM_Cell cell;

  Queue.ListRetrieve( Position, cell, Success );

  // use assert to check validility of ListRetrieve
  assert( Success );

  return( cell.output_port );
}

/*-------------------------------------------------------------*/
//   Name:        IsQueueEmpty
//   Parameters:  none
//   Returns:     TRUE or FALSE
//   Purpose:     This function returns TRUE if the queue is
//                empty; otherwise, it returns FALSE.
/*-------------------------------------------------------------*/
int InputQueue::IsQueueEmpty( void )
{
  return( Queue.ListIsEmpty() );

}


/*-------------------------------------------------------------*/
//   Name:        RemoveCell
//   Parameters:  Position (input, int) node position
//   Returns:     nothing
//   Purpose:     This function deletes the node from the input
//                queue which is specified by the Position.
/*-------------------------------------------------------------*/
void InputQueue::RemoveCell( int Position )
{
  int Success;
  Queue.DeleteNode( Position, Success );
  NrCellsDeparted++;
}


/*-------------------------------------------------------------*/
//   Name:        RetrieveNrCellsDeparted
//   Parameters:  void
//   Returns:     NrCellsLeaving (int)
//   Purpose:     This function retrieves the number of cells
```

```
//                departed from this queue.
/*-------------------------------------------------------------*/
int InputQueue::RetrieveNrCellsDeparted( void )
{
  return( NrCellsDeparted );
}

/*-------------------------------------------------------------*/
//   Name:        IncrementNrCellsStored
//   Parameters:  void
//   Returns:     void
//   Purpose:     This function increments the number of cells
//                currently stored in this input queue.
/*-------------------------------------------------------------*/
void InputQueue::IncrementNrCellsStored( void )
{
  NrCellsStored++;
}

/*-------------------------------------------------------------*/
//   Name:        DecrementNrCellsStored
//   Parameters:  void
//   Returns:     void
//   Purpose:     This function decrements the number of cells
//                currently stored in this input queue.
/*-------------------------------------------------------------*/
void InputQueue::DecrementNrCellsStored( void )
{
  NrCellsStored--;
}

/*-------------------------------------------------------------*/
//   Name:        RetrieveNrCellsStored
//   Parameters:  void
//   Returns:     NrCellsStored (int)
//   Purpose:     This function returns the number of cells
//                currently stored in this input queue.
/*-------------------------------------------------------------*/
int InputQueue::RetrieveNrCellsStored( void )
{
  return( NrCellsStored );
}


/*-------------------------------------------------------------*/
```

```
//   Name:        IncrementCellsLost
//   Parameters:  void
//   Returns:     void
//   Purpose:     This function increments the number of cells
//                dropped due to buffer overflow.
/*-------------------------------------------------------------*/
void InputQueue::IncrementCellsLost( void )
{
  CellsLost++;
}

/*-------------------------------------------------------------*/
//   Name:        RetrieveCellsLost
//   Parameters:  void
//   Returns:     Cells Lost(int)
//   Purpose:     This function returns the number of cells
//                dropped due to buffer overflow.
/*-------------------------------------------------------------*/
int InputQueue::RetrieveCellsLost( void )
{
  return( INPUT_BUFFER_CAPACITY );
}
////////////////////////////////////////////////////////////////////
//                                                                //
//    Filename:         list.h                                    //
//                                                                //
//    Author:           Rubens S. Gomes                           //
//    Date:             August, 1996.                             //
//    Site:             Electrical Engineering & Computer Science //
//                      The University of Kansas                  //
//                      Lawrence, KS, 66045                       //
//    Operating System: UNIX SunOS                                //
//    Language:         C++                                       //
//    Node Location:    noehter.eecs.ukans.edu                    //
//    Path:             ~rgomes/MS_Proj/Simulation/list.h         //
//                                                                //
//    Purpose:          Header file for the Doubly Linked List    //
//                      object.  This file defines the data member//
//                      and operations performed by the object.   //
//                                                                //
//                                                                //
//    Limitations:      None.                                     //
//                                                                //
//                                                                //
////////////////////////////////////////////////////////////////////
```

```
#include "atmcell.h"
#include <stddef.h>

#include <stdio.h>   // these header files are required
#include <assert.h>  // in order to use the assert run
#include <string.h>  // time library.

#define  TRUE  ( 1 )
#define  FALSE ( 0 )

#if !defined _LIST_HPP
#define      _LIST_HPP

struct  listNode;      // defined in the implementation file.

typedef listNode* ptrType;

class LinkedList
{

  public:
    LinkedList( void );
    ~LinkedList( void );
    int ListIsEmpty( void );
    int     ListSize( void );
    void    ListAddTail( ATM_Cell* NewItem );
    void    DeleteNode( int Position,
                        int& Success );
    void    ListRetrieve( int Position,
                          ATM_Cell& DataItem,
                          int& Success );

  private:
    ptrType PtrTo( int Position );
    ptrType Tail;     // points to last cell in the linked list
    ptrType PrevTail; // points to cell previous to last cell in the list
    int    Size;
    ptrType Head;

};

#endif
//
//  File:      List.cpp
```

```
//

#include "list.h"

struct listNode  // node on the doubly linked list
{
  ATM_Cell* Item;     // a node item on the list
  ptrType Next;       // pointer to next node
  ptrType Previous;   // pointer to previous node
};


/*-------------------------------------------------------------*/
//   Name:        LinkedList
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     Constructor function for the  Linked List. All
//                that needs to do is to initialize pointers.
/*-------------------------------------------------------------*/
LinkedList::LinkedList( void )
{
  Size = 0;
  Head = new listNode;
  Head->Next = NULL;
  Head->Previous = NULL;
  Head = NULL;
  Tail = NULL;
  PrevTail = NULL;
}

/*-------------------------------------------------------------*/
//   Name:        ~LinkedList
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     Destructor function for the  Linked List. All
//                that needs to do is to deallocate memory.
/*-------------------------------------------------------------*/
LinkedList::~LinkedList( void )
{
  int Success;

  while( !ListIsEmpty() )
  {
    DeleteNode( 1, Success );
  }
```

```
  // after deleting all the nodes, delete the Head...
  delete Head;
}

/*-------------------------------------------------------------*/
//   Name:        ListIsEmpty
//   Parameters:  None
//   Returns:     FALSE or TRUE
//   Purpose:     This function returns TRUE if the list is
//                empty; otherwise, it returns FALSE.
/*-------------------------------------------------------------*/
int LinkedList::ListIsEmpty( void )
{
  if( Size == 0 )
  {
    return( TRUE );
  }
  else
  {
    return( FALSE );
  }
}

/*-------------------------------------------------------------*/
//   Name:        ListSize
//   Parameters:  None
//   Returns:     size of the list
//   Purpose:     This function returns the number of nodes in
//                the input queue.
/*-------------------------------------------------------------*/
int LinkedList::ListSize( void )
{
  return Size;
}

/*-------------------------------------------------------------*/
//   Name:        PtrTo
//   Parameters:  Position (input, int) - node position
//   Returns:     Pointer to that node.
//   Purpose:     This function locates a specified node in
//                the linked list, then it returns a pointer to
//                that node which is located at the position
//                number passed in.
/*-------------------------------------------------------------*/
ptrType LinkedList::PtrTo( int Position )
```

```
{
  int Skip;      // used as counter for the for loop
  ptrType Trav;  // pointer to traverse the linked list

  if( (Position < 1) || (Position > ListSize()) )
  {
     return NULL;
  }
  else
  {
    Trav = Head;
    for( Skip = 1; Skip < Position; ++Skip )
    {
      Trav = Trav->Next;
    }
    return Trav;
  }
}


/*-------------------------------------------------------------*/
//   Name:        ListRetrieve
//   Parameters:  Position     (input, int) - node position
//                DataItem     (input, ATM_Cell) - node
//                Success      (input, int) - TRUE or FALSE
//   Returns:     nothing
//   Purpose:     This function retrieves the node item from the
//                position indicated by the Position value.
/*-------------------------------------------------------------*/
void LinkedList::ListRetrieve( int Position,
                               ATM_Cell& DataItem,
                               int& Success )
{
  ptrType Cur;   // current pointer to the linked list.

