MASTER

SHE as simulation tool for packet switch fabrics

van Hoof, S.A.H.

Award date:
1997

Link to publication
SHE AS SIMULATION TOOL
FOR PACKET SWITCH FABRICS

By S.A.H. van Hoof.

Author: S.A.H van Hoof (idnr.: 405343).
Supervisor: Ir. M.J.M. van Weert
Professor: Prof. Ir. M.P.J Stevens.
Date: 9 June 1997.
Abstract

Today's and future services ask for a broadband network. This network must be flexible and fast. ATM (Asynchronous Transfer Mode) is such a network. ATM information is transported by fixed sized packets called cells. Packet switches route these cells from source to destination. The transport of cells from packet switch input to packet switch output is called switching. Multiple switches together form a packet switch fabric.

Simulations can be done in order to have an indication of the performance of such a packet switch fabric. These simulations must be done by means of a computer. This asks for the principle of modeling. There are many ways to create a model. This thesis describes the method SHE used for modeling packet switch fabrics. The SHE method is a new Object Oriented specification method developed at the Technical University of Eindhoven. SHE supports formal and informal modeling. Non-formal modeling is necessary for a good understanding of the problem and for a smooth path towards a formal model. A formal model in SHE consists of a behavior description in the language POOSL. The SHE method comes with a special tool which is able to simulate the POOSL behavior descriptions of the packet switch fabrics. A 2 input 2 output (2m2) packet switch and a 4 input 4 output (4m4) packet switch fabric are modeled according to the SHE method and fed to the SHE simulation tool. During packet switch (fabric) simulations the cell delay and the switch buffer length are measured. These simulation results are displayed in graphics. Further, the performance of the simulation tool itself is tested.

The conclusion is that the SHE method is a very nice and adequate instrument for modeling packet switch fabrics but at this moment the SHE simulation tool is found not sufficient to do simulations for large packet switch fabric models.
## Contents

1. Introduction ........................................................................................................................................ 1

2. The SHE method ............................................................................................................................... 2
   2.1 Introduction................................................................................................................................. 2
   2.2 Conceptual solution .................................................................................................................. 2
   2.3 Graphical and textual modeling ............................................................................................... 2
     2.3.1 Non formal models ........................................................................................................... 3
     2.3.2 Formal model ................................................................................................................... 4
   2.4 Classes and objects ................................................................................................................... 4
   2.5 Process objects ......................................................................................................................... 5
   2.6 Data objects ............................................................................................................................. 5
   2.7 Summary .................................................................................................................................. 6

3. The conceptual solution .................................................................................................................... 7
   3.1 Introduction............................................................................................................................... 7
   3.2 The abstraction level ............................................................................................................... 7
   3.3 Finding the conceptual solution .............................................................................................. 7
   3.4 Cell generator .......................................................................................................................... 8
   3.5 Switch ...................................................................................................................................... 8
     3.5.1 Introduction......................................................................................................................... 8
     3.5.2 Cell routing ....................................................................................................................... 8
     3.5.3 The modularity concept ................................................................................................... 9
     3.5.4 Queuing in packet switches .............................................................................................. 10
   3.6 Stage addresser ....................................................................................................................... 11
     3.6.1 The need for an addresser .............................................................................................. 11
   3.7 Cell receiver ............................................................................................................................ 11
   3.8 Controller/Inspector ............................................................................................................... 11
     3.8.1 Introduction......................................................................................................................... 11
     3.8.2 The role as controller ....................................................................................................... 11
     3.8.3 The role as inspector ....................................................................................................... 11
     3.8.4 Controlling simulations ................................................................................................. 12
   3.9 Summary .................................................................................................................................. 12

4. Towards SHE models ...................................................................................................................... 13
   4.1 Introduction............................................................................................................................... 13
   4.2 Process objects in the simulator model .................................................................................... 13
   4.3 Data objects in the simulator model ....................................................................................... 14
   4.4 Instance structure diagram ..................................................................................................... 14
   4.5 Message flow diagram ............................................................................................................ 15
   4.6 Summary .................................................................................................................................. 15

5. Performing simulations ................................................................................................................... 16
   5.1 Introduction............................................................................................................................... 16
   5.2 The simulation tool coming with SHE .................................................................................... 16
     5.2.1 Introduction......................................................................................................................... 16
     5.2.2 The library coming with the SHE simulation tool ......................................................... 16
     5.2.3 Normal simulator operation ............................................................................................ 16
   5.3 Understanding the POOSL semantics ...................................................................................... 17
   5.4 Synchronize processes ............................................................................................................. 17
   5.5 Summary .................................................................................................................................. 19
   5.6 Conclusion ............................................................................................................................... 20

6. Conclusions ..................................................................................................................................... 20
1. Introduction

The growing need for a network which is able to deal with a lot of different services like voice, large amounts of data and video traffic leads to the development of a broadband integrated services digital network. Because this new broadband network has to support current and future services it must be flexible and fast.

ATM, which stands for Asynchronous Transfer Mode, is a rather new switching, multiplexing and transmission technique which meets the demands of speed and flexibility.

In ATM information is transported through the network in small fixed length packets called cells. Cells are identified by a header containing identification and network information. This header is needed to identify the transport of the payload, the information part of the cell.

For the transport and routing of cells high speed switches are needed. These switches must be able to route arriving cells at their inlets to the proper outlets at very high speed, typically 150 Mbps and up. Further demands include a low delay, the ability to have different traffic priorities and to combine several single switches to increase overall performance. Multiple interconnected switches form a switch fabric. This switch fabric in fact is one big switch build out of small switching elements.

Because of the stochastic behavior of the environment in which the switches operate it is difficult to determine the 'real-world-performance' just theoretically. The offered traffic for example can have much diversity. Therefore, the need for a simulator of switches and switch fabrics exists. The results of simulations can give interesting information and even reveal design errors.

This graduation report describes the building of a model for packet switch fabrics using the SHE-method. This model is simulated by the SHE simulator tool and some simulation results are given. In Chapter 2 the method SHE is described in short. In Chapter 3 the conceptual solution is given. Chapter 4 describes the SHE models used for modeling packet switch (fabrics). Chapter 5 describes the usage and performance of the model simulator coming with SHE. In Chapter 6 conclusions are given. In the Appendices detailed information, POOSL code and simulation results can be found.
2. The SHE method

2.1 Introduction

Many methods exist to assist the developer in his work, in this case that is: building and simulating a model for packet switch fabrics. Each method has its own strong and weak points. It is therefore important to find the right method, a method which fulfills (most of) the requirements. I started with the method OMT (Object Modeling Technique) by Rumbaugh [2]. After reading his book and discovering OMT I did not have the feeling of dealing with the correct method. Also Dr. Ir. Jeroen Voeten told me about his new specification method which he developed together with Dr. Ing Piet van der Putten called SHE (The method Software/Hardware Engineering). I choose this new method for the building of the model because a simulator tool for the model is available, it supports concurrency and this method is supported by many people on my working floor. My project further acts as another 'test' for the method itself, so I can give feedback to the developers.

This chapter gives a short description of SHE method concepts. A complete in depth explanation of the method SHE is given in [1].

2.2 Conceptual solution

As developer of a (new) system you are confronted with a problem. From scratch there are constraints and requirements in solving this problem. When the developer is thinking about the problem and takes the initial constants and requirements into account he or she creates a possible solution in mind. In SHE this possible solution is called the conceptual solution.

The dividing of a total system into its main parts and assigning each part its main functionality can be interpreted as making a conceptual solution in SHE. This early dividing in (functional) parts also enables the concept of modularity. The conceptual solution acts as basis in building a model. In Chapter 3 the conceptual solution for modeling the packet switch fabric is given. Note that while building a model the conceptual solution might change (e.g. due to a better understanding of the problem). The method SHE supports this.

2.3 Graphical and textual modeling

One of the first things to do when solving a complex 'real world problem' is to model it. A model is an abstract representation of a physical system. In general, when creating models the abstraction level has to be known or chosen. The method SHE delivers several modeling views for a system such as behavior views, architectural views, requirement views. In SHE, the model of a system can be represented in two ways: a formal and a non formal one. This paragraph describes both models. Some examples of these models are given in Chapter 4. In Figure 2-1 some models are displayed. The dividing line in this figure separates the formal and non formal models.
2.3.1 Non formal models

Graphical modeling results in understanding of the system to be designed. During modeling we play scenarios of system behavior in thought. This behavior is modeled by a collection of collaborating process objects (See 2.5). Process objects can perform their behavior concurrently. They communicate solely by exchanging messages over static channels. The process objects and their communication can be represented in graphical models. Graphical models are divided into object models and structure models.

2.3.1.1 Structure models

In SHE two structure models are known: The architecture structure diagrams and the implementation structure diagrams. These diagrams are in general block diagrams that show the physical and/or logical structure of a conceptual solution. Figure 3-1 can be identified as structure model. It foresees in a topology of the system to be designed. The arrows between the parts (or blocks) represent communication channels.

2.3.1.2 Object models

The object models are the object class model and the object instance model. The object class model visualizes the relations between classes and the object instance model visualizes communication between objects. Communication is shown in two ways. Message flow diagrams show the messages between objects and instance structure diagrams show objects and the static channels -carrying the messages- between them. The goal of message flow diagrams is to help reasoning about system behavior. SHE offers a collection of so-called message flow symbols, used in message flow diagrams. In Figure 2-2 the available message flow symbols are presented.
2.3.2 Formal model

The formal part of the total system model is determined and described in the language POOSL (Parallel Object-Oriented Specification Language). This description is called a unified model because it encompasses both behavior and structure. A unified model is a formal description of a system in the language POOSL. POOSL differs from traditional (parallel) object-oriented (specification) languages in a number of ways. First of all, it allows explicit representation of system architecture and hierarchy. Further, process objects (see 2.5) in POOSL communicate through channels using a one-way synchronous message passing mechanism, allowing true parallelism. Finally, the language explicitly distinguishes (statically interconnected) process objects from (dynamically moving) data objects (see 2.6). The POOSL description consists of the definition of the instance variables (see 2.4) and the communication channels of the class. The message interface gives a list of all possible messages on all channels of the class. The initial method is where an object starts performing behavior for the first time.

The language POOSL has a formal semantics. One big advantage of describing behavior unambiguously is the ability to simulate this behavior by means of a computer. The possibility to translate this formal description into a language like C++ is also very valuable. In Appendix II the POOSL descriptions for all developed classes can be found. In [3] a precise description of POOSL is given.

