DISCRETE-EVENT SIMULATION FOR TRIGGER AND DATA ACQUISITION SYSTEMS IN HIGH-ENERGY PHYSICS

Ralf Spiwoks
CERN, Geneva, Switzerland

Abstract
Simulation is a widely used method to analyse and design complex systems. It is applied where the complexity of a system inhibits a closed-form description or where the cost of experiments or of prototypes inhibits measurements. Simulation is based on an abstract model of a real system described in terms of objects and their behaviour. In discrete-event simulation the objects' behaviour is expressed in terms of state changes which can occur only at discrete events in time. This method is very suitable for computers and a wide variety of programming languages for this purpose is available. As an example of such a language, MODSIM will be described in some detail. The design of trigger and data acquisition systems for future experiments in high-energy physics is used as an example application of discrete-event simulation.

1 INTRODUCTION

Simulation is the imitation of a real-world system. It is a method which analyses a real-world system by creating an artificial history of that system and by asking questions of the type "what would happen if?".

Simulation, as a method, is widely used to understand systems in a variety of fields. It is a powerful technique because it is based on numerical methods and can be implemented on computers. With the ever-growing performance of computers, simulation is easier to apply and finds many more applications. Simulation is also a fashionable method which relates to some hot topics in computing science such as virtual reality. It is used to recreate the real world with digital methods.

Simulation is in particular useful

- to give insight into a system: it helps to understand and visualize the behaviour;
- to give numbers: it can be used to evaluate the capacity of a system and to find its bottlenecks;
- to enhance reliability: failures of the system and strategies to avoid failures, as well as strategies to recover from failures can be simulated;
- to make decisions: different parameters and policies of the system can be compared.

Simulation is used for analysis *and* design of complex systems. Simulation is in particular more powerful than other methods if the system under investigation is complex and has many active elements with different and sometimes concurrent activities. Simulation has to be used if experiments or prototypes are excluded because they would be too expensive, too dangerous or even impossible. Simulation can be used without affecting or disturbing the real-world system.

Simulation finds applications in many fields:

- in science: chemical reactions, behaviour of molecules or particle collisions, etc.
- in industry: material handling, assembly operations and inventory systems, etc.
- in business: stock market, insurance companies and project planning, etc.
- in public services: health care, transportation systems, telephone switchboards, etc.
- in computer systems: performance prediction and optimization, computer network evaluation, etc.
- and many more.

2 PRINCIPLES OF SIMULATION

2.1 Definition

Simulation is based on an abstract model which in turn is based on a real-world system.

A *system* is a set of entities, their relationships, attributes and activities. A system is embedded in an environment with which it exchanges some input and output, as shown in Fig. 1. The boundaries of the system are an important part of its definition. Every system has a structure, behaviour and input and output.

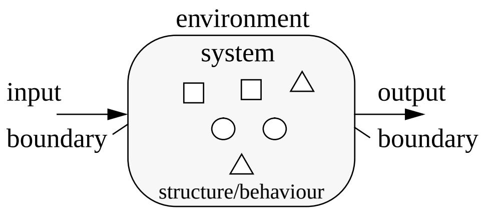

Fig. 1 Model of a System

Modelling is a very human activity. A *model* is the representation of a system. This representation maps the structure and behaviour of the real-world system to a logical structure and behaviour in the model. The representation can be

- physical, e.g. a scale model of a real-world system;
- logical, e.g. a textual description of the system;
- symbolic, e.g. using graphical symbols and diagrams to represent the system;
- mathematical, e.g. using mathematical expressions to describe system entities and behaviour (this is a special form of a symbolic representation).

Models can be classified as static or dynamic. A *static* model does not change over time or is only taken at one point in time. *Dynamic* models describe the evolution of the system over time. Models can be continuous or discrete. In a *discrete* model the attributes are described with discrete numbers while in a *continuous* model the attributes can take any arbitrary values. Models can be deterministic or stochastic. In a *deterministic* model the behaviour and attributes are well known and follow certain explicit rules. In a *stochastic* model the behaviour and attributes may vary according to some random processes. Some models are actually mixtures of the classifications mentioned.

A model also contains performance indices which are used to evaluate the system. These indices allow to “measure” the performance of the system without affecting the actual system. Typical performance indices are average values, minimum/maximum values or probability distributions of values.

Simulation is the execution of a model assuming a certain initial condition and a certain input and output, and calculating the performance indices. The model is taken and exposed to some input and output provided by the user. The behaviour described in the model is executed step by step and the performance indices are calculated. In that sense a simulation can be “run” and make “measurements”.

2.2 Simulation Cycle

When using simulation to analyse or design a system a certain cycle is followed:

1. *System analysis:*

   In the first step, the system has to be analysed. The boundaries between system and environment must be drawn and the entities of the system identified. The relationships of the entities, their attributes and behaviour have to be understood. The input and output of the system have to be identified. Statistical methods of data collection might be used to analyse the input and output.

2. *Modelling:*

   When the system has been analysed it can be modelled. The entities and their relations, attributes and behaviour have to be represented in some form. The level of detail of that representation has to be defined. The input and output can be modelled using probability functions. The aim of the simulation must be defined and the performance indices be introduced in the model. If necessary, the modelling step might be iterated when the system analysis reveals new aspects of the system or different aims of the simulation are formulated.

3. *Simulation:*

   The model has to be translated into a form which can be used to "run" a simulation. Usually a simulation language is used for this purpose. Computer programs are generated to execute the model. The calculation of the performance indices has to be included, initial configurations and input and output to be provided. The program can then be run and produce some results for the performance indices.

4. *Verification:*

   In this step the consistency between the simulation and the model is verified. Does the simulation agree with the model? It must be checked if the simulation contains all the necessary elements to deliver the requested performance indices and if they are meaningful. If necessary, the simulation has to be revised to be in agreement with the model.

5. *Validation:*

   When the simulation is known to be consistent with the model it has to be validated against the system under study. Does the model agree with the system? This step can be carried out comparing the simulation with experiments or prototypes. If necessary, the model has to be revised and a new simulation to be carried out. The level of detail which was used to model the system can be increased or reduced to give better agreement between the model and the system.

6. *Simulation analysis:*

   Once the simulation is validated it can be used to produce the expected performance indices. Several simulation runs can be used to understand and evaluate the system, to describe its performance and to find its critical points. The simulation can be used to investigate the system's behaviour for different parameters or for failures. In the end, different simulation runs can be used to compare different configurations, parameters and policies in order to decide on design issues.

The simulation cycle might be repeated for cases in which the simulation leads to a new analysis of the system. Often the system analysis and modelling phases are carried out in parallel. Sometimes the modelling and simulation phases merge if the tools for modelling can automatically create a simulation program from the model. The validation sometimes can only cover part of the model and the simulation results depend on assumptions made in the model.