  Success = (Position >= 1 ) && (Position <= ListSize());

  // use assert to check validility of Success
  assert( Success );
  if( Position == Size )
  {
    // Cell to retrieve is located in the end of the list
    Cur = Tail;
  }
  else if( Position == (Size-1) )
```

```
  {
    // Cell to retrieve is located before last cell in the list
    Cur = PrevTail;
  }
  else
  {
    Cur = PtrTo( Position );
  }
  DataItem = *Cur->Item;
}

/*-----------------------------------------------------------------*/
//   Name:        DeleteNode
//   Parameters:  Position   (input, int) - node position
//                Success    (input, int) - TRUE or FALSE
//   Returns:     nothing
//   Purpose:     This function is responsible for deleting an
//                item located in the Position value from the
//                linked list.
/*-----------------------------------------------------------------*/
void LinkedList::DeleteNode( int Position,
                             int& Success )
{
  ptrType Cur;        // current pointer to linked list
  ptrType AfterCur;   // points to position after cur pointer
  ptrType Prev;       // points to position previous to cur pointer

  Success = (Position >= 1 ) && (Position <= ListSize());

  // use assert to check validility of Success
  assert( Success );

  if( Position == 1 )
  {
    //
    // Delete ATM cell from the head of the list
    //
    Cur = Head;
    Head = Cur->Next;
    if( Size == 1 )
    {
      // if list has only one cell, the Tail and PrevTail
      // need to be relocated...
      Tail = Head;
      PrevTail = Head;
```

```
  }
  else if( Size == 2 )
  {
    // need to move PrevTail to point to location previous to Tail
    PrevTail = Tail->Previous;
  }
}
else if( Position == (Size-1) )
{
  //
  // Delete ATM cell previous to PrevTail...
  //
  Prev = PrevTail->Previous;
  Cur = Prev->Next;
  Prev->Next = Cur->Next;
  // move previous pointer of node pointed by tail to cell located
  // behind cell being deleted...
  Tail->Previous = Prev;
  //
  // Because the cell previous to the last cell is being deleted, we
  // need to move the PrevTail pointer to the same location as
  // the current Prev pointer.
  //
  PrevTail = Tail->Previous;
}
else if( Position == Size )
{
  //
  // Delete ATM cell at the end of the list ...
  //
  Prev = PrevTail;
  Cur = Prev->Next;
  Prev->Next = Cur->Next;
  //  Tail becomes current Prev pointer       //
  Tail = Prev;
  //  PrevTail becomes Prev->Previous pointer //
  PrevTail = Tail->Previous;
}
else
{
  //
  // Delete ATM cell from somewhere inside the list ...
  //
  Prev = PtrTo( Position - 1 );
  Cur = Prev->Next;
```

```
    AfterCur = Cur->Next;
    Prev->Next = Cur->Next;
    AfterCur->Previous = Prev;
  }

  Cur->Next     = NULL;
  Cur->Previous = NULL;
  delete Cur;
  Cur = NULL;
  --Size;
}


/*-------------------------------------------------------------*/
//   Name:        ListAddTail
//   Parameters:  NewItem (input, ATM_Cell) - node to insert
//   Returns:     nothing
//   Purpose:     This function adds a item node to the tail of
//                the linked list.
/*-------------------------------------------------------------*/
void LinkedList::ListAddTail( ATM_Cell* NewItem )
{
  ptrType NewPtr;          // new pointer to the linked list
  ptrType Prev;            // previous pointer to linked list

  NewPtr = new listNode;

  NewPtr->Item = NewItem;
  if( Size == 0 )
  {
    //
    // Insert new ATM cell at the head of the list
    //
    NewPtr->Next     = Head;
    NewPtr->Previous = Head;
    Head = NewPtr;
    // Because there is only one cell, PrevTail and Tail//
    // contain the same memory address.                 //
    Tail = NewPtr;
    PrevTail = NewPtr->Previous;
  }
  else
  {
    //
    // Insert new ATM cell at the end (tail) of the list
```

```
    //
    Prev = Tail;
    NewPtr->Next     = Prev->Next;
    NewPtr->Previous = Prev;
    Prev->Next = NewPtr;
    // current Tail becomes PrevTail //
    PrevTail = NewPtr->Previous;
    // Move Tail to last node        //
    Tail = NewPtr;
  }
  Size++;
}
////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         simu.h                                        //
//                                                                    //
//    Author:           Rubens S. Gomes                               //
//    Date:             October, 1996.                                //
//    Site:             Electrical Engineering & Computer Science     //
//                      The University of Kansas                      //
//                      Lawrence, KS, 66045                           //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows 95            //
//    Language:         C++                                           //
//    Node Location:    noehter.eecs.ukans.edu                        //
//    Path:             ~rgomes/MS_Proj/Simulation/simu.h             //
//                                                                    //
//    Purpose:          Header file for the Simulation object.        //
//                      This file defines the data member             //
//                      and operations performed by the object.       //
//                                                                    //
//    Limitations:      None.                                         //
//                                                                    //
//                                                                    //
////////////////////////////////////////////////////////////////////////


#if !defined _SIMU_HPP
#define      _SIMU_HPP

#define MAX_PORTS    (  65 )

class SimulationParameter;
class TrafficSource;
class InputQueue;
class StatisticalProbe;
```

```
class RGSProtocol;

class Simulation
{
  public:
    Simulation( SimulationParameter* Parm );
    ~Simulation( void );
    void Start( void );
    void CellSlotClock( long time );
    void printResults( void );

  private:
    void GenerateCells( long time, char steady_state );
    void RunAlgorithmTrans( void );
    void RunAlgorithmSteady( long time );
    int SOURCE_TYPE;
    int nr_ports;
    int nr_iterations;
    long transientTime;
    TrafficSource* Source[MAX_PORTS];
    InputQueue* Queue[MAX_PORTS];
    StatisticalProbe* Stats;
    RGSProtocol* rgs;
    SimulationParameter* Parameter;
};

#endif

//
// File:  simu.C
//

#include "simu.h"
#include "stats.h"
#include "atm_par.h"
#include "traffic.h"
#include "queue.h"
#include "rgs_prot.h"


/*-------------------------------------------------------------*/
//   Name:        Simulation
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     constructor method for the Simulation
```

```
/*-------------------------------------------------------------*/
Simulation::Simulation( SimulationParameter* Parm )
{
  int i = 0;

  Parameter = Parm;
  SOURCE_TYPE = Parameter->GetTypeOfSource();
  nr_ports = Parameter->GetNrPorts();
  nr_iterations = Parameter->GetNrIterations();
  transientTime = Parameter->GetTransientTime();

  if( SOURCE_TYPE == BERNOULLI )
  {
    for(i=0; i < nr_ports; i++)
    {
      Source[i] = new BernoulliSource( Parm );
    }
  }
  else
  {
    for(i=0; i < nr_ports; i++)
    {
      Source[i] = new OnOffSource( Parm );
    }
  }
  for(i=0; i < nr_ports; i++)
  {
    Queue[i] = new InputQueue;
  }
  Stats = new StatisticalProbe;
  rgs = new RGSProtocol( nr_ports );
}

/*-------------------------------------------------------------*/
//   Name:        Simulation
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     destructor method for the Simulation
/*-------------------------------------------------------------*/
Simulation::~Simulation( void )
{
  int i = 0;

  for(i=0; i < nr_ports; i++)
  {
```

```
    delete Source[i];
    delete Queue[i];
  }

  delete Stats;
  delete rgs;

}


/*-------------------------------------------------------------*/
//   Name:        CellSlotClock
//   Parameters:  long time
//   Returns:     nothing
//   Purpose:     at the beginning of each time slot the timer
//                sends a signal to the operation to indicate
//                the beginning of a cell slot.  Several tasks
//                should be performed: generate cell, store cell,
//                run the request-grant-status protocol
/*-------------------------------------------------------------*/
void Simulation::CellSlotClock( long time )
{
  if( time < transientTime )
  {
    GenerateCells( time, FALSE );
    Stats->UpdateTotalNrCellSlots();
    RunAlgorithmTrans();
  }
  else
  {
    GenerateCells( time, TRUE );
    Stats->UpdateSteadyStateNrCellSlots();
    Stats->UpdateTotalNrCellSlots();
    RunAlgorithmSteady( time );
  }
}