2.4 Classes and objects

The message flow diagram and object structure diagram (which together form the object instance diagram) display the objects inside the system. Note that these diagrams use objects, not classes. Object instance modeling plays a major role in the SHE method. In SHE an object models a concept or an entity from a problem domain. Objects have properties that characterize them. These properties fall apart into observable behavior and descriptive features. A class is a collection of identical objects. Objects are created (or instantiated) from a class. Different objects instantiated from the same class have the same externally observable behavior but (may) differ in their descriptive features. The object behavior can be influenced by sending messages to the object. What kind of messages an object responds to is defined in its object class definition. During analysis the descriptive features of an object are modeled as attributes in its class definition. When an object is instantiated from its class these attributes get specific values. These attributes become instance variables in the formal behavior description of the object classes in POOSL. Object classes can be visualized in
object class diagrams, see Figure 2-3. The method SHE distinguishes two sorts of objects: data objects and process objects. These sorts have a different semantics and purpose. What type an object is of, is identified in its class sort field. So, data objects are instantiated from data classes and process objects from process classes. As said before the set of attributes in a class determines the properties (=descriptive features) of objects of that class. These attributes are encapsulated and only operations belonging to the object can modify them. An operation in an object is activated after reception of a message. What messages an object will accept is defined in the messages field of its class.

2.5 Process objects
An important feature of process objects is that they are autonomous concurrent entities and can exchange messages without becoming passive. The internal behavior of process objects however is sequential. Process objects contain internal data in the form of data objects (see 2.6) which are stored in instance variables. Process objects offer the power to express concurrent independent system parts connected in a static structure. These process objects or processes are connected by statically defined channels through which they can communicate by exchanging messages. A specification in POOSL consists of a fixed set of statically interconnected distributed processes. The channel structure connecting the process objects is graphically displayed in instance structure diagrams (see also 4.4). The messages over that channels are graphically displayed in message flow diagrams (see also 4.5).

2.6 Data objects
A data object consists of some private data and has the ability to act on this data. This private data cannot be accessed directly by another object. It can only be read and changed by the object itself.
As said before, process objects can interact by sending messages to each other. These messages can contain data which must be passed to a next process for further transformation. We use data objects to model such data. A data object that is applied this way is called a travelling object. Data to be sent may have a complex structure.
Complex data structures are modeled as a composite of data objects, linked by references. The basic building blocks of such a structure can be relatively simple data objects. The work of defining all classes is decreased by having some primitive classes. POOSL offers the following primitive data classes: boolean, integer, real and char. In these classes the usual relational and mathematical operations are defined. These primitive data classes are extended with primitive data objects and the object nil. These objects can be used to model non-deterministic system behavior.

2.7 Summary

In this chapter a short explanation of important SHE concepts has been given. These concepts are: the conceptual solution, different formal and non-formal models, the principle of instance modeling, process objects and data objects. A complete in-depth explanation of the SHE method is given in [1]. In the following chapter (chapter 3) a conceptual solution for building the model for packet switch fabrics according to the SHE method is given. The conceptual solution is the first steps towards SHE models. The different SHE models used to model the packet switch fabric are described in chapter 4.
3. The conceptual solution

3.1 Introduction

As developer of a (new) system you are confronted with a problem. From scratch there are constraints and requirements in solving this problem. When the developer is thinking about the problem and takes the initial constraints and requirements into account he or she creates a possible solution in mind (see paragraph 2.2). In this thesis the modeling and simulation of a packet switch fabric using SHE are described. My conceptual solution for doing this is given in this chapter.

3.2 The abstraction level

Building a model for a complex system existing in 'the real World', in this case a packet switch fabric, asks for the principle of abstraction. Abstraction enables highlighting of important details. It implies hiding of irrelevant details. Abstraction involves identification of the important qualities, capabilities and properties of the phenomenon that must be modeled [1]: Page56 "Handling Complexity".

For building the model of the packet switch fabric the abstraction level is the cell. This abstraction on cell level enables us to reason about the generating, routing, switching and receiving of cells without concerning the amount and meaning of bits and bytes present in a cell. Because only the performance of the packet switch fabric is of interest a cell used in the simulator is modeled by a few parameters, just enough to simulate routing and switching principles and to measure cell-delay. Of course, future extensions are possible (e.g. cell-priority could be added to each cell).

3.3 Finding the conceptual solution

One important constraint in building the model is the possibility to do simulations of different packet switch fabrics under the same circumstances. Each switch (fabric) has its own architecture and functionality but must have uniform communication behavior to its environment. This enables a smooth replacement of switches in the model. The environment in this case are generators producing cell traffic and receivers to switch these cells to. To enable model simulations using different cell routing strategies (but using the same switch fabric) this task is assigned to a separate part and thus left out of the switch. This part is located between the generators and the switch fabric. Further a 'third party' is needed to control the environment and switch fabrics. With this in mind the dividing of the total system into its main parts is performed. Finding these parts (see Figure 3-1), their interconnection and assigning each part its main functionality leads to the conceptual solution.

We can identify the following parts in the conceptual solution:

- Controller/Inspector
- Cell generator
- Stage addresser
- Switch(fabric)
- Cell receiver

The division of my conceptual solution into these parts is useful in finding the main objects and to build a framework for the project. It also enables an informal description of each part individually, as done in the following paragraphs.
3.4 Cell generator

To let the switch do its work -that is switching cells- traffic must be applied to it. The task of the cell generators is to produce the desired traffic. Traffic in the context of packet switches is a flow of cells with properties like burstiness and load. For generating cell traffic the two state traffic model is used (see Appendix I). This model enables the generating of uniform and bursty traffic with properties as described above. The desired traffic load and average burst length the generators produce can be obtained by setting correct state transition probabilities in the two state traffic model. See Appendix I for an in-depth explanation concerning traffic generation using this two state traffic model.

The performance of a switch (fabric) is directly related to the offered traffic and simulation results are meaningless if the traffic properties during simulation are unknown. Each switch input has its own generator connected to it. The properties of the traffic and the offered load the generator produces (on an input) can be controlled by the controller/inspector. For routing through the switch the destination is important and to determine cell delay an ‘timestamp’ field must be added. A cell is generated by the cell generator (according to a predetermined traffic type and load) which fills in the destination and gives the cell the timestamp zero. The cell leaves the generator for its way to the destination.

3.5 Switch

3.5.1 Introduction

The purpose of a switch (fabric) is to route cells from a certain input port to a certain output port. This mechanism is called switching. The switching of cells must be done as quickly as possible but no cells may be lost. The goal of the model simulations is to obtain information about switch-performance in certain configurations (e.g. single/multistage) and environments (e.g. uniform/bursty traffic, low/high load). The ability to measure switch performance while changing certain switch parameters is also very interesting. Take for example the buffer size, the memory on the switch to temporarily hold cells. When we know by means of simulations how many cells are in these memories the size can be adjusted to these values and so it can be an important parameter in determining chip area.

3.5.2 Cell routing

Each generated cell contains a destination field -which is filled in by the generator- and arrives on a switch fabric input. The task of the switch fabric is to route these incoming cells to the correct switch fabric outputs, according to the cell destination. This switching can be
performed using various strategies. How the switching exactly is done depends on the switch definition and the switch architecture. I decided to switch cells by means of source routing, see 3.5.3.2. The only task of the stage addresser (see 3.6) is making up the routing table, that is appending routing tags to each generated cell.

3.5.3 The modularity concept

Switch systems come in a wide range of interconnection sizes and there are many applications that require switch fabric sizes larger than the size of a single switch element. The possibility for the modular growth is a very important feature of a switch architecture. Therefore, this feature is supported by the model. Two configurations can be used: single stage or multi stage. In the following two sub-paragraphs a short description of these configurations is given. In [4] more information concerning single stage and multi stage configurations can be found.

3.5.3.1 Single stage expansion

A single stage S*S switch fabric can be built from basic N*N switch elements. Figure 3-2 shows an example of a 2N*2N switch fabric configured from four N*N elements. Similarly, a 4N*4N switch fabric can be realized from four 2N*2N fabrics. Each input line has a unique path to every output port of the switch. For a switch fabric of moderate size, the single-stage approach is feasible but when a higher number of ports is needed the single stage takes too many switches for configuration and the multistage principle must be chosen.

![Figure 3-2 Single-stage expansion](image)

3.5.3.2 Multi stage expansion

In order to achieve very large fabric sizes, a multi stage configuration can be used. Figure 3-3 shows a three-stage configuration. The packet header contains one routing tag per stage. The routing tags are only used for the internal routing in the switch fabric. This concept is commonly referred to as source routing. When a switch fabric receives a cell on one of its inputs a control section inspects the cell destination and appends the correct routing tags for each stage to it in order to regulate the traffic through the switch.
A switch element uses the first field of the packet header as its routing tag and, in order to prepare the packet header for the next stage, removes its own tag so that the next tag becomes the very first field of the header. All tags are stored in a routing table inside the cell.

### 3.5.4 Queuing in packet switches

Due to the statistical nature of packet traffic, it can happen in any packet switch that packets/cells contend for the same output port. Therefore, queuing is mandatory for contention resolution. One of the main issues in developing switches is to find the right queue size and strategy. A good strategy results in good switch performance (e.g. low cell delay) and smaller queues need less chip area. As illustrated in Figure 3-4, there are various ways to place queues in packet switching fabrics, and they result in different performances.

![Figure 3-4 Queuing concepts for switches](image)

These are just some examples of queuing concepts, in practice there exist many more. It is obvious, the modeled switches used for simulation must incorporate a certain queuing concept. One is free to model and create other switches, each using their own strategy. Doing so results in the ability to compare performance using different switches under the same circumstances.
3.6 Stage addresser

3.6.1 The need for an addresser

The modularity principle is one requirement in developing the model, that is: it must be easy to create and perform a multistage simulation for switch fabrics. With this in mind, the idea of “addresser” is born. The addresser is located between the generator and the switch fabric. Each generated cell carries its destination in a destination field. This field is inspected by the addresser which makes up a routing table according to the used switch fabric configuration and (of course) the destination. This routing table contains one routing tag for each stage (see 3.5.2). The advantage of having this addresser is that the switches can operate independently and that these are not responsible for the overall routing, thus freed from the routing task. This central routing by one element also implies the ability to modify the used routing strategy quickly.

3.7 Cell receiver

The simplest part of the simulator is, for sure, the receiver. In case of an N*N switch fabric there are N receivers needed, one for each switch output port. The receivers are just there to have destinations to route the cells to. From nature they behave passively, but to obtain switch performance results some functionality must be added to them. The actions the receivers perform is the inspection of incoming cells, that is to read the cell delay, store this information in a table and write (on command) these data to disk so it becomes available for other applications. For instance, an application as MsExcel can import simulation results written by the receivers and create graphs out of it.