2.3 Types of Simulation

There are different types of simulation which can be classified as follows. The actual type depends on the system and the model which has been made of it.

In a physical simulation a physical model is made of the system under study and exposed to real-world input and output. In a mathematical simulation the model is described in mathematical terms. The mathematical simulation can then be carried out by finding an analytical solution or a numerical solution. In the last case, nearly always a computer is used to find the solution. The word "simulation" is applied, in a narrower sense, only to the mathematical simulation based on numerical methods carried out on a computer. It will be used in that sense from now on.

Different examples of simulation can be listed. This list is not exhaustive and only shows the most important types of simulation:

- *Differential or difference equations:*

  Some models can be represented in the form of (partial) differential or difference equations. Their solution might, however, not be available analytically and numerical methods have to be applied to simulate the system's behaviour.

- *Monte-Carlo simulation:*

  In a Monte-Carlo simulation the emphasis is on the stochastic aspect. Time is usually not important and the system is studied as before and after some random process. Usually this will be repeated many times to get statistical significance of the performance indices observed.

- *Cellular automata:*

  In cellular automata the system is described in discrete space. The model consists of many cells which have neighbours in one or more dimensions. The behaviour of each cell is usually described with respect to its immediately neighbouring cells.

- *Discrete-event simulation:*

  In discrete-event simulation [1-5] a system is described in discrete time. The model contains elements which have a state. The state can change at discrete moments in time.

Often a simulation is a mixture of types. Discrete-event simulation might (and usually does) contain elements of a Monte-Carlo simulation. Sometimes discrete and continuous elements are mixed: e.g. differential equations could be used to describe an attribute of an entity and when a certain value of that attribute is exceeded a state change occurs.

2.4 Discrete-event Simulation

2.4.1 Queueing Models

Discrete-event simulation is often applied to describe a model of a system in which there are entities of traffic flowing through the system. These entities which can be parts, products, vehicles, people, jobs or work orders need certain resources. These resources can be machines, handling equipment, space, manpower or processors. In such a typical model the behaviour of the system is expressed as a network of queues in front of each resource which the entities have to pass.

The most simple queueing model is a single queue where entities arrive and queue up in front of a server (= resource). Examples of such a basic queue are waiting queues of clients in a bank, cars in front of a petrol station or processes on a processor. The input parameters of such a queueing model are described by

- the interarrival time;
- the servicing time;
- the number of servers, in case there are more than one;
- the queue length, in case there is only a limited number of queue places;
- the customer population, in case there is a finite population and customers can only re-enter in the queue after they have been serviced.

The way customers and servers are chosen is called a policy and is part of the model. In particular, the policy to choose which customer from a queue of several waiting customers is serviced next, is called “queueing policy”. The policy to choose the next server to service a customer, if there are several servers available, is called “servicing policy”. Some policies are frequently used:

- first-in-first-out, e.g. the customers are served in the order they arrive;
- round-robin, e.g. the servers are chosen in a fixed order;
- random, e.g. the customers are chosen at random;
- prioritized, where the priority is either something staticly assigned to each customer/server or is something that can change over time. A typical example for a static priority is the shortest-job-first policy. In a scheme with dynamic priority the priority might e.g. be recalculated depending on how long one job has already been waiting.

The typical performance indices for the queueing model are

- for a customer: the probability to wait, the time spent waiting and the total time spent in the system;
- for a server: the time spent idle and the load which is the fraction of time the server is busy;
- for the queue: the average number of customers in the queue and the fraction of customers which had to be turned away because a finite queue was full.

There is a general mathematical theory for queueing models which is called “queueing theory” [6]. This theory tries to calculate the performance indices from first principles assuming that the system and its input and output can be described with the help of stochastic processes. For some queueing problems there exists an analytical solution. Of course, in such case a simulation does not need to be run. In most cases, however, with large networks of queues or complicated policies there is no other solution than simulation.

2.4.2 State and Event

Discrete-event simulation is a simulation where the system is modelled in terms of entities which have states. The *state* is a set of variables which describe the entity at any given time. The state changes can only occur at discrete moments in time. This is called an “*event*”. Between two consecutive events the entity does not change its state. The state of the whole system is described as the superposition of the states of all entities in the system and simulation is carried out by advancing simulation time from one event to the next.

In the example of a single-server queue the state of the system can be described by the total number of customers in the system. The arrival of a customer is an event which increments the number by one. The departure of a customer is another event which decrements the number by one.

Often the state of an entity can be described among a finite set of states. The entity following such a model is called a finite-state machine. When an event occurs a transition from one state into another is accomplished. The whole system can be described as a set of finite-state automata which communicate with each other in terms of events causing state transitions.

In the example of the single-server queue the server can be in state “idle” or “busy”. The events “customer arrival” and “customer departure” lead to state transitions depending on the queue status, as shown in Fig. 2. Some of the transitions have the same initial and final state.

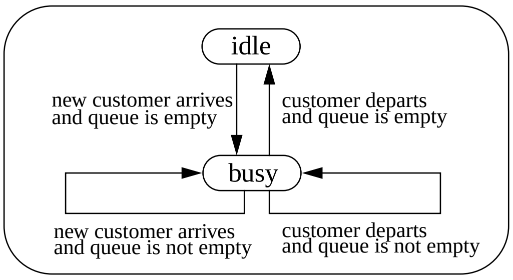
idle
new customer arrives and queue is empty
customer departs and queue is empty
busy
new customer arrives and queue is not empty
customer departs and queue is not empty

Fig. 2 Finite-state Machine of a Server in a Single-server Queue

Since simulation consists of advancing simulation time from one event to the next it should be noted that the simulation time is not the same as the real-world time. The advancing of simulation time usually goes in one step while carrying out the state changes related with each event is instantaneous in the real world but takes some time in the simulation program. This is a big advantage of discrete-event simulation as it can expand or compress time as needed to understand the behaviour of the system.

2.4.3 Activity and Process

Describing a complex system with many active elements in terms of individual events might be tedious. Therefore events are usually grouped to activities and processes. An *activity* is a duration of time and has an event to begin and an event to terminate. Between these two events an activity is supposed to be carried out. An activity usually can be unconditional, that is it takes a fixed amount of time. Or it might be conditional in which case it takes an unspecified period of time and the termination of the activity depends on the state of other entities.

A *process* is a time-ordered set of events and activities which describe the “life” of an entity, as shown in Fig. 3. Processes can have several concurrent activities and might depend on other processes.

Processes are a higher level of description than events and can often map more easily on the entities in the system.