/*-------------------------------------------------------------*/
//   Name:        GenerateCells()
//   Parameters:  time (long), steady_state (char)
//   Returns:     nothing
//   Purpose:     generate cells to be stored in respective
//                input queues.
/*-------------------------------------------------------------*/
void Simulation::GenerateCells( long time, char steady_state )
```

```
{
  int i = 0;
  ATM_Cell* cell;

  for(i=0; i < nr_ports; i++)
  {
    cell = Source[i]->GenerateCell( i, time );
    if( cell ) // cell was created !
    {
      Queue[i]->StoreCell( cell );
      Queue[i]->IncrementNrCellsStored();
      if( steady_state )
      {
        Stats->UpdateSteadyStateNrCellsArrived();
        Stats->UpdateTotalNrCellsArrived();
      }
      else
      {
        Stats->UpdateTotalNrCellsArrived();
      }
    }
  }
}

/*-------------------------------------------------------------*/
//   Name:        RunAlgorithmTrans()
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     runs the RGS algorithm during transient state.
/*-------------------------------------------------------------*/
void Simulation::RunAlgorithmTrans( void )
{
  int i = 0;
  int Converged = FALSE;

  i = 0;
  while( i < nr_iterations  && !Converged )
  {
    rgs->runRGSRequest( &Queue[0] );
    rgs->runRGSGrant();
    rgs->runRGSStatus();
    Converged = rgs->HasAlgorithmConverged();
    i++;
  }
  rgs->RouteCellsTransState( &Queue[0], Stats );
```

```
  rgs->ResetIcuOcu();
}


/*-------------------------------------------------------------*/
//   Name:        RunAlgorithmSteady()
//   Parameters:  time ( long )
//   Returns:     nothing
//   Purpose:     runs the RGS algorithm during steady state.
/*-------------------------------------------------------------*/
void Simulation::RunAlgorithmSteady( long time )
{
  int i = 0;
  int Converged = FALSE;

  i = 0;
  while( i < nr_iterations  && !Converged )
  {
    rgs->runRGSRequest( &Queue[0] );
    rgs->runRGSGrant();
    rgs->runRGSStatus();
    Converged = rgs->HasAlgorithmConverged();
    i++;
  }
  rgs->RouteCellsSteadyState( &Queue[0], Stats, time );
  rgs->ResetIcuOcu();
}



/*-------------------------------------------------------------*/
//   Name:        printResults
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     method to print results when program is done
/*-------------------------------------------------------------*/
void Simulation::printResults( void )
{
  char dummy[80];

  cout << endl;
  cout << endl;
  cout << "         Simulation Results  " << endl;
  cout << endl;
  if( SOURCE_TYPE == BERNOULLI )
```

```
{
  cout << "Traffic type:                    BERNOULLI";
}
else
{
  cout << "Traffic type:                    ON_OFF";
}
cout << endl;
cout << "Switch size:                    " << nr_ports;
cout << endl;
cout << "Total Number Cell Slots:        " <<
        Stats->GetTotalNrCellSlots();
cout << " slots" << endl;
cout << "Total Nr Steady State Slots:    " <<
         Stats->GetSteadyStateNrCellSlots();
cout << " slots" << endl;
cout << "Number of Iterations:           " <<
        Parameter->GetNrIterations();
cout << endl;
cout << "Total Nr of cells arrived:      " <<
        Stats->GetTotalNrCellsArrived();
cout << " cells" << endl;
cout << "Total Nr of cells routed:       " <<
        Stats->GetTotalNrCellsRouted();
cout << " cells" << endl;
cout << "Steady State Nr cells arrived:  " <<
         Stats->GetSteadyStateNrCellsArrived();
cout << " cells" << endl;
cout << "Steady State Nr cells routed:   " <<
         Stats->GetSteadyStateNrCellsRouted();
cout << " cells" << endl;
cout << "Switch Offered Load:            " <<
        Parameter->GetOfferedLoad() * 100;
cout << " %";
cout << endl;
if( Parameter->GetTypeOfSource() == ON_OFF )
{
  cout << "Avg Cell Burst Length:          " <<
         Parameter->GetCellBurstLength();
  cout << endl;
}
cout << endl;
cout << "Avg Utilization of Output Ports: " <<
         Stats->OutputPortAvgUtilization( nr_ports );
cout << " %";
```