3.8 Controller/Inspector

3.8.1 Introduction

The above mentioned parts together form a switch fabric configuration. The generator produces cells, the stage addresser fills in the routing tags, the switch(es) route(s) the cells and finally the receivers consume the cells. To obtain a correct interpretation concerning simulation results it is important to have information about the conditions under which the results were produced. Also the simulation must be started, stopped and initial simulation settings must be done. The controller/inspector is responsible for these tasks. In Figure 3-5 the normal operation sequence of the controller/inspector is displayed.

3.8.2 The role as controller

Each part of the model has its own tasks and together they form a complete switch fabric environment. The parts work independently on the same level, in fact the only relation between these parts is the cell flowing from one part to another. Having one controller in the system to which all parts ‘listen’ supports central simulation settings. As a user you make up the simulation settings on one logical place in the system. On startup the controller/inspector presets the individual parts, during simulation it can adjust these settings. For example the main settings for the generator are the load, number of destinations and -when generating bursty traffic- the average burst size. One important setting for the switch is its buffer length. Simulations are done with a number of cells and on different loads. The controller role is to let the generators switch from one load to another.

3.8.3 The role as inspector

A simulation without results is worth nothing. Therefore, the switch and receiver are able to inspect and store simulation information. The switch continuously inspects the buffer length
(how many cells are in) and the receiver reads the cell delay for each incoming cell. This information can reveal the need of bigger/smaller switch buffers and cell delay is a measure of switch fabric performance. To determine the end of inspection and the moment of storage to disk the inspector role is there. On receiving an ‘inspect’ command the current element stores its inspect information to disk.

![Diagram of control/inspect commands](image)

Figure 3-5 The control/inspect commands

### 3.8.4 Controlling simulations

After a model simulation is started the first control message is given by the controller/inspector. This control message carries simulation settings in it. The control message is sent to all model parts and all parts (re)start operation on receiving this message. After some predefined simulation time an inspect message is sent by the controller/inspector. This inspect message is sent to all objects. After receiving of an inspect message, the object stores its simulation results to disk. Next, another control message is sent by the controller/inspector and another simulation will be performed under modified conditions. This process is repeated over and over, until the controller/inspector decides to stop the total simulation. In one simulation loop simulations are done for different loads. See also Figure 3-5.

### 3.9 Summary

This chapter explains the principle of abstraction and modeling. The abstraction level used for building the simulator is determined and a dividing of the simulator in the main parts: controller/inspector, cell generator, stage addresser, switch(fabric) and cell receiver is given. Further the functionality and properties of each of individual parts is described. This descriptions together form the conceptual solution from which the SHE models are built.
4. Towards SHE models

4.1 Introduction
In previous chapter a conceptual solution for the model of a packet switch fabric is given. This conceptual solution can be seen as a very informal specification of the model. To reason about the exact behavior of a model we need a more formal specification of it. The SHE method assists in transforming this informal specification into a formal one. This formal behavior specification in POOSL can be simulated by a computer. SHE makes the step from (informal) conceptual solution to (formal) POOSL specification easier by offering more modeling views. Each view uses a different level of abstraction. This chapter gives some examples of SHE models (see Chapter 2) used for building the packet switch fabric simulator.

4.2 Process objects in the simulator model
The simulator complexity is handled according to the SHE method- by modeling it as a collection of collaborating process objects. Process objects have concurrent operation and can be interconnected by means of channels. This is an important feature because the simulator consists of parallel-operating parts which do communicate. Take for example a multi stage configuration, built out of concurrent operating switches. The interconnection between the switches is identified by channels and the messages over that channels are the cells a switch processes and forwards to its neighbor.

The process classes below are ‘found’ and used for building the packet switch fabric model. For each class a short description is given. A complete POOSL description of these classes can be found in the appendices.

Process classes used for the packet switch simulator:

- BurstSizeGenerator* Generates bursty traffic using bursts with a length Size-1,Size or Size+1. This class is currently not up to date.
- BurstGeometricGenerator* Generates bursty traffic using bursts with a length following a geometric distribution.
- UniformGenerator* Generates uniform traffic.
- StageAddresser4m4 Makes up the routing-table (See 3.5.2) for a 4m4 switch fabric. This 4m4 fabric is built from 2m2 switches.
- StageAddresser2m2 Makes up the routing-table (See 3.5.2) for a 2m2 switch.
- Switch2m2 Packet switch with 2 inputs and 2 outputs.
- CellReceiver Cell destination, here cells are routed to.
- ControllerInspector Gives commands to start, stop and influence the simulation. Takes care that the simulation results are stored to disk.

*The Generators used in the model consist of a traffic generator which can produce 2 types of traffic: uniform and burst. For future simulations other traffic types can be studied and implemented as cell-generators. The two state models used for the uniform and bursty traffic generators can be found in Appendix I.
4.3 Data objects in the simulator model

Data objects model reusable data structures and their accompanying operations. Data objects are instantiated from data classes. Data objects model passive entities. They must receive a message to become active. Data objects that exchange messages perform together sequential behavior. Only one data object is actively performing operations at the same time. Data objects can be linked together to form a complex structure. The basic building blocks of such a structure can be relative simple data objects. The work of defining all classes is decreased by having some primitive data classes. POOSL offers the following primitive data classes: boolean, integer, real and char. The SHE simulator tool comes with a library containing a lot of predefined data classes which are ready for (re)use (see 5.1.2).

If the static interconnected process objects want to exchange data this data can be modeled by data objects. A data object that is applied this way is called a travelling object. A cell in a packet switch fabric environment is a good example. The cell travels from generator to stage addresser, switch(es) and finally reaches the receiver. The data objects developed for the simulation of the packet switches are given below. A complete POOSL description of these classes can be found in the appendices.

- **Cell** Travels from generator to receiver. Holds age-information to measure cell-delay, an destination field and a routing-table.
- **FIFObuffer** This data object was already available in the SHE simulator data class library. In [3]-Paragraph 2.3 the FIFObuffer is described. For the packet switch simulation some additions to this object have been made.
- **SimulationSettings** This data object contains simulation settings for all parts of the total system. Each part reads its own settings from this broadcasted object.
- **RandomIntGenerator** The SHE simulator comes with a RandomGenerator object which generates real values between 0 and 1. The RandomIntGenerator generates integer values between 1 and a given parameter (e.g. to determine random a cell/burst destination)

4.4 Instance structure diagram

Process objects communicate via channels. Channels and messages accepted on channels are defined in the message interface of process definition. The channel interconnect topology of process objects can be visualized in so called instance structure diagrams. The static interconnect structure both models static structure of communication between concurrent instances (process objects), and a structure that may be imposed by architecture requirements.

Figure 4-1 shows an example of an instance structure diagram for a 2m2 Switch. This model plus the model for a 4m4 switch fabric can be found in Appendix III.
As described in chapter 3 the different 'parts' (e.g. switch and receiver) perform communication. These parts are process objects in the SHE model. In SHE the communication between process objects is visualized by a message flow diagram. Figure 4.2 shows an example of a message flow diagram for an 2m2 switch. This model plus the model for an 4m4 switch fabric can be found in Appendix III.

The conceptual solution consists of different parts (see chapter 3). These parts become process objects in the message flow diagram and the instance structure diagram. The data exchanged between the process objects is modeled by data objects. The informal models form the base for the formal model, consisting of a behavior description in the language POOSL.
5. Performing simulations

5.1 Introduction

When all SHE modeling and specification steps are taken an exact formal behavior description in the language POOSL is available. This formal behavior description can be fed to a computer which can simulate the described behavior. To support the simulation of POOSL behavior descriptions a special tool is developed. A short description of this tool is given in this chapter.

For correct interpretation of simulation results it is important to know the circumstances under which these results are produced. In case of the packet switch fabric model simulator for example we have results produced under a certain traffic load. How can we be sure that this traffic load is really applied to the switches and what do we mean with load in the context of packet switches? Answers to these questions can be found in this chapter.

5.2 The simulation tool coming with SHE

5.2.1 Introduction

This thesis concerns the building of a model for packet switch fabrics using SHE and SHE’s simulation tool. This simulation tool enables us to simulate the model described in the formal language POOSL. The tool itself is written in the language SmallTalk.

As said before, the SHE method includes a formal specification language called POOSL. The advantage of having a formal description of a system is the possibility to simulate system behavior on a computer. To do so, a (user friendly) simulator tool is necessary. The SHE-simulation tool consists of a user environment and a system part. For the user of the SHE simulator the working screen is called ‘drawing’ screen. From this screen new POOSL classes can be added to the SHE simulation tool, simulation commands can be given, objects can be inspected, messages can be displayed and different process objects with their interconnection channels can be drawn. The layout of this drawing screen can be found in Appendix IV.

5.2.2 The library coming with the SHE simulation tool

To support reuse the SHE simulator comes default with a lot of predefined data classes stored in a library. Default, this library contains no process classes. It is possible to add new classes to this library. When a new process or data class is added to the library a translation of the POOSL code into SmallTalk code is done. The library further has an import/export function to read/store classes from/to disk. This contributes to and supports the concept of class reuse. All used objects are instantiated of a class residing in the library.

5.2.3 Normal simulator operation

After starting the SHE simulator (that is loading the image file) the ‘drawing’ screen can be opened. The next step is input of system behavior by adding new POOSL classes to the SHE simulation library. Thereafter, the system architecture can be drawn graphically on the drawing screen. This drawn architecture can be saved to a file and saved architectures can be read directly from disk. On this drawing screen, objects are instantiated and their interconnection channels are made. All drawn/used objects are instantiated from a class available in the library. In case object classes are not coming with the SHE simulation tool, this library has to be extended with the new object classes. Normally, this is one of the first steps to take.
After completing the drawing of the process objects and their interconnection channels the simulation can be started. During simulation messages flowing from one object to another can be displayed on the channels in the form of so called interaction diagrams showing all instances and their communication. Also all drawn objects can be inspected (e.g. instance variables, local variables, current code-line).

When selecting a drawn object the appropriate POOSL code (in fact its class definition) can be changed and the object plus all objects instantiated from the same class take over the new behavior.

5.3 Understanding the POOSL semantics

The behavior description of all objects is written in the formal language POOSL. One important feature of this POOSL code is that the execution of it takes no time. With time in this context the simulation time is meant. Of course, the execution of this code (on a machine) takes 'real time' but to be independent of environment on which POOSL code is ran an simulation clock is necessary. As long as there are processes in the system performing POOSL code this simulation clock stands still. It only ticks if all processes cannot perform a POOSL instruction or execute a delay(t) POOSL instruction. This delay(t) instruction is used for synchronization in performing packet switch fabrics simulations. It can be interpreted as: stop performing behavior and sleep during a simulation time of t. In the following paragraphs t can be any number but in case of the packet switch simulation t = 1 has been used.