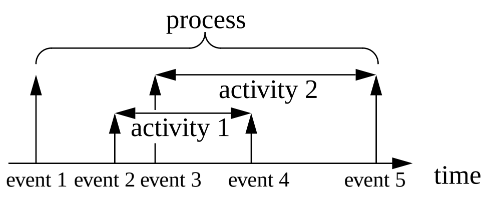

Fig. 3 Relation of Events, Activities and Process

E.g. a customer in a shop who chooses some goods, waits in front of the desk and pays can be described as a process which consists of several activities.

2.5 Examples

Other examples of models which can be used for discrete-event simulation are:

- *A computer can be described as a system with a set of jobs which wait for access to the processor and once they have run some time on the processor might have to wait for access to the disk or other I/O devices.*
- *An assembly line consists of several machines which produce and assemble parts in series. In front of each assembly station some waiting time must be taken into account. The parts following the line are the customers, the machines are the servers.*
- *A traffic intersection is a complicated network of queues which depend on each other. The resource is the space to cross the intersection safely. The policy for deciding which car can cross the intersection might be complicated taking into account the different rate of car flows in the different directions.*

3 IMPLEMENTATION OF DISCRETE-EVENT SIMULATION

3.1 World views

The system's behaviour for a discrete-event simulation is expressed in terms of events, activities and processes. According to which one of these the emphasis is put on there are three different paradigms of implementation: the event scheduling, the activity scanning and the process interaction. Though they all lead to different implementations, the application, or the point of view of a user, will be unchanged.

3.1.1 Event Scheduling

In the event scheduling paradigm the emphasis is on events, or state changes of the system. The simulation program is written by describing all possible events and what has to be done when an event occurs. The simulation program keeps a time-ordered list of events and runs through the following algorithm: first, the next event is read from the list and the action belonging to it executed. Then the simulation time is advanced to the next event in the list. The simulation keeps running by the fact that one action carried out for an event might include the scheduling of other events.

An example for the event scheduling paradigm is shown in Fig. 4 containing pseudo-code for the simulation of the single-server queue. The EVENT *statement describes the events which are the arrival of a customer and the departure of a customer. The additional event of generating a customer is used to keep the simulation going. The* SCHEDULE *statement describes the scheduling of an event on the list of future events at a given time. The simulation program has to run through the list of future events carrying out the actions associated with each event which include the scheduling of further events.*

In this paradigm the simulation program contains an event list and a time variable. The mechanism to schedule events is defined as well as a mechanism to advance time to the next event in the list.

```
EVENT ArrivalOfCustomer
    if(Server == idle)
        Server = busy
        SCHEDULE($t_1$,DepartureOfCustomer)
    else
        putCustomerIntoQueue
EVENT DepartureOfCustomer
    if(Queue == empty)
        Server = idle
    else
        getCustomerFromQueue
        Server = busy
        SCHEDULE($t_2$,DepartureOfCustomer)
EVENT GenerateCustomer
    SCHEDULE($t_3$,ArrivalOfCustomer)
    SCHEDULE($t_4$,GenerateCustomer)
```

Fig. 4 Example of the Single-server Queue in the Event Scheduling Paradigm

3.1.2 Activity Scanning

If the system's behaviour is described in terms of activities another paradigm for implementation can be used: all possible activities and their conditions are described and held in a list. The conditions being either a specific period of time to pass or some dependence on other activities. The simulation program then checks all activity conditions and executes the activities which can be carried out. Since the execution of some activities can produce conditions for other activities to start, the scanning process has to be repeated until there are no more activities which can be carried out. Only then can the time be advanced to the next activity.

An example for the activity scanning paradigm is shown in Fig. 5 containing pseudo-code for the simulation of the single-server queue. The ACTIVITY *statement describes the activity of which there is only one: the servicing of a customer. The additional activity of generating a customer is used to keep the simulation going. The* EXECUTE *statement sets the activity's internal due time and event type to the given values. These can be used in the simulation program which has to scan the conditions of the two activities and to advance the time once no more activities can be started or terminated.*

```
ACTIVITY ServicingCustomer
    if(event_type == Arrival)
        if(Server == idle)
            Server = busy
            EXECUTE($t_1$,Departure)
        else
            putCustomerIntoQueue
    else
        if(Queue == empty)
            Server = idle
        else
            getCustomerFromQueue
            Server = busy
            EXECUTE($t_2$,Departure)
ACTIVITY GeneratingCustomer
    if(event_type == Arrival)
        EXECUTE($t_3$,Arrival)
```

Fig. 5 Example of the Single-server Queue in the Activity Scanning Paradigm

In this paradigm there is no explicit scheduling of events and the calculation of a time order for the events is replaced by the scanning of all conditions of the activities. The simulation program also needs a list, which now is the list of all activities, and also has a time variable. The activity scanning can have a better performance compared to other paradigms in some cases. Nevertheless, there are only a few languages available which use this paradigm. It is relatively unpopular compared to the other two.

3.1.3 Process Interaction

In the process interaction the system's behaviour is described in processes. Every process in the model is translated into an independent process in the simulation. The processes can be active or suspended in which case they either wait a specified (or scheduled) time or wait on some condition depending on other processes. The simulation finishes with the imminent process until this one suspends and then checks all processes which are waiting on some condition starting the one whose condition is true. If there are no more processes which can be started immediately the time is advanced to the next event and the process to which it belongs is activated.

An example for the process interaction paradigm is shown in Fig. 6 containing pseudo-code for the simulation of the single-server queue. The PROCESS *statement describes the processes of which there is only one: the customer. The additional process of a customer generator is used to keep the simulation going. The* WAIT *statements suspends the process for a specific period of time or until a certain condition becomes true. The simulation program schedules the processes and activates them whenever necessary.*

```
PROCESS Customer
        WAIT FOR (Server==idle)
        Server = busy
        WAIT DURATION t1
        Server = idle
        deleteProcess(SELF)
PROCESS CustomerGenerator
        while(TIME < TIME_END_OF_SIMULATION)
                WAIT DURATION t2
                generateNewProcess(Customer)
```

Fig. 6 Example of the Single-server Queue in the Process Interaction Paradigm

In this paradigm the simulation program keeps a list of processes and their state. The list might be divided into one part for the processes scheduled at a given time and those which are waiting on some conditions. The simulation program has a time variable and a scheduling mechanism. Often the simulation program can be based on already existing packages for the scheduling of concurrent processes.

The process interaction paradigm can be used to run complex simulation programs in a parallel way on several processors. Each process can, in principle, have its own future event list and time variable, and run on a different processor. The processes exchange messages to tell each other what has to be done. Some mechanism has to ensure that causality is not violated, i.e. that a process does not receive a message about something that happened at a time earlier than its local time. Several algorithms to avoid this kind of problem have been discussed [7].