```
  cout << endl;
  cout << "Switch Throughput:               " << Stats->SwitchThroughput();
  cout << " %";
  cout << endl;
  cout << "Queueing Avg Cell Delay:         " <<
          Stats->QueueAvgCellDelay();
  cout << " sec.";
  cout << endl;
  cout << "Press Return to continue..." << endl;
  fgets( dummy, 2, stdin );
}
```

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         rgs_prot.h                                    //
//                                                                    //
//    Author:           Rubens S. Gomes                               //
//    Date:             October, 1996.                                //
//    Site:             Electrical Engineering & Computer Science     //
//                      The University of Kansas                      //
//                      Lawrence, KS, 66045                           //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows 95            //
//    Language:         C++                                           //
//    Node Location:    noehter.eecs.ukans.edu                        //
//    Path:             ~rgomes/MS_Proj/Simulation/rgs_prot.h         //
//                                                                    //
//    Purpose:          Header file for the RGS protocol object       //
//                      This file defines the data member             //
//                      and operations performed by the object.       //
//                                                                    //
//    Limitations:      None.                                         //
//                                                                    //
//                                                                    //
////////////////////////////////////////////////////////////////////////


#if !defined _RGSPROT_HPP
#define      _RGSPROT_HPP

#define MAX_PORTS  (  65 )

class InputQueue;
class SimulationParameter;
```

```
class StatisticalProbe;
class InputControlUnit;
class OutputControlUnit;

class RGSProtocol
{
  public:
    RGSProtocol( int NrPorts );
    ~RGSProtocol( void );
    void runRGSRequest( InputQueue* Queue[] );
    void runRGSGrant( void );
    void runRGSStatus( void );
    int HasAlgorithmConverged( void );
    void RouteCellsTransState( InputQueue* Queue[],
                               StatisticalProbe* Stats );
    void RouteCellsSteadyState( InputQueue* Queue[],
                                StatisticalProbe* Stats,
                                long cur_time );

    void ResetIcuOcu( void );

  private:
    InputControlUnit* ICU[MAX_PORTS];
    OutputControlUnit* OCU[MAX_PORTS];
    int nr_ports;
};

#endif


//
//  File:      rgs_prot.C
//

#include "rgs_prot.h"
#include "queue.h"
#include "icu_unit.h"
#include "ocu_unit.h"
#include "atm_par.h"
#include "stats.h"


/*-------------------------------------------------------------*/
//   Name:        RGSProtocol
//   Parameters:  None
```

```
//   Returns:     nothing
//   Purpose:     constructor method for the RGSProtocol object
/*-------------------------------------------------------------*/
RGSProtocol::RGSProtocol( int NrPorts)
{
  int i = 0;
  int mult = 0;
  long seed = 0;

  nr_ports = NrPorts;
  seed = 1000;  // initial seed
  mult = 1;     // initial multiplier entry.
  for( i = 0; i < nr_ports; i++ )
  {
    ICU[i] = new InputControlUnit;
    OCU[i] = new OutputControlUnit( mult, seed );
    // use different seed and multiplier for next
    // OCU units.  The OCU units use these seed
    // multiplier when creating a new random nr
    // generator.
    seed = seed + 5;
    mult = mult + 1;
  }
}

/*-------------------------------------------------------------*/
//   Name:        ~RGSProtocol
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     destructor method for the RGSProtocol object
/*-------------------------------------------------------------*/
RGSProtocol::~RGSProtocol( void )
{
  int i = 0;
  for( i = 0; i < nr_ports; i++ )
  {
    delete ICU[i];
    delete OCU[i];
  }
}

/*-------------------------------------------------------------*/
//   Name:        RunRGSRequest
//   Parameters:  InputQueue* Queue[]
//   Returns:     nothing
```

```
//   Purpose:     this method performs the request phase of the
//                RGS algorithm.
/*-------------------------------------------------------------*/
void RGSProtocol::runRGSRequest( InputQueue* Queue[] )
{
  int j = 0;

  for( j = 0; j < nr_ports; j++ )
  {
    if( !(ICU[j]->IsIRRBusy()) && !(Queue[j]->IsQueueEmpty()) )
    {
      ICU[j]->runRequest(j, Queue[j], &OCU[0]);
    }
  }
}

/*-------------------------------------------------------------*/
//   Name:        RunRGSGrant
//   Parameters:  void.
//   Returns:     nothing
//   Purpose:     this method performs the grant phases of the
//                RGS algorithm.
/*-------------------------------------------------------------*/
void RGSProtocol::runRGSGrant( void )
{
  int j = 0;
  long seed = 0;

  for( j=0; j < nr_ports; j++ )
  {
    if( !OCU[j]->IsOCUEmpty()  &&  !OCU[j]->IsORRBusy() )
    {
      OCU[j]->runGrant( &ICU[0] );
    }
  }
}

/*-------------------------------------------------------------*/
//   Name:        RunRGSStatus
//   Parameters:  void
//   Returns:     nothing
//   Purpose:     this method performs the status phases of the
//                RGS algorithm.
/*-------------------------------------------------------------*/
void RGSProtocol::runRGSStatus( void )
```

```
{
  int j = 0;

  for( j=0; j < nr_ports; j++ )
  {
    if( !OCU[j]->IsOCUEmpty()  &&  OCU[j]->IsORRBusy() )
    {
      OCU[j]->runStatus(j, nr_ports, &ICU[0]);
    }
  }
}

/*-----------------------------------------------------------------*/
//   Name:        HasAlgorithmConverged
//   Parameters:  void
//   Returns:     TRUE or FALSE
//   Purpose:     this method checks to see if algorithm has
//                converged; i.e., all icu's requests were
//                granted..
/*-----------------------------------------------------------------*/
int RGSProtocol::HasAlgorithmConverged( void )
{
  int j = 0;
  int converge = TRUE;

  for( j=0; j<nr_ports; j++ )
  {
    // it takes just one not matched ICU to determine if the
    // system has NOT converged
    if( !ICU[j]->IsIRRBusy() )
    {
      converge = FALSE;
      break;
    }
  }
  return( converge );
}

/*-----------------------------------------------------------------*/
//   Name:        RouteCellsTransState
//   Parameters:  InputQueue* Queue[], StatsProbe* Stats
//   Returns:     void
//   Purpose:     this method routes cells thru the switch, by
//                removing cell from queue, and updating the
//                the number of cells routed in the Stats Probe.
```

```
//                During transient state no cell delay measurements
//                are taken.
/*-------------------------------------------------------------*/
void RGSProtocol::RouteCellsTransState(InputQueue* Queue[],
                                       StatisticalProbe* Stats )
{
  int j = 0;
  int cell_location = 0;

  for( j = 0; j < nr_ports; j++ )
  {
    if( ICU[j]->IsIRRBusy() )
    {
      cell_location = ICU[j]->GetSelCellPosition();
      Queue[j]->RemoveCell( cell_location );
      Queue[j]->DecrementNrCellsStored();

      Stats->UpdateTotalNrCellsRouted();
    }
  }
}

/*-------------------------------------------------------------*/
//   Name:        RouteCellsSteadyState
//   Parameters:  InputQueue* Queue[], StatsProbe* Stats
//   Returns:     void
//   Purpose:     this method routes cells thru the switch, by
//                removing cell from queue, and updating the
//                the number of cells routed in the Stats Probe.
//                During steady state cell delay measurements
//                are taken.
/*-------------------------------------------------------------*/
void RGSProtocol::RouteCellsSteadyState(InputQueue* Queue[],
                                        StatisticalProbe* Stats,
                                        long cur_time )
{
  int j = 0;
  int cell_location = 0;
  long cell_delay = 0;

  for( j = 0; j < nr_ports; j++ )
  {
    if( ICU[j]->IsIRRBusy() )
    {
      cell_location = ICU[j]->GetSelCellPosition();
```

```
      cell_delay = Queue[j]->CellElapsedTime( cell_location, cur_time );

      Stats->UpdateSteadyStateCellDelay( cell_delay );

      Stats->UpdateTotalNrCellsRouted();
      Stats->UpdateSteadyStateNrCellsRouted();

      Queue[j]->RemoveCell( cell_location );
      Queue[j]->DecrementNrCellsStored();
    }
  }
}


/*-------------------------------------------------------------*/
//   Name:        ResetIcuOcu
//   Parameters:  void
//   Returns:     void
//   Purpose:     this method resets the states of ICUs and OCUs.
/*-------------------------------------------------------------*/
void RGSProtocol::ResetIcuOcu( void )
{
  int j = 0;

  for ( j = 0; j < nr_ports; j++ )
  {
    ICU[j]->Reset();
    OCU[j]->Reset();
  }
}

////////////////////////////////////////////////////////////////////
//                                                                //
//    Filename:         timer.h                                   //
//                                                                //
//    Author:           Rubens S. Gomes                           //
//    Date:             October, 1996.                            //
//    Site:             Electrical Engineering & Computer Science //
//                      The University of Kansas                  //
//                      Lawrence, KS, 66045                       //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows95         //
//    Language:         C++                                       //
//    Node Location:    noehter.eecs.ukans.edu                    //
//    Path:             ~rgomes/MS_Proj/Simulation/timer.h        //
```

```
//                                                                      //
//    Purpose:          Header file for the Timer object.  This       //
//                      file defines the data members contained in    //
//                      the object.                                   //
//                                                                    //
//                                                                    //
//    Limitations:      None.                                         //
//                                                                    //
//                                                                    //
//////////////////////////////////////////////////////////////////////////


#if !defined _TIMER_HPP
#define      _TIMER_HPP

class SimulationParameter;
class Simulation;

class Timer
{
  public:
    Timer( SimulationParameter* Parm,
           Simulation* Simu );
    ~Timer( void );
    long GetCurrentSimulationTime( void );
    void NewCellSlot( void );
    int  SimulationTimeDone( void );

  private:
    long CurrentSimulationTime;
    long TransientSimulationTime;
    long TotalSimulationTime;
    Simulation* Simu;
    void SetSimulationTimes( SimulationParameter* Parm );
    void UpdateCurrentSimulationTime( void );
    void CellSlotBegin( void );
};

#endif
//
// File:  timer.C
//

#include "timer.h"
#include "atm_par.h"
```

```
#include "simu.h"


/*-------------------------------------------------------------*/
//   Name:        Timer
//   Parameters:  SimulationParameter* Parm
//   Returns:     nothing
//   Purpose:     constructor method for the Timer
/*-------------------------------------------------------------*/
Timer::Timer( SimulationParameter* Parm,
              Simulation* Simulator )
{
  CurrentSimulationTime = 0;
  TransientSimulationTime = 0;
  TotalSimulationTime = 0;
  Simu = Simulator;
  SetSimulationTimes( Parm );
}

/*-------------------------------------------------------------*/