5.4 Synchronize processes

In POOSL all process objects perform concurrent behavior. In the context of the packet switch fabric simulation: the generators, stage addresser, switches and receivers do operate concurrently. As said before the performance of switch fabrics is dependent of the offered load by the generators. To have an good understanding of what is meant by load in this context a definition of load is given below.

\[
\text{Load is the ratio of the offered amount off cells on a switch input during a time interval } t \text{ to the maximum amount of cells which can be accepted on that switch input during that time interval } t.
\]

The problem now is: how do we model this time interval t if POOSL instruction execution takes no time? This problem is solved by using the POOSL 'delay(t)' instruction. The location and usage of this instruction for each simulation part is given below. Note however that these parts perform their behavior concurrently.

Generator
The Generator performs POOSL instructions in which it decides to send a cell or to keep silent (according to the two state traffic model, see Appendix I). When the generator decides to send a cell it fills in the cell destination field, adds a timestamp zero to it and finally sends the cell to the stage addresser. It then performs a delay(t) instruction and waits till the simulation time is ‘t’ further. After simulation time is t further it starts again. Note that the whole generator behavior including sending the cell takes no simulation time!

Stage addresser
The stage addresser does perform an endless loop. Incoming cells can interrupt this loop (they are transported using interrupt messages, See [1]: Paragraph 6.3.6.5). In this loop a dummy instruction is performed and thereafter the stage addresser waits till the simulation time is t further. After the simulation time is t further the loop is started over. As said before, while waiting it accepts the cells coming from the generator. While receiving a cell on a certain input it fills in the routing table for that cell and forwards this cell immediately to a
switch. Note that the whole process of receiving a cell, making up the routing table and sending the cell consumes no simulation time. The dummy instruction is needed for the creation of the loop. The delay(t) in the endless loop is needed to let the stage addresser stop performing behavior. The simulator clock only ticks when all processes are standing still and an endless loop with POOSL instructions (except the delay) in it consumes no simulation time!

Why using an endless loop with interrupt messages? Cells can arrive at each time on each input. Polling all inputs once (after all generators are ready and waiting) for incoming cells would be possible but a fair polling strategy would be needed. Receiving cells on interrupt base is more fair (first come, first serve) and no special strategy is needed.

Switch2m2
The switch handles two-way traffic. Cells come in on its input ports and cells present in the output buffers must be sent further over its output ports. To prevent cells from flowing directly from input to output the switches perform a two stage operation. In the first stage the switches send a cell for each output port if the appropriate output buffer is not empty. Meanwhile, incoming cells are stored in a input buffer. There is one input buffer available for each switch input port. The first stage ends with a delay(t/2) POOSL instruction. When the simulation time is t/2 further we are sure that all generators are waiting (stopped behavior), that is, we are sure no cell will be generated till the simulation time is another t/2 further (= during second stage). Next the switches inspect their input buffers, take the tag from the cell routing table and store the cells in the right output buffers. Finally the second delay(t/2) instruction is performed and the switches wait for the simulation time to tick t/2 further. When this moment is reached the switch, stage addresser and generator start performing their behavior again.

Receivers
The receivers do not use the delay(t) instruction. They just wait for incoming cells or commands from the controller/inspector. Its whole behavior consumes no time.

Controller/Inspector
During the simulation time interval t, zero or one cell is generated by each generator and for each switch input a maximum of one cell can be accepted. With this in thought we can produce a desired load. If a generator has to produce a load of x (0<=x<=1) it produces an average of x cells in the interval t. Because the generator can only generate one whole cell in the simulation time t this implies that a load of x means 1 generated cell in average (1/x * t) simulation time.

The controller/inspector is responsible for the simulation settings. One of these settings is the amount of cells used for a simulation under a certain load. Once this amount of cells is generated the controller/inspector gives inspect commands and instructs the generators to increase load. A load of x and the setting of k cells per load results in a simulation time of (1/x * t) * k. The controller/inspector calculates this simulation time for each load and waits till this time is reached.

In Figure 5-1 the timing and usage of the delay(t) is graphically displayed.
5.5 Summary

The method SHE comes with a special simulation tool. This tool is able to simulate the formal POOSL behavior description. According to the POOSL semantics the execution of POOSL instructions consumes no simulation time. As long as there are objects performing behavior the simulation clock stands still. It only ticks if all objects are stopped or execute the delay(t) instruction. To perform correct model simulations we need to synchronize the different processes in it. This synchronization is done by means of the delay(t) instruction.

5.6 Conclusion

For most systems (e.g. protocol specifications) the current speed at which a model can be simulated is sufficient. However, for statistical analysis -currently- the speed is not high enough. The simulation of packet switch fabrics is such a statistical simulation using very much cells to measure the behavior. This implies that very many communication steps (messages between objects) must be taken. Each communication step takes little time, but all together they consume hours. The SHE simulator tool is written in SmallTalk. SmallTalk is an interpreted language and for high speed simulations the usage of compiled code is more appropriate. Therefore, the SHE-team decided to develop a POOSL to C++ compiler. At the moment of writing this thesis the POOSL to C++ translation is almost possible. The simulation performance using C++ is definitely better than the SmallTalk simulations, but the actual performance of a model simulation using C++ is difficult to predict.
6. Conclusions

To determine packet switch fabric performance, simulations can be done. These simulations can reveal important information needed in developing (new) packet switches. For instance, knowledge about switch queue length can assist in determining chip-area needed for buffers to hold those queues.

A computer is capable to simulate formal (behavior) descriptions. This behavior is defined in a formal model. However, the step towards a correct formal model is far from trivial. There exist many modeling methods. Most of them only assist in the creation of non formal models. The SHE method supports besides the creation of these non formal models the creation of a formal model. This formal model consists of a behavior description in the language POOSL. The step from POOSL description towards implementation is easy to take.

SHE comes with a tool which is able to simulate the formal POOSL behavior descriptions. This tool is very helpful in reasoning about the performance and correct operation of a complex system or protocol. For statistical analysis such as the simulation of a packet switch fabric, the simulation speed of the tool is not high enough. This is due to the interpreted Smalltalk language the tool is written in. Currently, a POOSL to C++ compiler is almost available. The simulation performance using C++ is definitely better than the SmallTalk simulations, but the actual performance of a model simulation using C++ is difficult to predict.

During packet switch fabric simulations the cell delay and the switch buffer length are measured. Simulations are done using uniform and bursty traffic. The cell delay for bursty traffic in all cases is larger than the cell delay for uniform traffic. Simulations reveal that switch buffer length and cell delay are directly related to another.
7. References

[1] Putten, Piet H.A. van der en Jeroen P.M. Voeten
SPECIFICATION OF REACTIVE HARDWARE /SOFTWARE SYSTEMS.
Universiteitsdrukkerij Eindhoven, 1997

[2] Rumbaugh, James en Micheal Blaha, William Premerlani, Eddy Frederick,
Willim Lorrensen
OBJECT-ORIENTED MODELLING AND DESIGN.
New Jersey: Prentice-Hall, 1991

POOSL: AN OBJECT-ORIENTED SPECIFICATION LANGUAGE FOR THE
ANALYSIS AND DESIGN OF HARDWARE/SOFTWARE SYSTEMS.
Eindhoven University of Technology, Fac. Electrical Engineering EUT Report 95-E-290
Cip-data Koninklijke Bibliotheek Den Haag, 1995

A FLEXIBLE SHARED-BUFFER SWITCH FOR ATM AT Gb/s RATES
Computer Networks and ISDN Systems
Appendix I Two state traffic model

1 Uniform Traffic

Traffic is said to be uniformly distributed if the cells arrive independently from each other, the load on all input links is the same and all cells have an equal probability to be directed to an arbitrary output link.

The model used for uniform traffic is shown in Figure 7-2. The model consists of two states: a burst state where a cell is generated and a silence state where no cell is generated. Each time slot the system remains in its present state or makes a transition to the other state. If the system is in the burst state B, it has a probability of $P_{BB}$ to stay in this state and a probability $P_{SS}$ to go to state S. If the system is in the silence state S, it has a probability $P_{SS}$ to remain in this state, and a probability $P_{SB}$ to go to state B. If $P(B,t)$ and $P(S,t)$ are the probabilities that the system is in state B and state S respectively in time slot $t$ then the following equations can be made:

\[ P(B,t+1) = P(B,t) \cdot P_{BB} + P(S,t) \cdot P_{SB} \]
\[ P(S,t+1) = P(B,t) \cdot P_{BS} + P(S,t) \cdot P_{SS} \]

Equation 1
Equation 2

For uniform traffic the system is memoryless, which means that the probability that the system enters state B at the next time slot is independent of the current state the system is in. The same holds for state S. Therefore the probabilities $P_{BS}$ and $P_{SB}$ are the same and thus the probabilities $P_{BS}$ and $P_{SS}$. Define $P_{B}$ and $P_{S}$ as $P_{B}=P_{BB}=P_{SB}$ and $P_{S}=P_{BS}=P_{SS}$ respectively, with $P_{B}=1-P_{S}$. For steady state the probabilities that the system is in state B or state S is independent from $t$. The steady state equations for this uniform traffic model become:

\[ P(B) = P(B) \cdot P_{B} + P(S) \cdot P_{B} = P_{B} \cdot (P(B) + P(S)) = P_{B} \]
\[ P(S) = P(B) \cdot P_{S} + P(S) \cdot P_{S} = P_{S} \cdot (P(B) + P(S)) = P_{S} \]

Equation 3
Equation 4

From the equations follow that the probability that the system is in state B is equal to the transition probabilities $P_{BB}$ and $P_{SB}$. Likewise follows that the probability $P(S)$ that the system is in state S is equal to $P_{BB}$ and $P_{SS}$. The probability $P(B)$ is equal to the part of the time the system spends in state B and thus equal to the load on the input link. Therefore $P_{BB}$ and $P_{SB}$ are equal to the load $\rho$ and the probabilities $P_{BS}$ and $P_{SS}$ are equal to $1-\rho$.

Uniform traffic has been simulated as follows: Choose the desired load $\rho$ for the switch and the number of inputs $N$. At every time slot there is a probability $\rho$ that a cell arrives on an
input link. If a cell arrives on $P_{BB}$ an input link it has a probability $1/N$ that it is directed to a specific output. Thus the probability that a cell arrives destined for a specific output is $p/N$. This is done for each input link.