3.2 Simulation Program

Though there are different paradigms to implement a simulation program they all require a list of future events, activities or processes and a time variable. The list needs a mechanism to schedule items on it and the time variable corresponds to a mechanism to advance time to the time of the next event. This can also be done in fixed steps to make the simulation faster or because there is already a specific periodic time in the model.

Other than these two elements the simulation program also needs some mechanism to collect statistics for the performance indices. Usually some histogramming mechanism is available to give average values and distributions. Another important element are the random variables which are taken from random number generators [1],[2],[8]. These provide the element of randomness for attributes, waiting times, and so on. Some frequently used random number probability density functions include the uniform, the exponential and the normal distribution.

In addition, a simulation program needs some input/output functions to exchange parameters with the user and to print reports for the performance indices. Sometimes a graphical user interface can be used for representation of the model, for graphical input/output of parameters, for animation of the system behaviour (changing some parameters on the screen while the simulation runs) and for the presentation of the results. Other utility functions include libraries for list handling or pre-defined simulation entities which can be re-used for the simulation. A simulation program should also include some mechanism to debug and trace the actions it takes, in particular if there are many concurrent actions.

The common structure of a simulation program with all the elements mentioned before then will have to run through the following steps:

1. *Initialization:*

   At initialization all entities have to be declared in terms of events, activities or processes depending on which paradigm is chosen. The first events, activities or processes have to be scheduled.

2. *Simulation:*

   In the main part the simulation program then starts the time advancing mechanism and keeps going through the list of events, activities or processes. At the same time the statistics are collected for the performance indices.

   The simulation will stop either when there is nothing more to do, a certain simulation time has elapsed or a certain number of entities have passed through the system.

3. *Report:*

   In the end the simulation program should print out the statistics collected for the performance parameters. Some histograms may be plotted and the simulation be analysed.

3.3 Languages and Tools

There are many languages and tools available for discrete-event simulation (and other types of simulation). They exist for general and specific problems and for all kinds of platforms: PCs, workstations and main frames. There are some overviews of the existing languages and tools available on the WWW which are updated regularly [9]. Different classes of languages and tools can be classified as follows:

- *High-level languages*

  Any high-level general-purpose language can be used to write a simulation program. FORTRAN, C, C++ and PASCAL are some examples. When using a general-purpose language the user has to program the model *and* all the simulation specific part.

- *Simulation languages*

  Simulation languages are specifically designed for discrete-event simulation, like e.g. C++SIM, GPSS, SIMSCRIPT, SIMULA, MODSIM and so on (see [9]). In these languages the user has to write the model and the languages provide all the syntactic elements for the simulation as well as usually some library of simulation entities ready to use.

- *Simulation tools*

  Simulation tools exist for discrete-event simulation of specific problems, like for manufacturing systems, health care, networking systems, scheduling problems, and so on. In these tools, the model and the simulation are provided, so that no programming needs to be done. The user provides input parameters to the model and then observes the results delivered by the program which usually is called a “simulator”.

When choosing a language or tool for a problem at hand the user should consider the following points:

- *Suitability*: is the language/tool suitable for the problem? Are there other methods like analytical solutions, experiments or prototypes?
- *Completeness*: does the language/tool contain the right random number generators? Does it cover the

statistics collection for the performance indices? Does it have a graphical user interface and animation? Can it be interfaced with spreadsheet applications, computer aided design tools, graphical user interfaces and virtual reality programs?

- *Programming environment*: does the language/tool use the right editors, compilers and debuggers?
- *Resources*: what resources does the language/tool require? What does it cost? What hardware does it require and what execution speed does it provide? Does it require training and what is the learning curve?

3.4 MODSIM

MODSIM is a commercial language for discrete-event simulation [10]. It is based on *MODULA2* and has a block structure. It is object-oriented and gives encapsulation of data, polymorphism and inheritance. It accomplishes discrete-event simulation in the process interaction paradigm with each process being described as one object.

MODSIM is widely used in high energy physics. It is a high-level simulation language in the sense that it is not restricted to a specific problem. Therefore it can easily be used to describe data flow systems from a high-level point of view (see Sect. 5).

3.4.1 Language Features

In *MODSIM* all statements are grouped in blocks and each block has a begin and end statement. The whole software for a model can be divided into modules: the *MAIN* modules contains the behaviour of the model as a whole. The *DEFINITION* and *IMPLEMENTATION* modules contain the behaviour of the individual objects (which are similar to the .h or .c files in C, respectively). With the help of the *IMPORT* statement the objects can be used in other modules (like the #include statement in C).

MODSIM code is translated into C and can easily include other code already available in C. *MODSIM* further has a utility to re-compile only those modules of a simulation program which need re-compilation (much like the “make” utility in UNIX). Furthermore it contains a browser taking into account the structure in terms of modules, objects and their inheritance trees. A debugger can be used to trace actions of objects and to check the list of pending objects.

3.4.2 Object-oriented Features

Like in any object-oriented language *MODSIM* groups attributes and methods to objects to make logical units. This is called “encapsulation”. Each process of the simulation is expressed as an object in the simulation program and has attributes and methods.

The objects are usually declared in separate *DEFINITION* modules and the implementation of the objects’ methods in corresponding *IMPLEMENTATION* modules. The objects are dynamically instantiated using the *NEW* statement, and deleted using the *DISPOSE* statement. The methods of an object are invoked in *ASK* or *TELL* statements, e.g. *ASK ActualObject TO DoSomething,* where *ActualObject* is an instantiation of an object and *DoSomething* is a method defined for that object.

Inheritance can be used to reflect logical relations between objects of the type “object B is a kind of object A”. A derived object inherits all the attributes and methods from the parent object, can add other attributes and methods and override the inherited methods to give them a specific meaning. This is called “polymorphism”.

3.4.3 Simulation Features

The simulation in *MODSIM* is expressed in terms of several syntactic elements like the *TELL* and the *WAIT* statements. The *TELL* statement is used to start a method of another (or the same) object asynchronously. The invoked object will be activated with the specified method while the current method will continue in parallel. The *WAIT* statement is used to either suspend the invoking method for a specified period of time or to start a method of another (or the same) object and to suspend itself until the invoked method has finished. This is a mechanism to synchronize two different methods of two objects or of the same object. Methods invoked with the *ASK* statement are instantaneous and do not spend any simulation time.