//   Name:        Timer
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     destructor method for the Timer
/*-------------------------------------------------------------*/
Timer::~Timer( void )
{
}


/*-------------------------------------------------------------*/
//   Name:        NewCellSlot
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     this method is sent by an outside object to
//                tell the Timer that it is time for the next cell
//                slot.  The timer updates the simulation time, and
//                then sends a time signal to the Simulation object
//                if it is not the end of the simulation time.
/*-------------------------------------------------------------*/
void Timer::NewCellSlot( void )
{
  CellSlotBegin();
  UpdateCurrentSimulationTime();
}
```

```
/*-------------------------------------------------------------*/
//   Name:        CellSlotBegin
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     sends a signal to simulation object to start
//                a new ATM cell slot cycle
/*-------------------------------------------------------------*/
void Timer::CellSlotBegin( void )
{
  Simu->CellSlotClock( CurrentSimulationTime );
}


/*-------------------------------------------------------------*/
//   Name:        SetSimulationTimes
//   Parameters:  SimulationParameter* Parm
//   Returns:     nothing
//   Purpose:     sets the transient + total times to run simulation
/*-------------------------------------------------------------*/
void Timer::SetSimulationTimes( SimulationParameter* Parm )
{
  TransientSimulationTime = Parm->GetTransientTime();
  TotalSimulationTime = Parm->GetTotalTime();
}

/*-------------------------------------------------------------*/
//   Name:        UpdateCurrentSimulationTime
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     increments the current simulation time by
//                1(one) slot. The total simulation time is
//                given in number of ATM cell slots. If the
//                current time is greater or equal to Total
//                Simulation Time, a done message is sent to
//                the simulation object.
/*-------------------------------------------------------------*/
void Timer::UpdateCurrentSimulationTime( void )
{
  CurrentSimulationTime++;
}

/*-------------------------------------------------------------*/
//   Name:        GetCurrentSimulationTime
```

```
//   Parameters:  none
//   Returns:     current simulation time (long)
//   Purpose:     returns current simulation time
/*-------------------------------------------------------------*/
long Timer::GetCurrentSimulationTime( void )
{
  return( CurrentSimulationTime );
}

/*-------------------------------------------------------------*/
//   Name:        SimulationTimeDone
//   Parameters:  none
//   Returns:     TRUE or FALSE
//   Purpose:     returns TRUE if the simulation time is done.
/*-------------------------------------------------------------*/
int Timer::SimulationTimeDone( void )
{
  if( CurrentSimulationTime >= TotalSimulationTime )
  {
    return( TRUE );
  }
  else
  {
    return( FALSE );
  }

}
////////////////////////////////////////////////////////////////////
//                                                                //
//    Filename:         stats.h                                   //
//                                                                //
//    Author:           Rubens S. Gomes                           //
//    Date:             August, 1996.                             //
//    Site:             Electrical Engineering & Computer Science //
//                      The University of Kansas                  //
//                      Lawrence, KS, 66045                       //
//    Operating System: UNIX SunOS                                //
//    Language:         C++                                       //
//    Node Location:    noehter.eecs.ukans.edu                    //
//    Path:             ~rgomes/MS_Proj/Simulation/stats.h        //
//                                                                //
//    Purpose:          Header file for the Statistical Probe     //
//                      object.  This file defines the data member//
//                      and operations performed by the object.   //
//                                                                //
```

```
//                                                                      //
//    Limitations:      None.                                           //
//                                                                      //
//                                                                      //
//////////////////////////////////////////////////////////////////////////


#if !defined _STATS_HPP
#define      _STATS_HPP

const double CELL_TIME_SLOT       = 2.74e-6;  // 2.74 useconds


class StatisticalProbe
{
  public:
    StatisticalProbe( void );

    void   UpdateTotalNrCellSlots( void );
    void   UpdateSteadyStateNrCellSlots( void );

    void   UpdateTotalNrCellsRouted( void );
    void   UpdateSteadyStateNrCellsRouted( void );

    void   UpdateTotalNrCellsArrived( void );
    void   UpdateSteadyStateNrCellsArrived( void );

    void   UpdateSteadyStateCellDelay( long cell_delay );

    double QueueAvgCellDelay( void );
    double OutputPortAvgUtilization( int switch_size );
    double SwitchThroughput( void );

    long   GetTotalNrCellsRouted( void );
    long   GetSteadyStateNrCellsRouted( void );

    long   GetTotalNrCellsArrived( void );
    long   GetSteadyStateNrCellsArrived( void );

    long   GetTotalNrCellSlots( void );
    long   GetSteadyStateNrCellSlots( void );

  private:
    long   TotalNrCellsRouted;
    long   SteadyStateNrCellsRouted;
```

```
    long   TotalNrCellSlots;
    long   SteadyStateNrCellSlots;

    long   TotalNrCellsArrived;
    long   SteadyStateNrCellsArrived;

    double TotalOfSteadyStateCellDelays;
};

#endif
//
//  File:      stats.C
//

#include "stats.h"

/*-------------------------------------------------------------*/
//   Name:        StatisticalProbe
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     constructor function for the StatisticalProbe
/*-------------------------------------------------------------*/
StatisticalProbe::StatisticalProbe( void )
{
  TotalNrCellsRouted        = 0;
  SteadyStateNrCellsRouted  = 0;

  TotalNrCellSlots          = 0;
  SteadyStateNrCellSlots    = 0;

  TotalNrCellsArrived       = 0;
  SteadyStateNrCellsArrived = 0;

  TotalOfSteadyStateCellDelays   = 0;
}

/*-------------------------------------------------------------*/
//   Name:        UpdateTotalNrCellsRouted
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This member function is called every time a
//                cell is routed through the switch.  It
//                increments the number of cells routed through
//                the ATM switch for both transient and steady
```

```
//                states.
/*-------------------------------------------------------------*/
void StatisticalProbe::UpdateTotalNrCellsRouted( void )
{
  TotalNrCellsRouted++;
}

/*-------------------------------------------------------------*/
//   Name:        UpdateSteadyStateNrCellsRouted
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This member function is called every time a
//                cell is routed through the switch.  It
//                increments the number of cells routed through
//                the ATM switch for steady state only.
/*-------------------------------------------------------------*/
void StatisticalProbe::UpdateSteadyStateNrCellsRouted( void )
{
  SteadyStateNrCellsRouted++;
}


/*-------------------------------------------------------------*/
//   Name:        UpdateTotalNrCellsArrived
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This member function is called every time a
//                cell arrived in the switch.  It increments the
//                number of cells that arrive in the ATM switch
//                during thansient and steady states.
/*-------------------------------------------------------------*/
void StatisticalProbe::UpdateTotalNrCellsArrived( void )
{
  TotalNrCellsArrived++;
}

/*-------------------------------------------------------------*/
//   Name:        UpdateSteadyStateNrCellsArrived
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     This member function is called every time a
//                cell arrived in the switch.  It increments the
//                number of cells that arrive in the ATM switch
//                during steady state.
/*-------------------------------------------------------------*/
```

```
void StatisticalProbe::UpdateSteadyStateNrCellsArrived( void )
{
  SteadyStateNrCellsArrived++;
}

/*-------------------------------------------------------------*/
//   Name:        UpdateSteadyStateCellDelay
//   Parameters:  cell_delay  (input, long) cell delay
//   Returns:     nothing
//   Purpose:     This function keeps track of the cell time
//                delays, which will be used to calculate the
//                switch average cell delay.  The cell delay is
//                based on the number of cell slots elapsed.
//                This data is only updated during steady state.
/*-------------------------------------------------------------*/
void StatisticalProbe::UpdateSteadyStateCellDelay( long cell_delay )
{
  TotalOfSteadyStateCellDelays = TotalOfSteadyStateCellDelays +
                                 cell_delay;
}

/*-------------------------------------------------------------*/
//   Name:        UpdateTotalNrCellSlots
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     This function is called everytime a new cell
//                time slot is created.  It keeps track of the
//                number of the total number of cell slots used
//                by the switch during transient and steady states.
/*-------------------------------------------------------------*/
void StatisticalProbe::UpdateTotalNrCellSlots( void )
{
  TotalNrCellSlots++;
}

/*-------------------------------------------------------------*/
//   Name:        UpdateSteadyStateNrCellSlots
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     This function is called everytime a new cell
//                time slot is created.  It keeps track of the
//                number of the number of cell slots used
//                by the switch during the steady states
/*-------------------------------------------------------------*/
void StatisticalProbe::UpdateSteadyStateNrCellSlots( void )
```

```
{
  SteadyStateNrCellSlots++;
}

/*-------------------------------------------------------------*/
//   Name:        QueueAvgCellDelay
//   Parameters:  none
//   Returns:     Queueing Average Cell delay in the ATM switch
//   Purpose:     This method is called at the very end of
//                the simulation to calculate the average queueing
//                cell delay in the switch.  It determines
//                the average by dividing (the total Nr of cell
//                slots delay times the time per slot) by the number
//                of cells routed in the switch.  This method
//                calculates the average cell delay for a simulation
//                run during the steady state.
/*-------------------------------------------------------------*/
double StatisticalProbe::QueueAvgCellDelay( void )
{
  double Delay = 0.0;

  Delay = ( TotalOfSteadyStateCellDelays * CELL_TIME_SLOT )
         / (double) SteadyStateNrCellsRouted;
  return( Delay );
}


/*-------------------------------------------------------------*/
//   Name:        OutputPortAvgUtilization
//   Parameters:  switch_size (int)
//   Returns:     Average Utilization of output ports
//   Purpose:     This method calculates the utilization of