2 Bursty traffic

A burst is a sequence of cells coming right after another and having the same destination. Traffic is said to be bursty if the bursts arrive independently from each other, the load on all input links is the same and all bursts have an equal probability to be directed to an arbitrary output link.

In practice it is not likely that the cells offered to a switch will be as smoothly distributed as in uniform traffic. Especially data traffic is known to be very bursty. Therefore the generator also must be able to produce this kind of traffic.

The model used for bursty traffic is essentially the same as the one for uniform traffic with different values for the transition probabilities $P_{BB}, P_{BS}, P_{SB}$ and $P_{SS}$. A burst in this model is a number of cells in consecutive time slots destined for the same output. Two parameters describe the system fully. It is chosen to describe the system by the desired load $p$ on the input links and the average number of cells $b$ in a burst. The average time the system spends in state $B$ is equal to the load $p$ on that link, therefore $P(B) = p$.

The average burst length can be calculated as follows: The probability that a burst consists of $k$ cells is equal to the probability that the system spends $k-1$ time slots in state $B$ and then makes a transition to state $S$:

$$P(\text{burst length} = k) = P_{BB}^{k-1} \cdot P_{BS} = P_{BB}^{k-1} \cdot (1 - P_{BB})$$  \hspace{1cm} \text{Equation 5}

The burst length follows a geometric distribution. The same way it can be shown that the number of empty slots follow a geometric distribution with probability $P_{SS}$. The average burst length $b$ now becomes:

$$b = \sum_{k=0}^{\infty} k \cdot P_{BB}^{k-1} \cdot (1 - P_{BB})$$

$$b = \frac{1}{P_{BB}} \sum_{k=0}^{\infty} k \cdot P_{BB}^k \cdot (1 - P_{BB})$$

$$b = \frac{1}{P_{BB}} \cdot \frac{P_{BB}}{(1 - P_{BB})}$$  \hspace{1cm} \text{Equation 6}

Using $P_{BS} = 1 - P_{BB}$ the transition probabilities $P_{BB}$ and $P_{BS}$ become:

$$P_{BS} = \frac{1}{b}$$  \hspace{1cm} \text{Equation 7}
Using Equation 1 and recalling $P(B) = \rho$ and thus $P(S) = 1 - \rho$:

$$\rho = P_{bb} \cdot \rho + P_{sb} (1 - \rho)$$

$$P_{sb} = \left(1 - \frac{b}{b} - 1\right) \cdot \frac{\rho}{1 - \rho}$$

$$P_{sb} = \frac{1}{b} \cdot \frac{\rho}{1 - \rho}$$

Equation 8

And using $P_{ss} = 1 - P_{sb}$:

$$P_{ss} = 1 - \frac{1}{b} \cdot \frac{\rho}{1 - \rho}$$

Equation 9

Take for example a desired load $\rho = 0.6$ and an average burst length $b = 4$. The transition probabilities become: $P_{bs} = 0.25$, $P_{bb} = 0.75$, $P_{sb} = 0.375$ and $P_{ss} = 0.625$.

This model cannot be used for high loads with a small average burst length. Take for example a load of 0.8. The term $\rho/(1-\rho)$ becomes 4. This means that the average burst length should be at least 4. A desired load of 0.9 and an average burst length $b = 3$ results in $P_{bs} = 0.33$, $P_{bb} = 0.67$, $P_{sb} = 3$ and $P_{ss} = -2$. The impossible values of the probabilities $P_{sb}$ and $P_{bb}$ imply that if the system enters state $S$, it will return to state $B$ at the next time slot. The number of empty slots between bursts cannot vary any more, but is always one. The effective load becomes:

$$P(B) = P(B) \cdot P_{bb} + P(S) \cdot P_{sb}$$

$$P(B) = P(B) \cdot P_{bb} + (1 - P(B)) \cdot P_{sb}$$

$$P(B) \cdot (1 - P_{bb} + P_{sb}) = P_{sb}$$

$$P(B) = P_{sb} / (1 - P_{bb} + P_{sb})$$

$$P(B) = 1 / (1 - 0.67 + 1)$$

$$P(B) = 0.75$$

The actual load during the simulation will be 0.75, which is significantly lower than the expected load of 0.9. Another way to look at it is as follows: If the desired average burst length is chosen to be 3 and after each burst has to follow at least one empty slot then the maximum load is $burst\ length/(burst\ length + silence\ length) = 3/(3+1) = 0.75$. The maximum throughput at a desired average burst length 3 is 0.75. The reason that no any combination of average burst length and load is possible is that both the burst length and the silence length are integers which poses an upper and lower limit on the desired load. In the previous mentioned example of an average burst length of 3 and a load $\rho$ of 0.9, the average silence length should be $1/3$, which is smaller than the lowest possible integer 1.
Appendix II  Process and Data Objects used in the Simulator

In this appendix the process- and data-objects used for building the simulator (see chapter 2) are described. The complete POOSL code of each object is added directly after this object description.

1 Process Objects used for the simulator

• BurstGeometricGenerator

The BurstGeometricGenerator objects generate cells in a burst. The average burst size is determined by the value of instance variable 'averageburstlength'. The BurstGeometricGenerator gives each generated burst a destination, that is: the cells in the same burst all have the same destination. To do so the number of used destinations has to be known and is therefore stored in instance variable 'nrofdest'. For communication it has two interfaces, one to receive control information and one to send the generated cells. The BurstGeometricGenerator further uses the data objects RandomGenerator and RandomIntGenerator to do the 'statistical work', that is implementing the two-state model as described in the paragraph 'Cell Generator' (using RandomGenerator) and determine the cell destination (RandomIntGenerator).