The whole simulation can be regarded as a collection of objects with several methods being active at the same time. The synchronization between the different methods of the objects is achieved either by using the *WAIT* statement (as described above), by a dedicated trigger object or by using interrupts. The trigger object is part of the *MODSIM* library and allows arbitrary synchronization between methods. One object uses the *WAIT* statement to invoke the *Fire* method of an instantiation of a trigger object and another object uses the *TELL* statement to invoke the *Trigger* method of the instantiation of the trigger object. When the latter method is invoked the object waiting on the instantiation of the trigger object is activated.

The interrupt procedure allows to interrupt the waiting of an object which was suspended using a *WAIT* statement. The suspended object will be activated and continue its action after a dedicated *INTERRUPT* clause in the *WAIT* statement.

The user has to program the model in terms of objects (= processes) which spend time and interact with each other. This is following the process interaction paradigm. The language provides the necessary syntactic or library elements and takes care of the scheduling of the processes which is completely hidden from the user.

Examples of *MODSIM* code can be found in Subsect. 5.1.3 for the simulation of an event building system.

3.4.4 Library and Utilities

MODSIM has a library which contains many useful objects for simulation. These objects include random number generators, queue objects for all kind of lists and list operations, resource objects and the aforementioned trigger object.

Another important utility in *MODSIM* is the *MONITOR* method. This is a method of an object associated with an attribute of the object. Each time the attribute appears in a left-hand or right-hand statement the *MONITOR* method will be called and can print out a statement, collect some statistics or update a graphics object.

MODSIM also has a rich graphics library which can be used to represent the model, to input/ output parameters, to animate the model while the simulation is running and to present the results at the end of the simulation.

4 TRIGGER AND DATA ACQUISITION SYSTEMS

4.1 Definition

In high-energy physics and other experiments a trigger and data acquisition (TDAQ) system takes the signals of a detector and delivers them to the physicist, usually by recording the data on tape. For more details see e.g. [11].

The TDAQ system has several tasks: it digitises the signals from the detector if necessary, using an analogue-to-digital conversion. It collects the data from several parts of the detector which usually consists of several sub-detectors and modules, and multiplexes these data into one stream of data. The TDAQ system can do zero suppression in which only the data from signals above a certain threshold are kept and may also format the data according to the requirements defined for the analysis step which is usually carried out off-line. The TDAQ system also has to select the useful data and to drop the signals which the physicist is not interested in. This is called the “trigger” of an experiment.

An example architecture of a TDAQ system consists of an analogue-to-digital converter and multiplexer which writes the data from several channels of the detector into one buffer. The trigger either receives directly the signals from the detector or the data from the buffer and decides which data are read out and which are dropped. The read-out task reads the data from the buffer and writes them to a permanent storage. Some control tasks take care that everything works as it should.

A TDAQ system can be regarded as a queueing system in which the buffer is the queue and the trigger and the read-out tasks are servers. In addition, the physics in a detector usually follows some stochastic processes, so that they have random parameters. The trigger rates, i.e. the number of accepted data per time unit, depend on the physics. The processing times for trigger and read-out tasks depend on the physics and the data size (if after zero suppression it varies). A varying data size also influences the transfer times of data between buffer and read-out. Often, TDAQ systems have several data streams in parallel and include several levels of triggers. All in all, TDAQ systems can be described as networks of queues, and discrete-event simulation is an appropriate way to analyse and design them.

4.2 ATLAS Experiment

As an example, the ATLAS TDAQ System will be taken. ATLAS [12] is a future experiment at the LHC at CERN. The LHC will be a proton-proton collider running in the year 2005 and delivering proton-proton collisions every 25 ns at an energy of 14 TeV. This will be the highest energy for proton-proton collisions achieved so far and will give new insight into the world of the smallest particles. It will increase the understanding of the Standard Model which is a model of the world of particles as it is conceived now. It might also lead to new observations and discovery of new particles.

The ATLAS experiment is a general-purpose experiment which tries to exploit all of the new phenomena observable at the LHC. The ATLAS detector is built of several sub-detectors and has altogether 10^8 electronic channels which need to be read out in principle every 25 ns.

The TDAQ system of the ATLAS experiment has to deal with an unprecedented data rate. The current design is based on three levels of triggers and several streams of data in parallel, as schematically shown in Fig. 7. The signals produced by the detector are stored in front-end pipelines while the level-1 trigger finds a decision. If accepted, the signals are read out by the read-out driver and converted to digital data, if not already done before. The data are stored in read-out buffers until the level-2 trigger, which uses part of the data stored in these buffers, makes a decision. When this decision is positive, the data belonging to the same proton-proton collision are assembled from all the read-out buffers and sent to a farm of processors which run the level-3 trigger selection. This way the TDAQ system will be able to select only the interesting data and reduce the data rate to a level which can be written to a storage system for further analysis. For a more detailed description see [12].

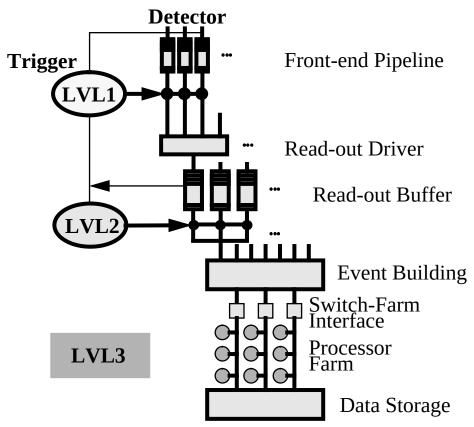

Fig. 7 The Trigger and Data Acquisition System of the ATLAS Experiment

New technologies and new algorithms have to be applied to accomplish the task of the TDAQ system. Simulation is one way to get an understanding of how such a system can be built. Together with prototypes simulation will be used to make evaluations for the design before the TDAQ system will be built.

4.3 Event Building

One part of the ATLAS TDAQ system is the so-called “event-building”[1]. The event building (EB) system assembles the data fragments which come from several sub-detectors and modules of the experiment and which belong to the same collision that happened in the detector. The EB system

1. Sometimes in physics the data from a detector are called an “event”; this should not be confused with the notion of events as developed for the discrete-event simulation. The term “data” will be used for the TDAQ system instead.

collects the data fragments from the different buffers of the modules and sub-detectors and provides fully assembled data in other buffers from which the data can either be used for further triggers or be written to tape. The EB system consists of sources, destinations, an interconnecting network and a data flow manager, see Fig. 8. The sources provide the data fragments, the destinations collect the fully assembled data. The interconnecting network is used to transfer the data between sources and destinations and the data flow manager controls the operation of the event building system and ensures that all data fragments which belong to the same collision go to the same destination.