//                the output ports in the switch, which I define
//                to be the % of cell slots (time) that the
//                output ports are busy.  The % of time that
//                the output ports are busy can also be understood
//                to be the number of cells routed divided by
//                the number of cell slots.  The switch utilization
//                only takes into account the cell slots that
//                generated after the transient state.
/*-------------------------------------------------------------*/
double StatisticalProbe::OutputPortAvgUtilization( int switch_size )
{
  double local_utilization = 0.0;
```

```
    local_utilization = (double) SteadyStateNrCellsRouted /
                                 ((double)  SteadyStateNrCellSlots  *  (double)
switch_size);
    local_utilization = local_utilization  *  100.0;
     return( local_utilization );
  }

  /*-------------------------------------------------------------*/
  //   Name:        SwitchThroughput
  //   Parameters:  void
  //   Returns:     Average Switch Throughput
  //   Purpose:     This method calculates the switch throughput,
  //                which I define to be the Total_Nr_of_Cells_Routed
  //                / Total_Nr_of_Cells_Arrived.  Transient state
  //                is not being taken into account, which means that
  //                it is assumed that this result is valid only if
  //                the simulation runs for a large number of cells
  //                slots (50,000 or greater)
  /*-------------------------------------------------------------*/
  double StatisticalProbe::SwitchThroughput( void )
  {
    double thruput = 0.0;

    thruput = (double) TotalNrCellsRouted/
              (double) TotalNrCellsArrived;
    thruput = thruput *  100.0;
    return( thruput );
  }


  /*-------------------------------------------------------------*/
  //   Name:        GetTotalNrCellsRouted
  //   Parameters:  none
  //   Returns:     Total Nr of cells routed in the switch
  //   Purpose:     This function returns the total number of
  //                ATM cells routed in the switch up to the
  //                point this function is called.  This means
  //                that this number includes both transient and
  //                steady states.
  /*-------------------------------------------------------------*/
  long StatisticalProbe::GetTotalNrCellsRouted( void )
  {
    return( TotalNrCellsRouted );
  }
```

```
/*-------------------------------------------------------------*/
//   Name:        GetSteadyStateNrCellsRouted
//   Parameters:  none
//   Returns:     Total Nr of cells routed in the switch
//   Purpose:     This function returns the total number of
//                ATM cells routed in the switch up to the
//                point this function is called.  This means
//                that this number includes steady state only.
/*-------------------------------------------------------------*/
long StatisticalProbe::GetSteadyStateNrCellsRouted( void )
{
  return( SteadyStateNrCellsRouted );
}

/*-------------------------------------------------------------*/
//   Name:        GetTotalNrCellsArrived
//   Parameters:  none
//   Returns:     Total Nr of cells that arrived in the switch
//   Purpose:     This function returns the total number of
//                ATM cells that arrived in the switch up to the
//                point this function is called.
//                This means that this number includes both transient
//                and steady states.
/*-------------------------------------------------------------*/
long StatisticalProbe::GetTotalNrCellsArrived( void )
{
  return( TotalNrCellsArrived );
}

/*-------------------------------------------------------------*/
//   Name:        GetSteadyStateNrCellsArrived
//   Parameters:  none
//   Returns:     Total Nr of cells that arrived in the switch
//   Purpose:     This function returns the total number of
//                ATM cells that arrived in the switch up to the
//                point this function is called.
//                This means that this number includes steady
//                state only.
/*-------------------------------------------------------------*/
long StatisticalProbe::GetSteadyStateNrCellsArrived( void )
{
  return( SteadyStateNrCellsArrived );
}

/*-------------------------------------------------------------*/
```

```
//   Name:        GetTotalNrCellSlots
//   Parameters:  none
//   Returns:     Total Nr of cell slots that have elapsed in
//                the ATM switch
//   Purpose:     This function returns the total number of
//                ATM cell slots that elapsed in the switch up to the
//                point this function is called.
//                This means that this number includes both transient
//                and steady states.
/*-------------------------------------------------------------*/
long StatisticalProbe::GetTotalNrCellSlots( void )
{
  return( TotalNrCellSlots );
}

/*-------------------------------------------------------------*/
//   Name:        GetSteadyStateNrCellSlots
//   Parameters:  none
//   Returns:     Total Nr of cell slots that have elapsed in
//                the ATM switch
//   Purpose:     This function returns the total number of
//                ATM cell slots that elapsed in the switch up to the
//                point this function is called.
//                This means that this number includes steady
//                state only.
/*-------------------------------------------------------------*/
long StatisticalProbe::GetSteadyStateNrCellSlots( void )
{
  return( SteadyStateNrCellSlots );
}
////////////////////////////////////////////////////////////////////
//                                                                //
//    Filename:         icu_unit.h                                //
//                                                                //
//    Author:           Rubens S. Gomes                           //
//    Date:             August, 1996.                             //
//    Site:             Electrical Engineering & Computer Science //
//                      The University of Kansas                  //
//                      Lawrence, KS, 66045                       //
//    Operating System: HP UNIX, DEC OS/F UNIX, Windows95         //
//    Language:         C++                                       //
//    Node Location:    noehter.eecs.ukans.edu                    //
//    Path:             ~rgomes/MS_Proj/Simulation/icu_unit.h     //
//                                                                //
//    Purpose:          Header file for the ICU object.  This file  //
```

```
//                      defines the data members and operations      //
//                      performed by the object.                     //
//                                                                   //
//                                                                   //
//    Limitations:      None.                                        //
//                                                                   //
//                                                                   //
///////////////////////////////////////////////////////////////////////

#include "queue.h"


#if !defined  _ICU_HPP
#define       _ICU_HPP


typedef enum
{
  IRR_FREE,
  IRR_BUSY
}IRR_STATES;

#define  SIR_BUSY   (  1 )
#define  SIR_FREE   (  0 )
#define  MAX_PORTS  ( 65 )


class OutputControlUnit;

class InputControlUnit
{
  public:
    InputControlUnit( void );
     int  IsIRRBusy( void );
     int  GetSelCellPosition( void );
     void Grant( void );
     void Reset( void );
    void ResetCurCellPosition( void );
    void StopFreeOutputSearch( void );
    void ResetFreeOutputSearch( void );
    void runRequest( int icu_nr,
                     InputQueue* Queue,
                     OutputControlUnit* OCU[] );
    void Status( int ocu_nr );
```

```
   private:
      void StoreSelectedCellPosition( int position );
     int  ContinueSearch( void );
     int  GetCurCellPosition( void );
     void IncrementCurCellPosition( void );
      IRR_STATES  icu_register;   // status of ICU register
      int  selected_cell_position;
     int  cur_cell_position;
     int  ContinueFreeOutputSearch;
     char SIR[MAX_PORTS];
     int  IsSIRFree( int ocu_nr );

};

#endif
//
//  File:      icu_unit.C
//

#include "icu_unit.h"
#include "ocu_unit.h"

/*-----------------------------------------------------------------*/
//   Name:        InputControlUnit
//   Parameters:  None
//   Returns:     nothing
//   Purpose:     constructor function for the  ICU object.
/*-----------------------------------------------------------------*/
InputControlUnit::InputControlUnit( void )
{
   icu_register = IRR_FREE;
  //
  // the minimum value allowed for cur_cell_position is '1'
  // because it is used as an input parameter for the InputQueue
  // object which uses the value of '1' for the head of the queue.
  //
   cur_cell_position = 1;
   selected_cell_position = 0;
  ContinueFreeOutputSearch = TRUE;
  memset( &SIR[0], '\0', sizeof SIR );
}


/*-----------------------------------------------------------------*/
//   Name:        IsIRRBusy
```

```
//   Parameters:  None
//   Returns:     TRUE or FALSE (int)
//   Purpose:     Returns the status of the Input Control Register.
//                If IRR is busy then this input port has already
//                been matched with an output port.  Otherwise,
//                the port queue is either empty, and/or has not
//                received a grant yet.
/*-------------------------------------------------------------*/
int InputControlUnit::IsIRRBusy( void )
{
  if( icu_register == IRR_BUSY )
  {
    return( TRUE );
  }
  else
  {
    return( FALSE );
  }
}

/*-------------------------------------------------------------*/
//   Name:        StoreSelectedCellPosition
//   Parameters:  position  (input, long)  - position in the queue
//   Returns:     nothing
//   Purpose:     This function stores the position in the queue
//                which points to the cell whose request has been
//                granted.  This cell will be routed in the next
//                routing cycle.
/*-------------------------------------------------------------*/
void InputControlUnit::StoreSelectedCellPosition( int position )
{
  selected_cell_position = position;
}

/*-------------------------------------------------------------*/
//   Name:        GetCurCellPosition
//   Parameters:  none
//   Returns:     current cell position (int).
//   Purpose:     This function returns the position for the
//                cell that should send the request at the
//                current iteration.
/*-------------------------------------------------------------*/
int InputControlUnit::GetCurCellPosition( void )
{
   return( cur_cell_position );
```

```
}

/*-------------------------------------------------------------*/
//   Name:        IncrementCurCellPosition
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     This function moves the position pointer to
//                next cell node in the queue.
/*-------------------------------------------------------------*/
void InputControlUnit::IncrementCurCellPosition( void )
{
   cur_cell_position++;
}

/*-------------------------------------------------------------*/
//   Name:        GetSelCellPosition
//   Parameters:  none
//   Returns:     selected cell position.
//   Purpose:     This function returns the position for the
//                cell whose request has been granted.
/*-------------------------------------------------------------*/
int InputControlUnit::GetSelCellPosition( void )
{
   return( selected_cell_position );
}


/*-------------------------------------------------------------*/
//   Name:        Grant