POOSL description:

```poosl
process class BurstGeometricGenerator()

instance variables
statecounter: Object; psilence: Object; pburst: Object;
cellcounter: Object; statebcounter: Object;
averageburstlength: Object; load: Object; nrofdest: Object

communication channels
out, ctrl

message interface
out ! route(UnKnownDataClass);
ctrl ? control(UnKnownDataClass)

initial method call generateBurst()()

instance methods
resetGenerator(settings : SimulationSettings)()
cellcounter:=0;
statebcounter:=0;
statecounter:=0;
averageburstlength:=settings getBurstSize();
load:=settings getLoad();
nrofdest:=settings getNumberOfDestinations();
psilence:=1-((1/averageburstlength)*(load/(1-load)));
pburst:=1-(1/averageburstlength).

generateBurst()()

| settings : SimulationSettings;
currentstate : integer; datacell : Cell;
randomgetal : object; randomizer : object;
destinationizer : object; destination : integer |```
datacell := new(Cell) init();

/* Wait for reset signal */
ctrl?control(settings);
resetGenerator(settings)();

/* Randomizer: real(0..1); destinationizer: integer(1...N), N=#destinations */
randomizer := new(RandomGenerator);
destinationizer := new(RandomIntGenerator)
initRandomIntGenerator();
datacell setDestinationTo(destinationizer getRandomIntNumber(nrofdest));

/* Determine initial state */
randomgetal := randomizer random();
if (randomgetal <= 0.5)
  then currentstate := 0;
    statebcounter := 0
  else currentstate := 1;
    statebcounter := 1
fi;

/* Generate bursts in a loop */
while (true) do
  if (currentstate = 1)
    then datacell setTotalDelayTo(0);
      out!route(datacell);
      cellcounter := cellcounter + 1;
  fi;
randomgetal := randomizer random();
if (currentstate = 0)
  then if (randomgetal > psilence)
      then currentstate := 1;
          statebcounter := statebcounter + 1;
      datacell setDestinationTo(destinationizer getRandomIntNumber(nrofdest));
      fi;
  else if (randomgetal > pburst)
    then currentstate := 0;
    fi;
fi;
delay(1);
statecounter := statecounter + 1;

od interrupt(ctrl?control(settings);
  resetGenerator(settings)();
).

- **BurstSizeGenerator**
  This class is currently NOT up to data. It was used early in the project and objects instantiated from it do not work properly in the current version of the simulator. The main class attribute is called 'burstSize'. The BurstSizeGenerator generates bursts with an burstsize of burstSize-1, burstSize or burstSize+1.

POOSL description:

process class BurstSizeGenerator(burstSize: Object; load: Object)

instance variables
randomizer: Object; randomgetal: Object; desiredload: Object; statecounter: Object; currentstate: Object; cellcounter: Object; dataCell: Object; silence: Object; burstcounter: Object; silencecounter: Object; bMinCnt: Object; bAverageCnt: Object; bMaxCnt: Object; silenceStateSetting: Object; randomtest: Object; random1: Object; random2: Object; randomtestnumber: Object; random3: Object; random4: Object

communication channels
out

message interface
out ! route(UnKnownDataClass)

initial method call setLoad()

instance methods
setLoad()
randomtest:=new(RandomN) initRandomNGenerator();
random1:=0; random2:=0;
random3:=0; random4:=0;
cellcounter:=0;
silenceStateSetting:=(burstSize*(1-load)/load;
silencecounter:=silenceStateSetting;
bMinCnt:=0; bAverageCnt:=0; bMaxCnt:=0;
dataCell:=new(Cell);
dataCell setDestinationTo(l);
statecounter:=0;
desiredload:=0.6;
randomizer:=new(RandomGenerator);
randomgetal:=randomizer random;
if (randomgetal<=0.5)
  then currentstate:=0;
else currentstate:=1;
fi;
if (load>0.99999) then currentstate:=1; fi;
randomgetal:=randomizer random;
if (randomgetal<=0.25)
  then burstcounter:=3;
else if (randomgetal<=0.75)
    then burstcounter:=4
    else burstcounter:=5
  fi;
fi;
while (true) do
  if (currentstate=0)
    then silencecounter:=silencecounter-1;
    if (silencecounter=0)
      then currentstate:=1;
      randomgetal:=randomizer random;
      if (randomgetal<=0.25)
        then burstcounter:=burstSize-1;
        bMinCnt:=bMinCnt+1;
      else if (randomgetal<=0.75)
        then burstcounter:=burstSize
        bAverageCnt:=bAverageCnt+1;
      else burstcounter:=burstSize+1;
      bMaxCnt:=bMaxCnt+1;
    fi;
  fi;
while (true) do
  if (currentstate=0)
    then silencecounter:=silencecounter-1;
    if (silencecounter=0)
      then currentstate:=1;
      randomgetal:=randomizer random;
      if (randomgetal<=0.25)
        then burstcounter:=burstSize-1;
        bMinCnt:=bMinCnt+1;
      else if (randomgetal<=0.75)
        then burstcounter:=burstSize
        bAverageCnt:=bAverageCnt+1;
      else burstcounter:=burstSize+1;
      bMaxCnt:=bMaxCnt+1;
  fi;
else out!route(dataCell);
    cellcounter:=cellcounter+1;
    if (load<O.999999) then burstcounter:=burstcounter-1;
    else
        if (burstcounter=0)
            then currentstate:=0;
        silencecounter:=silenceStateSetting;
    fi;
fi;
statecounter:=statecounter+1;
randomtestnumber:=randomtest getRandomNumber(4);
if randomtestnumber=1 then random1:=random1+1; fi;
if randomtestnumber=2 then random2:=random2+1; fi;
if randomtestnumber=3 then random3:=random3+1; fi;
if randomtestnumber=4 then random4:=random4+1; fi;
delay(1);
od.

- **UniformGenerator**

In principle the behavior of objects of this class is the same as that of the BurstGeometricGenerator objects. The only difference is in the probability that cells come immediately after another (=burst).

UniformGenerator objects give each cell its own destination. The interface and used data objects are the same as for the BurstGeometricGenerator

**POOSL description:**

process class UniformGenerator()

instance variables

cellcounter: Object; load: Object; nrofdest: Object; pburst: Object; psilence: Object; statecounter: Object

communication channels

out, ctrl

message interface

out ! route(UnKnownDataClass);
ctrl ? control(UnKnownDataClass)

initial method call generateUniform()()

instance methods

generateUniform()()
| settings : SimulationSettings;
  currentstate : integer; datacell1 : Cell;
  randomgetal : object; randomizer : object;
  destinationizer : object; destination : integer |
datacell:=new(Cell) init();
/* Wait for reset signal */
ctrl?control(settings);
resetGenerator(settings)();

28
/* Randomizer: real(0..1); destinationizer: integer(1...N), N=#destinations */
randomizer:=new(RandomGenerator);
destinationizer:=new(RandomIntGenerator)
initRandomIntGenerator();
datacell setDestinationTo(destinationizer getRandomIntNumber(nrofdest));
/* Determine initial state */
randomgetal:=randomizer random();
if (randomgetal<=0.5)
   then currentstate:=0
   else currentstate:=1;
/* Generate cells in a loop */
while(true) do
   if (currentstate=1)
      then datacell setTotalDelayTo(0);
      datacell setDestinationTo(destinationizer getRandomIntNumber(nrofdest));
      out!route(datacell);
      cellcounter:=cellcounter+1;
   randomgetal:=randomizer random;
   if (currentstate=0)
      then if (randomgetal>psilence)
           then currentstate:=1;
           fi;
      else if (randomgetal>pburst)
           then currentstate:=0;
           fi;
   fi;
   delay(1);
   statecounter:=statecounter+1;
   od interrupt(ctrl?control(settings));
   resetGenerator(settings());
resetGenerator(settings : SimulationSettings) ()
cellcounter:=0;
statecounter:=0;
load:=settings getLoad();
nrofdest:=settings getNumberOfDestinations();
psilence:=1-load;
pburst:=load.

- ControllerInspector
As described before this object controls the simulation. There is only one object of this class present in a simulation. Here the simulation settings are stored. These settings can/must be modified by the user before starting a simulation. The initial method is called 'simulate' and is implemented as some loops in which simulation settings are changed and instructions to other simulation objects are given. One loop (or run) results in a simulation using a traffic load of 0.2, 0.4, 0.6, 0.8 and 0.9. Inspection is done before switching to another load, that is after 'numberofcells_per_load' (a local variable) cells are generated by each generator. The communication channels are used to send control- and inspect commands and to receive acknowledges. The messages over the channels are typed 'broadcast', thus reaching all objects connected to the channels carrying them.

The POOSL message-interface-definition for ControllerInspector is as follows:
ctrl !* control(<Object Containing Simulation Settings>);
inspect !* inspectswitch(<SwitchNumber>);
inspect ? response();
inspect !* inspectreceiver(<ReceiverNumber>)

Here messages to be broadcasted are identified by '!' and messages to be received are identified by '?'. The response message is used as acknowledge. After some simulation time the ControllerInspector may decide to instruct receiver 1 to store simulation results. To do so it broadcasts an inspectreceiver(1) message and waits for a response message send by receiver 1. On receiving this response the ControllerInspector knows that the results are stored and continues operation.

POOSL description:

process class ControllerInspector()

instance variables
statecntr: Object; simulationstates: Object; currentload: Object; runnumber: Object; totalruns: Object

communication channels
inspect, ctrl

message interface
ctrl !* control(UnKnownDataClass);
inspect !* inspectswitch(UnKnownDataClass);
inspect ? response();
inspect !* inspectreceiver(UnKnownDataClass)

initial method call simulate()()

instance methods
administrate(nrofdest : integer)()
| index : integer |
index:=1;
/* Command all the Receivers to store their information */
/*
while (index<=nrofdest) do
inspect!*inspectreceiver(index);
inspect?response();
index:=index+1;
od;
/*
/* For the time being inspect only receiver 1 */
inspect!*inspectreceiver(1);
inspect?response();

/* Command Switches to store their information */
inspect!*inspectswitch(1);
inspect?response();
/* if Multistage --> command switch 2 (second stage) also */
if nrofdest>2 then
inspect!*inspectswitch(2);
inspect?response()
fi.

simulate()()
| burstsize : Integer; outputbufsize : Integer; nrofdest : Integer; numberofcells_per_load : Integer; settings : SimulationSettings |
| runnumber:=1; totalruns:=4; |
| while(runnumber<=totalruns) do |
  /* ------ Edit Here The Simulation Settings --------------- */
  burstsize:=10; outputbufsize:=2000; nrofdest:=4; numberofcells_per_load:=50;
  /* ------------------------------- */
  currentload:=0.2; simulationstates:=numberofcells_per_load/currentload;
  settings:=new(SimulationSettings);
  settings setLoad(currentload);
  settings setRunNr(runnumber);
  settings setBurstSize(burstsize);
  settings setOutputBufferSize(outputbufsize);
  settings setNumberOfDestinations(nrofdest);
  delay(1);
  ctrl!*control(settings);
  loop();
  administrate(nrofdest);
  currentload:=0.4; simulationstates:=numberofcells_per_load/currentload;
  ctrl!*control(settings);
  loop();
  administrate(nrofdest);
  currentload:=0.6; simulationstates:=numberofcells_per_load/currentload;
  ctrl!*control(settings);
  loop();
  administrate(nrofdest);
  currentload:=0.8; simulationstates:=numberofcells_per_load/currentload;
  ctrl!*control(settings);
  loop();
  administrate(nrofdest);
  currentload:=0.9; simulationstates:=numberofcells_per_load/currentload;
  ctrl!*control(settings);
  loop();
  administrate(nrofdest);
  runnumber:=runnumber+1;
  od.

loop() /* Loop the simulations states and activate/deactievate the 'inspecting process' in the receivers to eliminate startup distortion */
statecntr:=0;
while (simulationstates>statecntr) do
delay(1);
statecntr:=statecntr+1;

CellReceiver
This object consumes cells. Each output port of a switch is connected to a receiver. The
CellReceivers all have different numbers, so the ControllerInspector can address a receiver
individually. The receivers assign their 'receiver numbers' by themselves! After receiving the
first cell the receiver inspects the destination field of this cell and takes this as its number.
This principle enables dynamically assigning of receiver numbers and frees the user from the
task to do it manually. During simulation each received cell is inspected on its delay. This
data is stored in a dataarray, according to the following structure:

\[ \text{<index (integer)> <amount of received cells with a delay of 'index'>} \]

On receiving an inspectreceiver(number) message the receiver checks if the number equals
its receiver number and if so it calls its inspectReceiver method. From there a call to the
method toFile is made. The method toFile writes the dataarray to disk. In a text-file the data
is stored. This text-file has the following structure:

\[ \text{<index (integer)> <amount of received cells with a delay of 'index'>} \]

After the 'write to disk' action is completed the CellReceiver object sends a response
message to the ControllerInspector which continues operation. On receiving an control
message the CellReceiver calls its resetReceiver method in which the dataarray and
cellcounter are reset. From that moment the inspection data is stored in a new (empty)
dataarray.

POOSL description:

process class CellReceiver()

instance variables
cellcntr: Object; dataarray: Object; delaytime: Object;
mynumber: Object

communication channels
ctrl, inspct, in

message interface
inspct ! response();
ctrl ? control(UnKnownDataClass);
in ? route(UnKnownDataClass);
inspct ? inspectreceiver(UnKnownDataClass)

initial method call receive()()

instance methods
toFile(settings : SimulationSettings)()
| index : Object; waarde : Integer; file : FileOut; filename : String |
/* Make up a filename for storing results to disk*/
filename:="G:\r";
Switch2m2

In current simulations switches with a size of 2*2 (2 input and 2 output ports) are used. In future new switches with a higher number of ports can be built. The switches are output
buffered and perform a 'two phase operation'. During the first phase the output-buffers are inspected and if not empty one cell (the 'oldest') is taken out of it and sent to a receiver or neighbor switch. The switch receives cells on interrupt-basis. Incoming cells are stored in a one-place inputbuffer. During the second phase the input-buffers are inspected and if not empty the cell is taken out and stored in the appropriate outputbuffer. To find the correct outputbuffer the switch performs the 'cell readPath()' operation. The cell is a dataObject containing a writePath() and readPath() method. WritePath is used by the StageAdresser object to fill in the path through the switch(fabric) for the cell. When a switch calls ReadPath, the destination output port (=output buffer) for that cell is returned. See also Figure.7.3.

![Diagram of Switch2m2 used in simulator](image)

FIGURE 7.3 Switch2m2 used in simulator

POOSL description:

```poosl
process class Switch2m2(mynumber: Object)

    instance variables
    celloutputbuffer1: Object; celloutputbuffer2: Object;
    statecntr: Object; inputbufsize: Object; outputbufsize: Object; cellincntr: Object; lostcells: Object;
    cellinputbuffer1: Object; cellinputbuffer2: Object;
    dataarray1: Object; dataarray2: Object; samplecntr1: Object;
    samplecntr2: Object

    communication channels
    ctrl, in2, in1, inspct, out1, out2

    message interface
    inl ? route(UnKnownDataClass);
    in2 ? route(UnKnownDataClass);
    out2 ! route(UnKnownDataClass);
    out1 ! route(UnKnownDataClass);
    ctrl ? control(UnKnownDataClass);
    inspct ! response();
    inspct ? inspectswitch(UnKnownDataClass)

    initial method call switch()

    instance methods
    toFile(settings : SimulationSettings; bufnumber : Integer)()
    | index : Object; waarde : Integer; file : FileOut; filename : String |
    | /* Make up a filename for storing results to disk*/
    | filename:="G:\s";
    | filename:=filename concat(mynumber asString())
    | concat(bufnumber asString()) concat("_") concat(settings
```

34
getLoad() asString() concat("_") concat(settings getRunNr()) asString() concat(".txt");
index:=0;
file:=new(FileOut) destination(filename);
/* Writ array to disk, NOTE! number of occurences of 'x' can be found at array-position x+1 */
if (bufnumber=1) then
    file writeString("0 ") write(samplecntrl) writeString(" ");
    while(index<=dataarray1 getSize()-1) do
        waarde:=dataarray1 valget(index+1);
        file write(index) writeString(" ") write(waarde) writeString(" ");
        index:=index+1;
    od;
fi;
if (bufnumber=2) then
    file writeString("0 ") write(samplecntrl2) writeString(" ");
    while(index<=dataarray2 getSize()-1) do
        waarde:=dataarray2 valget(index+1);
        file write(index) writeString(" ") write(waarde) writeString(" ");
        index:=index+1;
    od;
fi;
file close.

inspectSwitch(settings : object) ()
toFile(settings,1)();
toFile(settings,2)();
inspect!response().

resetSwitch(settings : object) ()
cellincntr:=0;
statecntr:=0;
lostcells:=0;
dataarray1:=new(Array);
dataarray2:=new(Array);
samplecntrl1:=0;
samplecntrl2:=0;
outputbufsize:=settings getOutputBufferSize();
celloutputbuffer1 clear(outputbufsize);
celloutputbuffer2 clear(outputbufsize);
cellinputbuffer1 clear(1);
cellinputbuffer2 clear(1).

switch()()
| number,destination, switchnumber : integer;
cell, incell : Object; settings : SimulationSettings |
/* samplecntrl1:=new(Integer);
samplecntrl2:=new(Integer);
*/
cellinputbuffer1:=new(FIFOBuffer) clear(1);
cellinputbuffer2:=new(FIFOBuffer) clear(1);
celloutputbuffer1:=new(FIFOBuffer)
celloutputbuffer2:=new(FIFOBuffer);
cellinputbuffer2:=new(FIFOBuffer);
ctrl?control(settings);
resetSwitch(settings)();
while(true) do
  if (celloutputbuffer1 isNotEmpty)
    then cell:=celloutputbuffer1 read();
        cell setTotalDelayTo(cell getTotalDelay + (statecntr -
        cell getTimeStamp));
        out1!route(cell);
    fi;
  if (celloutputbuffer2 isNotEmpty)
    then cell:=celloutputbuffer2 read();
        cell setTotalDelayTo(cell getTotalDelay + (statecntr -
        cell getTimeStamp));
        out2!route(cell);
    fi;
  delay(0.5);
  if (cellinputbuffer1 isNotEmpty)
    then cell:=cellinputbuffer1 read();
        toOutputBuf(cell, cell readPath());
    fi;
  if (cellinputbuffer2 isNotEmpty)
    then cell:=cellinputbuffer2 read();
        toOutputBuf(cell, cell readPath());
    fi;
  delay(0.5);
  statecntr:=statecntr+1;
  if (statecntr>50) then
    number:=celloutputbuffer1 getNrOfElements();
    /* NOTE! number of occurrences of 'x' must be stored at array-
       position x+1          */
    /* Why? we cannot store values at array-position zero! 
     Therefore offset-array by 1 */
    dataarray1 put(number+1, dataarray1 valget(number+1) + 1);
    samplecntr1:=samplecntr1+1;
    number:=celloutputbuffer2 getNrOfElements();
    /* NOTE! number of occurrences of 'x' must be stored at array-
       position x+1          */
    /* Why? we cannot store values at array-position zero! 
     Therefore offset-array by 1 */
    dataarray2 put(number+1, dataarray2 valget(number+1) + 1);
    samplecntr2:=samplecntr2+1;
  fi;
  od interrupt(sel
    (in1?route(incell); incell setTimeStampTo(statecntr);
       cellincntr:=cellincntr+1; cellinputbuffer1 write(incell));
    or
    (in2?route(incell); incell setTimeStampTo(statecntr);
       cellincntr:=cellincntr+1; cellinputbuffer2 write(incell));
    or
    (ctrl?control(settings); resetSwitch(settings)());
    or
    (inspct?inspectswitch(switchnumber);
       if switchnumber=mynumber
          then inspectSwitch(settings)()
          fi;
    fi);)
  toOutputBuf(cell : object; outputbufnr : integer)()
if (outputbufnr=1) 
then if (celloutputbuffer1 getNrOfElements < outputbufsize) 
    then celloutputbuffer1 write(cell); 
    else lostcells:=lostcells+1; 
    fi; 
else 
if (outputbufnr=2) 
then if (celloutputbuffer2 getNrOfElements < outputbufsize) 
    then celloutputbuffer2 write(cell); 
    else lostcells:=lostcells+1; 
    fi; 
fi.

• StageAddresser2m2

The StageAddresser2m2 object occurs only once in a simulation. It is located between the generators and the 2m2switch. This object is responsible for the routing. For each generator the StageAddressor has an input (2) and for each 2m2switch input it has an output (2).

Below some POOSL code for handling a cell arriving on input port 1 is given.

........
<an incoming cell is detected (interrupt message)>
in1?route(cell); makePathList(cell)(); out1!route(cell)
........

Explanation: First the cell coming with the message 'route' is read from channel 'in1'. Next the routing method makePathList() is called. This method fills the cell's routing table with the correct values using the writePath method belonging to the cell. Finally the cell is written to output 'out1' and thus can be received by the (first) switch.

POOSL description:

process class StageAddresser2m2()

instance variables

communication channels
out1, in2, out2, in1

message interface
in1 ? route(UnKnownDataClass);
out1 ! route(UnKnownDataClass);
in2 ? route(UnKnownDataClass);
out2 ! route(UnKnownDataClass)

initial method call handleCells()()

instance methods
handleCells()()
    | cell : Object |
    while(true) do
    /* Dummy operation */
delay 1;
od interrupt(sel
    (in1?route(cell); makePathList(cell)();
     out1!route(cell))
    or
    (in2?route(cell); makePathList(cell)());
out2!route(cell))
  les
).

makePathList(cell : object)()
  | dest : integer |
  dest:=cell getDestination();
  if dest=1
    then cell writePath(1)
  else cell writePath(2);
  fi.

• StageAddresser4m4
This object has exactly the same behavior as the StageAddresser2m2 but can handle double
the amount of cells (4). This object is used for operation in a multistage configuration. In the
simulations of a 4*4 switch fabric build out of 4 2*2-switches this object adds a routing tag
for each stage resulting in a PathList depth of 2.

POOSL description :

process class StageAddresser4m4()

instance variables

communication channels
out4, out3, in3, in1, in4, in2, out1, out2

message interface

in1 ? route(UnKnownDataClass);
in2 ? route(UnKnownDataClass);
in3 ? route(UnKnownDataClass);
in4 ? route(UnKnownDataClass);
out3 ! route(UnKnownDataClass);
out1 ! route(UnKnownDataClass);
out2 ! route(UnKnownDataClass);
out4 ! route(UnKnownDataClass)

initial method call handleCells()()

instance methods

handleCells()()
  | x : integer; cell : Object |
  while(true) do
    /* Dummy operation */
    delay 1;
    od interrupt(sel
     (in1?route(cell); makePathList(cell)();
  out1!route(cell))
     or
     (in2?route(cell); makePathList(cell)();
  out2!route(cell))
     or
     (in3?route(cell); makePathList(cell)();
  out3!route(cell))
     or
     (in4?route(cell); makePathList(cell)());
2 Data Objects used for the simulator

- Cell
The cell is the most travelling object in the system. The cells used for the simulation contain information needed for the routing and switching, e.g., a destination field filled in by the generator. To obtain simulation information additional variables and methods are added. The following scenario explains the operations on a Cell which are initiated by different process objects during a simulation.

Generator object:
A certain generator instantiates a cell and makes a call to the Cell init() method. In this init() method the cell instance variable ‘totalDelay’ is set to zero and a pathbuffer to store routing tags is made available. Just before sending the cell to the StageAddresser object the generator fills in the destination, using the Cell method setDestinationToO.

StageAddresser:
This process object receives (a deep copy of) the cell just generated. The cell destination is read using the Cell method getDestination() and the correct routing tags are added to the Cell’s pathlist using the method writePath. This pathList is implemented as a FIFOBuffer.

POOSL description:

data class Cell

instance variables
destination: Object; timestamp1: Object; timestamp2: Object;
timestamp: Object; totaldelay: Object; pathbuffer: Object

instance methods
getTimeStamp1() : Integer
return (timestamp1).

init() : cell
pathbuffer:=new(FIFOBuffer) clear;
return (self).

getTotalDelay() : Integer
return (totaldelay).

addToTimeStamp1(value : integer) : Cell
timestamp:=timestamp+value;
return (self).

out4!route(cell))
    les
).

makePathList(cell : object)()
    | dest : integer |
    dest:=cell getDestination();
    if dest <=2
    then cell writePath(1)
    else cell writePath(2);
    fi;
    if (dest % 2) = 1
    then cell writePath(1)
    else cell writePath(2);
    fi.
getTimeStamp() : Integer
return (timestamp).

writePath(value : integer) : object
pathbuffer write(value).

addToTimeStamp(value : integer) : Cell
timestamp:=timestamp+value;
return(self).

getDestination() : Integer
return(destination).

setDestinationTo(dest : integer) : Cell
destination:=dest;
return(self).

setTimeStamp1To(value : integer) : Cell
timestamp1:=value;
return(self).

getTimeStamp2() : Integer
return(timstamp2).

setTimeStamp2To(value : integer) : Cell
timestamp2:=value;
return(self).

setTotalDelayTo(value : integer) : Cell
totaldelay:=value;
return(self).

readPath() : integer
return(pathbuffer read()).

• SimulationSettings
In order to obtain correct/new simulation settings the Controller/Inspector sends a data-object to all objects connected to the control channel (see also paragraph 2.6). This data object is called SimulationSettings. This SimulationSettings object contains the simulation settings for the generator, switch and receiver. Each of this process-objects reads its own settings from the SimulationSettings object. The advantage of this mechanism is having one object containing all the simulation settings. This object is broadcasted once to all “interested” objects listening to the control messages.
The Controller/Inspector object uses the set<Simulation-Parameter> methods from the SimulationSettings object and the Generator-, Switch-, Receiver-objects use the get<Simulation-Parameter> methods. New parameters can easily be added to the simulator; just add a set<> get<> method to the SimulationSettings object.

N.B.
The method set/getRunNr() is used as parameter in writing simulation results to disk. Each simulation is performed as a sequence of different simulation runs. Each run delivers its own
results (which are stored to disk). To distinguish results of different runs a run-number is used in the file-name before writing result-data to disk.

**POOSL description:**