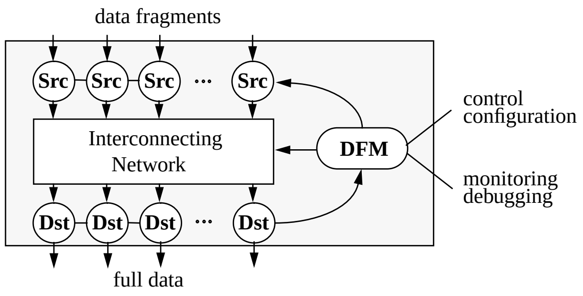

Fig. 8 Model of an Event Building System

The input parameters for the EB system of the ATLAS detector are as follows: about 100 to 200 sources provide data fragments of around 10 kByte each, at a frequency of about 1 kHz. There shall be about 100 destinations to collect fully assembled data. This means that the interconnecting network will have to be capable of connecting about 200 to 300 ports (sources or destinations) and have a total throughput of data of several GByte/s. The data flow manager will have to work at a typical frequency of 1 kHz for all fragments.

The performance indices of an event building system are:
- the throughput of data;
- the frequency at which data fragments can be put into the system;
- the latency, which is the time between the first data fragment arriving at a source and fully assembled data being available at the destination;
- the buffer size necessary to buffer data fragments and fully assembled data without losing any;
- the percentage of data which get lost due to finite buffer sizes.

4.4 High-speed Interconnects

The event building system has strong requirements on the interconnecting network. It cannot be built from conventional network or bus standards but new high-speed interconnects [13] have to be used. There are a few new standards evolving in industry which are promising candidates for implementing the interconnecting network of the ATLAS EB system. They include:
- ATM - the Asynchronous Transfer Mode [14];
- FCS - the Fibre Channel Standard [15];
- HIPPI - the High Performance Parallel Interface [16];
- SCI - the Scalable Coherent Interface [17].

All these standards define data transfers of 10 to 100 MByte/s over twisted-pair cables or optical fibres. They all define point-to-point connections and networks of different topologies, e.g. bridges, switches and rings. The standards are usually based on the notion of “packets” which are the smallest unit of contiguous data which can be transferred in one block. Deciding the way a packet takes from the sender to the receiver is called “routing”. Two different approaches to routing exist: in the packet-oriented approach, each packet is sent and routed through the network separately. The packet contains information on sender and receiver. In the connection-oriented approach, sender and receiver establish a connection, much like in a telephone connection. The packets which do not need to contain routing information can use this connection. For more details see [18].

For building networks of many hundreds of ports, switches are used in all the standards which can in principle be of several types: in a crossbar switch every input port can be connected to every output port on a switch matrix which is controlled according to the requests for transfers. In a hub there is a fast, shared medium over which all the data flow. The performance of that shared medium is usually big enough to provide throughput for several concurrent data flows. In a network the switching is achieved by a network of several small switching elements.

Discrete-event simulation is a method to understand if and which of these new high-speed interconnects could be used for the EB system of the ATLAS experiment.

5 SIMULATION OF TDAQ SYSTEMS

5.1 Simulation of EB Systems for ATLAS

5.1.1 System Analysis

In the previous section (in particular in Subsect. 4.3 and 4.4) the EB system has been defined. The problem of the system has been analysed and possible solutions using high-speed interconnects and fast switches have been proposed. The entities of the system have been identified, as well as the input and output of the system which are the data fragments and the full data. The behaviour of the system is dominated by the high-speed interconnect standard and the control mechanism of the data flow manager.

5.1.2 Modelling

The previous sections already contain elements of a model of the EB system, in particular the high-level model presented in Fig. 8. This model needs refinement in order to be used with discrete-event simulation. Such an EB system based on the HIPPI standard will be used as an example[19].

The model is detailed by the choice of connection-oriented transfers which assume a linear transfer time, i.e. the time for a data fragment to be transferred from source to destination can be calculated by a constant overhead and a size dependent part. The overhead represents mainly the time to establish a connection while the size dependent part is defined by the speed at which data can be transferred once a connection is established. The model of the interconnecting network is that of a crossbar switch which instantaneously establishes the connection from a source to a destination if the destination is not already connected to another source. In the latter case the source will simply wait until the destination is ready. Other details of the model are taken from the ATLAS EB system input parameters, summarized in Subsect. 4.3. The interarrival time is taken to be Poisson distributed with an average of 1 ms which corresponds to an average rate of 1 kHz. The data fragment size is taken to be constant, with 10 kByte.

5.1.3 Simulation

The implementation of the model using *MODSIM* is straight-forward. The sources, the destinations and the data flow manager are implemented as objects which interact according to the function of an EB system, see Fig. 9. The data fragments and full data are implemented as passive objects which are exchanged between the sources and destinations. A data generator object is used to inject the data fragments into the sources. There is no particular object for the crossbar switch. The sources and destinations implement this functionality by the way connections are established between them. The *MODSIM* code for the source and destination objects is shown in Fig. 10 and Fig. 11, respectively.

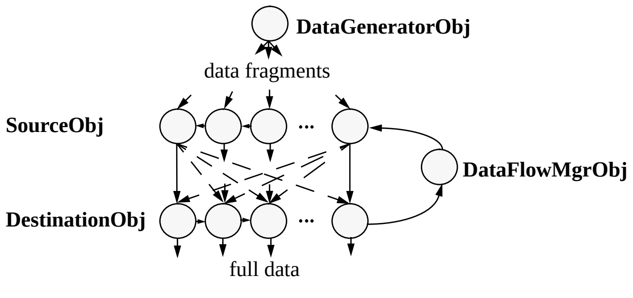

Fig. 9 MODSIM Objects for the Event Building System

```
OBJECT SourceObj;
    TELL METHOD ProcessNewData(newData);
    BEGIN
        IF state = busy
            ASK Buffer TO AddData(newData);
        ELSE
            state := busy;
                {get destination identifier}
            WAIT FOR DFM TO GetDestination(newData);
            END WAIT;
                {send fragment and wait for trigger object}
            TELL Destination TO ProcessNewData(newData);
            WAIT FOR TransferEnd TO Fire;
            END WAIT;
            state := idle;
            CheckDataInBuffer;          {loop on fragments}
        END IF;
    END METHOD;
END OBJECT;
```

Fig. 10 MODSIM Code of EB Source Object

```
OBJECT DestinationObj;
    TELL METHOD ProcessNewData(newData);
    BEGIN
        IF state = busy
            ASK Buffer TO AddData(newData);
        ELSE
            state := busy;
                {calculate transfer time and wait}
            transferTime := newData.Size / speed + overhead;
            WAIT DURATION transferTime;
            END WAIT;
                {synchronize with source: trigger object}
            TELL TransferEnd TO Trigger;
            state := idle;
            CheckDataInBuffer;          {loop on fragments}
        END IF;
    END METHOD;
END OBJECT;
```

Fig. 11 MODSIM Code of EB Destination Object

5.1.4 Verification

The next step in the simulation cycle (see Subsect. 2.2) is the verification. First, it has been tested how many data fragments have to be simulated in order for the system to reach a steady state. This effect which is called “warm-up” is necessary since at the beginning the EB system does not contain any data fragment and only slowly fills up until the performance indices do not vary much except for statistical fluctuation due to the random interarrival time. It can be seen in Fig. 12 that after 30,000 data fragments in the given configuration the system reaches a steady state and the performance indices have a precision of better than 1%.