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     When a cell request has been granted by an OCU
//                the ICU must set the IRR to busy, and store
//                the current cell location position.
/*-------------------------------------------------------------*/
void InputControlUnit::Grant( void )
{
  icu_register = IRR_BUSY;
   selected_cell_position = cur_cell_position;
}

/*-------------------------------------------------------------*/
//   Name:        Reset
//   Parameters:  none
//   Returns:     nothing
```

```
//   Purpose:     This function restores the initial configuration
//                of the ICU unit.  This function should be
//                called after the cells are routed through
//                the switch.
/*-------------------------------------------------------------*/
void InputControlUnit::Reset( void )
{
   icu_register = IRR_FREE;
   cur_cell_position = 1;
   selected_cell_position = 0;
  ContinueFreeOutputSearch = TRUE;
  memset( &SIR[0], '\0', sizeof SIR );
}


/*-------------------------------------------------------------*/
//   Name:        ResetCurCellPosition
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     This function resets the position pointer to
//                the head of the queue.
/*-------------------------------------------------------------*/
void InputControlUnit::ResetCurCellPosition( void )
{
   cur_cell_position = 1;
}


/*-------------------------------------------------------------*/
//   Name:        StopFreeOutputSearch
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     This function sets the ContinueFreeOutputSearch
//                variable to FALSE(false) in order to stop the ICU
//                from doing a search for free output in the current
//                cell slot.  This variable is set when the ICU has
//                checked for all the cells in the queue, and has not
//                found a free output..
/*-------------------------------------------------------------*/
void InputControlUnit::StopFreeOutputSearch( void )
{
   ContinueFreeOutputSearch = FALSE;
}

/*-------------------------------------------------------------*/
```

```
//   Name:        ResetContinueFreeOutputSearch
//   Parameters:  none
//   Returns:     nothing
//   Purpose:     This function resets the ContinueFreeOutputSearch
//                variable to TRUE (true) so that in the next cell
//                slot, the ICU can start traversing the queue again
//                in search of a free output in the current
//                cell slot.
/*-------------------------------------------------------------*/
void InputControlUnit::ResetFreeOutputSearch( void )
{
   ContinueFreeOutputSearch = TRUE;
}

/*-------------------------------------------------------------*/
//   Name:        ContinueSearch
//   Parameters:  none
//   Returns:     ContinueFreeOutputSearch ( int)
//   Purpose:     This function returns the current value of
//                the ContinueFreeOutputSearch
/*-------------------------------------------------------------*/
int InputControlUnit::ContinueSearch( void )
{
   return( ContinueFreeOutputSearch );
}

/*-------------------------------------------------------------*/
//   Name:        runRequest
//   Parameters:  icu_nr (int)
//                Queue  (InputQueue*)
//                OCU    (OutputControlUnit*)
//   Returns:     void
//   Purpose:     This function runs the request phase in the ICU
/*-------------------------------------------------------------*/
void InputControlUnit::runRequest(int icu_nr,
                                  InputQueue* Queue,
                                  OutputControlUnit* OCU[] )
{
  int out_port = 0;
  char RequestDone = FALSE;

  while(   Queue->IsThereCellAt( GetCurCellPosition() )
        && ContinueSearch()
        && !RequestDone )
  {
```

```
    out_port = Queue->CellOutputPort( GetCurCellPosition() );

    // search for free output...
    if( IsSIRFree( out_port ) )
    {
      OCU[out_port]->Request( icu_nr );
      RequestDone = TRUE;
    }
    else
    {
      IncrementCurCellPosition();
    }
  }
}

/*-------------------------------------------------------------*/
//   Name:        Status
//   Parameters:  ocu_nr (int)
//   Returns:     void
//   Purpose:     stores the status of the OCU in the SIR.
/*-------------------------------------------------------------*/
void InputControlUnit::Status( int ocu_nr )
{
  SIR[ocu_nr] = SIR_BUSY;
}


/*-------------------------------------------------------------*/
//   Name:        IsSIRFree
//   Parameters:  ocu_nr (int)
//   Returns:     TRUE or FALSE
//   Purpose:     checks to see if the SIR register for the OCU
//                specified in ocu_nr is free.
/*-------------------------------------------------------------*/
int InputControlUnit::IsSIRFree( int ocu_nr )
{
  if( SIR[ocu_nr] == '\0' )
  {
    return( TRUE );
  }
  else
  {
    return( FALSE );
  }
}
```

```
////////////////////////////////////////////////////////////////////////
//                                                                    //
//    Filename:         ocu_unit.h                                    //
//                                                                    //
//    Author:           Rubens S. Gomes                               //
//    Date:             August, 1996.                                 //
//    Site:             Electrical Engineering & Computer Science     //
//                      The University of Kansas                      //
//                      Lawrence, KS, 66045                           //
//    Operating System: UNIX SunOS                                    //
//    Language:         C++                                           //
//    Node Location:    noehter.eecs.ukans.edu                        //
//    Path:             ~rgomes/MS_Proj/Simulation/ocu_unit.h         //
//                                                                    //
//    Purpose:          Header file for the OCU object.  This file    //
//                      defines the data member and operations        //
//                      performed by the object.                      //
//                                                                    //
//                                                                    //
//    Limitations:      None.                                         //
//                                                                    //
//                                                                    //
////////////////////////////////////////////////////////////////////////

#include "rand.h"

#if !defined _OCU_HPP
#define      _OCU_HPP


#define TRUE      (  1  )
#define FALSE     (  0  )
#define MAX_PORTS ( 65  )

typedef enum
{
  FREE,
  BUSY
}ORR_STATES;

#define  EMPTY               (    46 )  // ascii code for "."


class InputControlUnit;
```

```
class OutputControlUnit
{
  public:
    OutputControlUnit( int mult, long seed );
    ~OutputControlUnit( void );

    void Request( int ICU_nr );
     void Reset( void );
    int  IsORRBusy( void );
    int  IsOCUEmpty( void );
    int  GrantSignal( int ICU_nr );
    void runGrant( InputControlUnit* ICU[]);
    void runStatus( int ocu_nr,
                    int nr_ports,
                    InputControlUnit* ICU[]);

  private:
    int  SelectRequest( void );
    RandNrGenerator* RandNrGen;
    int              requests[MAX_PORTS+1];
    int              selected_request;
    ORR_STATES       ocu_register;
};

#endif
//
//  File:      ocu_unit.C
//

#include "ocu_unit.h"
#include "icu_unit.h"


/*-------------------------------------------------------------*/
//   Name:        OutputControlUnit
//   Parameters:  int mult, long seed.
//   Returns:     nothing
//   Purpose:     constructor function for the OCU object
/*-------------------------------------------------------------*/
OutputControlUnit::OutputControlUnit( int mult, long seed )
{
  int i;

  ocu_register = FREE;
```

```
  selected_request = EMPTY;
  for( i = 0; i < (MAX_PORTS + 1); i++ )
  {
    requests[i] = EMPTY;
  }
  // creates a random nr generEyreK4II5IIIEyIEy);K4HH5HHHEy K4II5IIIEy K4II5IIIE/j4II5IIIEygK4
 _ 1 ge□y K4II/NIdNNX_yeK4II5IIpII5IIIEy_K4IryeK4II5IIC  NNWrEyrF K4II5IIIEy K4II5IIIN/NINXXNN
c_y□{K□{:|LpII5IIIEy_K4IryeK4II5IIC  XrEyrF K4II5IIIEy K4II5IIIN/NINXXNNWc//5//WI5IIIEy1K4II5v
PI□NW  NNWNNWrEyrF K4II5IIIHIXHeNXRNX □{    rEyr□4XX5XXXEyFK4II5IIIEy□y K4II5IIIEy1K4II5IIy□{K
  rEyr:HK4II5IIIEyi K4II5IIIIEN/e
eIIyH K4II5IIIINNWq#RXrEyrmIIEy K4II5IIIEy1K4II5IIXHNNHK4II5IIIEyqII5IIIEy_K4IrK4II5IIIEyeqK4I
    □{
 NX 1P
 eEyNXXK4II5IIIEyoK4II5IIXHNNHq4Irequ□q#RX1N/e
PIIIEN/e□NW_
   □{
                        eq11 1eEyNXXK4II5IIIEyoK4II5IIXHNNHq4Irequ□q#RX1NI5IINE4X+5+E|(4IIXEy
  PII5IIIEy_K++II5IIIEy_K;NI5IIX □{
```

```
/*-------------------------------------------------------------*/
//   Name:        SelectRequest
//   Parameters:  void
//   Returns:     Selected ICU port number request
//   Purpose:     This function selects one among all the requests
//                stored in the requests array.  It also sets the
//                OCU register to TRUE (busy) after a selection
//                is made.
/*-------------------------------------------------------------*/
int OutputControlUnit::SelectRequest( void )
{
  int count;      // number of requests stored
  int position;   // position in requests array that holds selected request

  // determine number of requests stored.
  count = 0;
  while( requests[count] != EMPTY )
  {
    count++;
  }

  // uniformily random select one of the requests.
  position = RandNrGen->RandDiscUnifValue( 0, (count - 1) );
  selected_request = requests[position];
```

```
  {
    requests[i] = EMPTY;
  }
}

/*-------------------------------------------------------------*/
//   Name:        IsORRBusy
//   Parameters:  none
//   Returns:     TRUE or FALSE
//   Purpose:     This function returns "TRUE" if the OCU
//                register is busy; otherwise, it returns FALSE
/*-------------------------------------------------------------*/
int OutputControlUnit::IsORRBusy( void )
{
```

```
/*-------------------------------------------------------------*/
//   Name:        GrantSignal
//   Parameters:  ICU number (input, int)
//   Returns:     TRUE or FALSE
//   Purpose:     This function returns TRUE if the ICU number
//                passed in matches the selected_request.  In
//                other words, this function responds with either
//                a GRANT or DENIED response to the requesting
//                ICU unit.
/*-------------------------------------------------------------*/
int OutputControlUnit::GrantSignal( int ICU_nr )
{
  if( ICU_nr == selected_request )
  {
    return( TRUE );
  }
  else
  {
    return( FALSE );
  }
}


/*-------------------------------------------------------------*/
//   Name:        runGrant
//   Parameters:  ICU  (InputControlUnit*)
//   Returns:     none
//   Purpose:     This function runs the grant phase in the OCU.
/*-------------------------------------------------------------*/
void OutputControlUnit::runGrant( InputControlUnit* ICU[] )
{
  int icu_selected = 0;

  icu_selected = SelectRequest();
  ICU[icu_selected]->Grant();
}

/*-------------------------------------------------------------*/
//   Name:        runStatus
//   Parameters:  ocu_nr   (int)
//                nr_ports (int)
//                ICU      (InputControlUnit*)
//   Returns:     none
//   Purpose:     This function runs the status phase in the OCU.
/*-------------------------------------------------------------*/
```

```
void OutputControlUnit::runStatus(int ocu_nr,
                                  int nr_ports,
                                  InputControlUnit* ICU[] )
{
  int i = 0;

  for( i; i < nr_ports; i++)
  {
    ICU[i]->Status( ocu_nr );
  }
}
```