```plaintext
data class SimulationSettings

instance variables
load: Object; burstsize: Object; outputbuffersize: Object;
nrofdestinations: Object; runnumber: Object

instance methods
getBurstSize() : Integer
return (burstsize).

getOutputBufferSize() : Integer
return (outputbuffersize).

setNumberOfDestinations(newnrofdest : Integer) :
SimulationSettings
nrofdestinations:=newnrofdest;
return (self).

getLoad() : Integer
return (load).

setRunNr(runnr : Integer) : SimulationSettings
runnumber:=runnr;
return (self).

getNumberOfDestinations() : Integer
return (nrofdestinations).

setBurstSize(newburstsize : Integer) : SimulationSettings
burstsize:=newburstsize;
return (self).

setOutputBufferSize(newobufsize : Integer) : SimulationSettings
outputbuffersize:=newobufsize;
return (self).

setLoad(newload : Integer) : SimulationSettings
load:=newload;
return (self).

getRunNr() : Integer
return (runnumber).
```

- **FIFO buffer**
  As said before all switches must contain buffers to temporarily hold cells. The size and position of these buffers is determined by the switch architecture. The type of buffer (e.g. FIFO, LIFO, LIFO) used to build a switch must also be chosen. Taking cell-delay in account FIFO (First In First Out) buffers are the most fair. A switch constraint is the cell sequence, which must be the same on switch-output as switch-input. This constraint is fullfilled by
using FIFO buffers. Cell-delay and cell-sequence aspects together resulted in using FIFO buffers for the switches.

**POOSL description:**

data class FIFOBuffer

instance variables
fifoDepth: Object; firstLink: Object; lastLink: Object;
maxElements: Object

instance methods
isEmpty: Boolean
  return fifoDepth=0.

getNrOfElements: integer
  return fifoDepth.

write(anElement: Object): FIFOBuffer
  | aLink: FIFOLink |
  if fifoDepth=0 then
    firstLink:=new(FIFOLink) setElement(anElement);
    lastLink:=firstLink
  else
    aLink:=new(FIFOLink) setElement(anElement)
    setNextLink(firstLink);
    firstLink setPreviousLink(aLink);
    firstLink:=aLink
  fi;
  fifoDepth:=fifoDepth+1;
  return self.

read: Object
  | aLink: FIFOLink |
  if fifoDepth=0 then self error() fi;
  aLink:=lastLink;
  if fifoDepth>1 then
    lastLink:=lastLink previousLink;
    lastLink setNextLink(nil)
  else
    firstLink:=nil;
    lastLink:=nil;
  fi;
  fifoDepth:=fifoDepth-1;
  return aLink element.

isNotEmpty: Boolean
  return fifoDepth>0.

clear(size : integer): FIFOBuffer
  fifoDepth:=0;
  maxElements:=size;
  firstLink:=nil;
  lastLink:=nil;
  return self.
RandomIntGenerator
The data-class library coming with the SHE simulation tool contains a RandomGenerator which generates real numbers between 0 and 1 (uniform distributed). This RandomGenerator is used for performing statistical operations e.g. to determine the state-transitions in the CellGenerator object. In the lines below the reason for making a randomgenerator object producing integer numbers is given.
During simulation each cell/burst has its own destination. This destination is determined by the Generator and given to the cell/burst leaving the generator. In case of N destinations the probability that a cell/burst gets destination p (1 <= p <= N) is 1/N. To determine the destination for cells/bursts the RandomIntGenerator object is developed. When calling the getRandomIntGenerator(N) method the RandomIntGenerator object responses with a integer, number between 1..N (where each number in 1..N occurs with same probability). Internally this RandomIntGenerator object uses the RandomGenerator object, but this is not of interest for the users of the RandomIntGenerator object methods.

POOSL description:
data class RandomIntGenerator

instance variables
randomizer: Object

instance methods
getRandomIntNumber(interval : integer) : integer
return ((randomizer random())*interval+0.5) round().

initRandomIntGenerator(): RandomN
randomizer:=new(RandomGenerator);
return(self).
Appendix III  Message Flow and Instance Structure Diagrams

This appendix contains:

- Message Flow Diagram for a 2m2 Switch
- Instance Structure Diagram for a 2m2 Switch
- Message Flow Diagram for a 4m4 Switch Fabric
- Instance Structure Diagram for a 4m4 Switch Fabric
2*2 Switch Simulator, Message Flow Diagram

- **Controller/Inspector**
- **Generator_1**
  - route(cell)
- **Generator_2**
  - route(cell)
- **StageAddresser**
  - route(cell)
- **Switch**
  - route(cell)
- **Receiver_1**
  - route(cell)
- **Inspector**
  - control(settings)
- **Inspecreceiver(index)**
  - control(settings)
- **control(settings)**
- **control(settings)**
- **control(settings)**
- **control(settings)**
- **control(settings)**
- **control(settings)**
- **route(cell)**
- **route(cell)**
- **route(cell)**
- **route(cell)**
2*2 Switch Simulator, Instance Structure Diagram

Controller/Inspector

Generator_1

StageAddresser

Switch

Controller

Inspector

Generator_2

StageAddresser

Switch

Controller

Inspector

Receiver_1

Receiver_1

Receiver_1

Receiver_1

Controller

Inspector

Generator_1

StageAddresser

Switch

Controller

Inspector

Generator_2

StageAddresser

Switch

Controller

Inspector

Receiver_1

Receiver_1

Receiver_1

Controller

Inspector

Generator_1

StageAddresser

Switch

Controller

Inspector

Generator_2

StageAddresser

Switch

Controller

Inspector

Receiver_1

Receiver_1

Receiver_1

Controller

Inspector
4*4 Switch Simulator, Message Flow Diagram
Appendix IV  Drawing Screen

Simulation Drawing 2m2 Switch

Simulation Drawing 4m4 Switch
Appendix V Simulation Results

See next page for correct interpretation of the graphs showing simulation results.

This appendix contains the following measured simulation results:

- cell delay probability using a 2m2 Switch, Uniform Traffic, 40,000 cells per load
- cell delay probability using a 2m2 Switch, Bursty Traffic, 40,000 cells per load
- cell delay probability using a 2m2 Switch, Bursty Traffic, 100,000 cells per load
- switch bufferlength probability using a 2m2 Switch, Uniform Traffic, 40,000 cells per load
- switch bufferlength probability using a 2m2 Switch, Bursty Traffic, 40,000 cells per load
- switch bufferlength probability using a 2m2 Switch, Bursty Traffic, 100,000 cells per load
- cell delay probability using a 4m4 Switch Fabric, Uniform Traffic, 40,000 cells per load
- cell delay probability using a 4m4 Switch Fabric, Bursty Traffic, 40,000 cells per load
- first stage switch bufferlength probability using a 4m4 Switch Fabric, Uniform Traffic, 40,000 cells per load
- second stage switch bufferlength probability using a 4m4 Switch Fabric, Uniform Traffic, 40,000 cells per load
- first stage switch bufferlength probability using a 4m4 Switch Fabric, Bursty Traffic, 40,000 cells per load
- second stage switch bufferlength probability using a 4m4 Switch Fabric, Bursty Traffic, 40,000 cells per load
Interpretation of simulation results

This unnumbered paragraph is written for correct interpretation of simulation results displayed graphically on the following pages.

On each graph 5 lines are displayed. Each line corresponds to simulation results coming from simulations using a load of 0.2, 0.4, 0.6, 0.8 or 0.9. What load is used for certain line is identified in the graph on the corresponding line. Also two different traffic types are used to perform simulations: uniform and bursty traffic. When using bursty traffic the average burst length was set to 10 (this burst length followed a geometric distribution).

There are two different graphs: one showing results coming from receivers (measure cell delay) and one showing results coming from switches (measure buffer length).

Cell delay graphs.
As said before these results come from the receivers. The receivers inspect each incoming cell for their delay and store this information in a table. After certain time this table is written to disk in a simple text file. The lines in this text file have the following meaning:

0 <total number of inspected cells>
1 <amount of cells having a delay of 0>
1 <amount of cells having a delay of 1>
2 <amount of cells having a delay of 2>
3 <amount of cells having a delay of 3>
...

From this table the probability graphs for cell delay were constructed. They horizontal show a cell delay of x and vertically the chance for cells of having a delay equal or larger than x.

Buffer length graphs.
These results come from the switches. The switches inspect each 'switch period' (see 5.4) their buffers for their length and store this information in a table. After certain time this table is written to disk in a simple text file. The lines in this text file have the following meaning:

0 <total number of taken buffer inspections>
1 <amount of inspections where buffer length was 0>
1 <amount of inspections where buffer length was 1>
2 <amount of inspections where buffer length was 2>
3 <amount of inspections where buffer length was 3>
...

From this table the probability graphs for switch buffer length were constructed. They horizontal show a buffer length of x and vertically the chance for a switch buffer of having a length equal or larger than x.

For (multistage) Packet Switch Fabric model simulations two graphs are constructed: one displaying the buffer length probability for the first stage 2n2 Switch buffer length and one displaying the buffer length probability for the second stage 2n2 Switch buffer length.
1 Cell delay probability using a 2m2 Switch, Uniform Traffic, 40,000 cells per load
2 Cell delay probability using a 2m2 Switch, Bursty Traffic, 40,000 cells per load
3 Cell delay probability using a 2m2 Switch, Bursty Traffic, 100,000 cells per load
4 Switch buffer length probability using a 2m2 Switch, Uniform Traffic, 40,000 cells per load
5 Switch buffer length probability using a 2m2 Switch, Bursty Traffic, 40,000 cells per load
7 Cell delay probability using a 4m4 Switch Fabric, Uniform Traffic, 40,000 cells per load
8 Cell delay probability using a 4m4 Switch Fabric, Bursty Traffic, 40,000 cells per load
9 First stage switch buffer length probability using a 4m4 Switch Fabric, Uniform Traffic, 40,000 cells per load
10 Second stage switch buffer length probability using a 4m4 Switch Fabric, Uniform Traffic, 40,000 cells per load
First stage switch buffer length probability using a 4m4 Switch Fabric, Bursty Traffic, 40,000 cells/load.
12 Second stage switch buffer length probability using a 4m4 Switch Fabric, Bursty Traffic, 40,000 cells per load