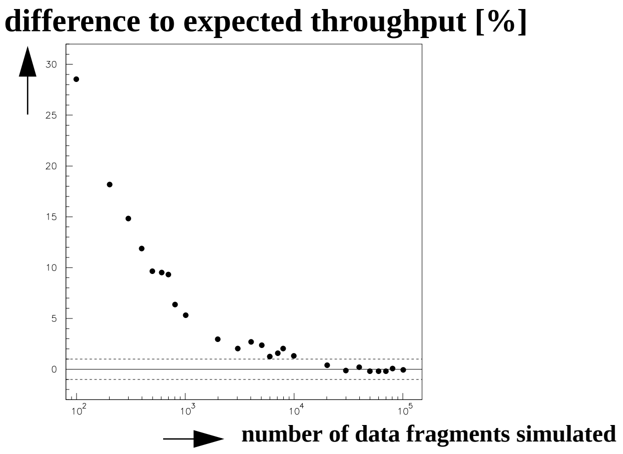

Fig. 12 Verification of Simulation: “Warm-up” Effect

A second consistency check is carried out by comparing the simulation to a theoretical model [19]. Due to the fact that all data fragments have the same size the switch goes into a cyclic mode of operation which is known as the “barrel shifter”. In this mode, which is schematically shown in Fig. 13, the switch goes periodically through a cycle of connection patterns and the performance indices can be calculated from the time for one connection and the interarrival time of data fragments. The simulation showed perfect agreement with the behaviour of the “barrel shifter”.

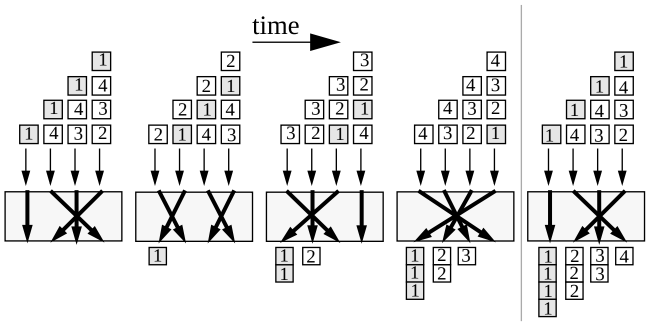

Fig. 13 Cyclic Scheme of a “Barrel Shifter”

5.1.5 *Validation*

After the verification, the next step is to validate the model with a prototype [19]. A VME based system with HIPPI sources and destinations and a HIPPI switch, as shown schematically in Fig. 14, was used to obtain the same kind of performance indices as in the simulation. It was found that the simulation had to be “calibrated” with the prototype measurements: the overhead and the speed of the transfer were measured to be 100 µs or 41.5 MByte/s, respectively. With these parameters used in the simulation a very good agreement between simulation and the prototype measurements could be achieved as can bee seen in Fig. 15.

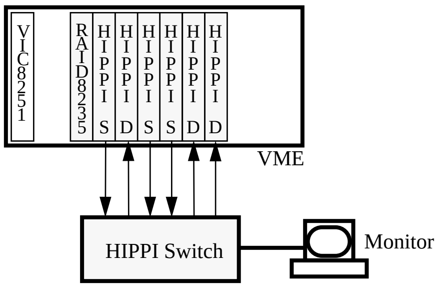

Fig. 14 Setup of the Prototype EB System based on HIPPI

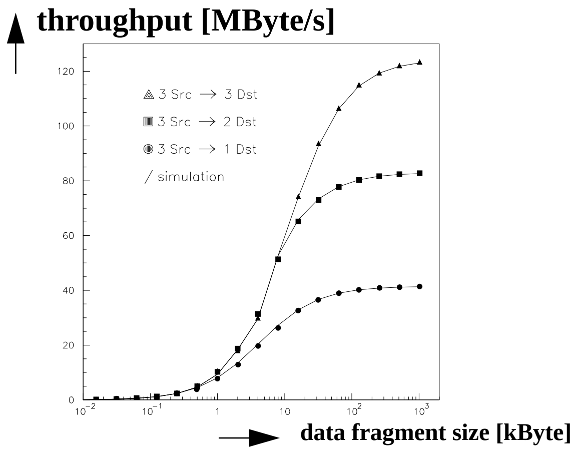

Fig. 15 Comparison of Simulation and Prototype Measurement

5.1.6 Simulation Analysis

After validation, the simulation was used to give performance indices for different configurations of the EB system. Some results include that the EB system with the parameters described above can accept an input frequency of 3.5 kHz of constant-size data fragments, or 3.5 times what is required. However, varying data fragment sizes as obtained from another simulation for the ATLAS detector decreases the performance by a factor of 2. This can be explained by the fact that variations in the data size disturb the “barrel shifter” mode and cause the switch to be used less efficiently. The buffer space required could be estimated to be about 1 MByte at a frequency of 1 kHz in order to lose less than 1 fragment out of 100,000.

The simulation was then used to investigate other specific topics of interest for EB systems [19]:

- *Destination assignment*

  Different strategies for the assignment of the destination by the data flow manager were compared. In the *PUSH* algorithm the destinations are assigned by using a static relation between the sequence number of the data and the destination. In the *PULL* algorithm the destinations initiate the assignment whenever they are ready to receive new data. In the *DFM* algorithm the data flow manager communicates with the sources and destinations and uses lists to assign a destination to the data fragments. The simulation showed that there is no principal difference between the algorithms. However, if the data flow manager is implemented as a specific processor which needs time to assign destinations, this time is limited if the requirements of the EB system are to be met. Simulation can be used to define this limit.

- *Two-stage system*

  Another simulation study was investigating what happened if not one big switch would be used for the interconnecting network but rather a two-stage network built up from smaller switches. It was found that in this case the throughput and the latency are slightly better than using a single switch but that, of course, some buffer and some routing algorithm had to be used between the two stages. The simulation showed, however, that building up the full EB system from several smaller switches is possible.

Discrete-event simulation was used to understand the issues for the ATLAS EB system, in particular the importance of the switch and the control algorithms. The results of the simulation are used to design a next-step prototype [20] and will be used to design the final system. Some confidence was gained that the ATLAS EB system can be built using high-speed interconnects and that the requirements will be met.