[1] S. Motoyama, D. W. Petr, V. S. Frost, "Input-queued switch based on a scheduling algorithm," *Electronics Letters*, vol. 31, No. 14, pp. 1127-1128, July 1995.

[2] Y. S. Yeh, M. G. Hluchyj, and A. S. Acampora, "The Knockout switch: A simple, modular architecture for high-performance packet switching," *IEEE J. Select. Areas Commun.*, vol. SAC-5, pp. 1274-1283, Oct. 1987.

[3] J. Karol, M. G. Hluchyj, and S. P. Morgan, "Input vs. output queueing on a space-division packet switch," *IEEE Trans.* Commun., vol. COM-35, pp. 1347-1356, Dec. 1987.

[4] T. E. Anderson, S. S. Owicki, J. B. Saxe, and C. P. Thacker, "High Speed Switch Scheduling for Local Area Networks," *Proc. Fifth International Conference on Architecture Support for Programming Languages and Operating Systems*, pp. 98-110, Oct. 1992.

[5] P. A. Fishwick, *Simulation Model Design and Execution*, Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[6] I. M. Jacobson, P. Johnson, and G. Overgaard, *Object-Oriented Software Engineering - A Use Case Driven Approach*, Addison-Wesley, Reading, MA, 1992.

[7] D. Coleman, P. Arnold, S. Bodoff, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremaes, *Object-Oriented Development: The Fusion Method*, Prentice Hall, Englewood Cliffs, NJ, 1994.

[8] J. Karol, and M. G. Hluchyj , "Queueing in High-Performance Packet Switching," *IEEE Journal on Selected Areas in Communications*, vol. 6, No. 9, pp. 1587-1597, Dec. 1988.

[9] T. G. Robertazzi, *Computer Networks and Systems - Queueing Theory and Performance Evaluation*, 2nd. edition, Springer-Verlag, New York, NY, 1994.

[10] C. P. Thacker, M. D. Schroeder, *AN2 Switch Overview*, Digital Equipment Corporation, Systems Research Center, Palo Alto, CA, July, 1994.

[11]F. M. Carrano, *Data Abstraction and Problem Solving with C++*, the Benjamin/Cummings Publishing Company, Inc., Redwood City, CA, 1995.

[12]Fisher, J.A., "*Object Oriented Simulation tools for Discrete-Continuos, Stochastic-Deterministic Simulation Models*,", Master Thesis, Oregon State Univerisy, Covalli, OR, 1992.