5.2 Other TDAQ Simulation

Many other examples of applications of discrete-event simulation for TDAQ systems exist. In the following a few more examples using discrete-event simulation are given for ATLAS and ALICE [21]. The latter being another experiment planned for LHC.

- The use of ATM for the ATLAS EB system has been studied using one simulation program written in μC++ and another one written in *MODSIM*. The two could be compared which helped a lot in the verification phase of the simulation cycle [22].
- The use of SCI for the ATLAS EB system or more generally for TDAQ systems has been studied extensively using *MODSIM*. Some C code concerning details of the SCI standard was included and in particular the scaling of performance with respect to the number of SCI nodes on a ring investigated [23].
- ATLAS has built a general framework for simulation of the full ATLAS TDAQ system. There is a version using *MODSIM* and one using C++. The latter was translated from the *MODSIM* code and enhanced in order to increase the execution speed of the simulation [24].
- ALICE has built a similar framework for simulation of the full ALICE TDAQ system. It is based on *MODSIM* and has a graphical user interface which allows interactive configuration of the simulation, animation of the model and presents the results of the simulation [25].

To summarize, it can be said that discrete-event simulation has proven to be a very useful method to understand TDAQ systems which are very sophisticated and are becoming more complex. They are also quite expensive which makes extensive prototyping nearly impossible. Together with small prototypes, however, simulation can be used to understand and design the full system before it is actually built.

6 CONCLUSION

This paper contains a general introduction to the method of discrete-event simulation. The implementation of discrete-event simulation was discussed and *MODSIM* presented as an example for a discrete-event simulation language. TDAQ systems were shown as an example application of discrete-event simulation and the whole cycle of simulation was demonstrated.

In fact, discrete-event simulation is used in many fields and with an ever growing number of applications. Due to the increase of computing performance and of the number of discrete-event simulation languages or tools this method has a bright future. It supports the trend to describe the real world in digital methods and to perform “measurements” on models instead of measurements directly on the real-world systems. The method of simulation will lead to a proliferation of models which will create a new way to experience the world that surrounds us.

Acknowledgements

I would like to thank R. Jones and T. Wildish for their useful comments on this paper. I am also grateful to all those people with whom I had discussions about discrete-event simulation and who improved my understanding of this method.

References

[1] G.S. Fishman, Concepts and Methods in Discrete-event Digital Simulation, Wiley, New York, 1973.
[2] J. Banks et al., Discrete-event System Simulation, 2nd ed., Prentice-Hall, New Jersey, 1996.
[3] P.J. Erard and P. Déguénon, Simulations par Evenements Discrets (in French), Presses Polytechniques et Universitaires Romandes, Lausanne, 1996.
[4] J.A. Spriet and G.C. Vansteenkiste, Computer-aided Modelling and Simulation, Academic Press, London, 1982.
[5] http://www.spsc.ucalgary.ca/~gomes/HTML/sim.html;
http://www.cs.utsa.edu/research/ParSim;
http://www.science.gmu/edu/~akathri/cne.html.

[6] L. Kleinrock, Queueing Systems, Vols. I&II, Wiley, New York, 1975.
[7] J. Misra, Distributed Discrete-event Simulation, Computing Surveys, Vol. 18, No. 1, March 1986.
[8] F. James, A Review of Pseudorandom Number Generators, Comp. Phys. Comm. 60 (1990), 329.
[9] http://www.isye.gatech.edu/informs-sim/index.html;
http://www.nmsr.labmed.imn.edu/~michael/dbase/outgoing/FAQ.html;
http://aid.wu-wien.ac.at/%7Emitloehn/faq/simul.faq.
[10] CACI Products Comp., La Jolla, California; http://www.cacisl.com.
[11] L. Mapelli, Collisions at Future Supercolliders: the First 10 Microseconds, CERN School of Computing 1991, CERN/92-02;
S. Cittolin, Overview of Data Acquisition for LHC, CERN School of Computing 1994, CERN/95-01;
R. Hubbard, LHC Trigger Design, CERN School of Computing 1997, these proceedings.
[12] ATLAS Collaboration, Technical Proposal for a general-purpose pp Experiment at the Large Hadron Collider at CERN, CERN/LHCC/94-43, 1994; ftp://www.cern.ch/pub/Atlas/TP/tp.html.
[13] CERN's high-speed interconnect pages: http://www.cern.ch/HSI/Welcome.html.
[14] ATM Forum, ATM User Network Interface Specification.
[15] ANSI X3T11, Fibre Channel Standard, ANSI X3.230 (FC-PH).
[16] ANSI X3T11, High Performance Parallel Interface, ANSI X3.183 (HIPPI-PH).
[17] IEEE, Scalable Coherent Interface, IEEE Standard 1596-1992.
[18] M. Letheren, Switching Techniques in Data Acquisition Systems for Future Experiments, CERN School of Computing 1995, CERN/95-05; also: CERN/ECP/95-19.
[19] R. Spiwoks, Evaluation and Simulation of Event Building Techniques for a Detector at the LHC, PhD Thesis, University of Dortmund, 1995, CERN/THESIS/96-002; also: http://atddoc.cern.ch/~spiwoks/papers/diss.ps.
[20] G. Ambrosini et al., The ATLAS DAQ and Event Filter Prototype "-1" Project, presented at the Conf. for Computing in High-energy Physics (CHEP 97), Berlin, 1997; http://atddoc.cern.ch/Atlas/Conferences/CHEP/ID388/ID388.ps.
[21] ALICE Collaboration, Technical Proposal for a Large Ion Collider Experiment at the CERN LHC, CERN/LHCC/95-71, 1995.
[22] RD31 Collaboration, RD31 Status Report 1997, NEBULAS: High Performance Data-driven Event Building Architectures based on Asynchronous Self-routing Packet-switching Networks, CERN/LHCC/97-05, 1997.
[23] RD24 Collaboration, RD24 Status Report 1995, Application of the Scalable Coherent Interface to Data Acquisition at LHC, CERN/LHCC/95-42, 1995.
[24] S. Hunt et al. (ATLAS DAQ Modelling Group), SIMDAQ - A System for Modelling DAQ/Trigger Systems, Proc. Real-Time 95 Conf., 1995; also: IEEE Trans. Nucl. Sci., Vol. 43, No. 1 (Feb. 1996), pp. 69-73.
[25] J. Yuan et al., Simulation of the ALICE Data Acquisition System with GALSIM, presented at the 2nd Int. DAQ Workshop on Networked Data Acquisition Systems, RCNP, Osaka, 13-15 November, 1996.