Discrete Simulation of "Muon Chamber Read-Out" electronics for LHC/ATLAS

an event based simulation in C++

J.M. Weeland

— small edition —

University of Nijmegen, The Netherlands
Faculty of Mathematics and Informatics
Department of Experimental Informatics for Technical Applications
Faculty of Natural Sciences
Department of Experimental High Energy Physics

Katholieke Universiteit Nijmegen
Faculteit der Wiskunde en Informatica
Vakgroep Experimentele Technische Toepassingen
Faculteit der Natuurwetenschappen
Vakgroep Experimentele Hoge Energie Fysica

Master's Thesis, number **326**
August 1994

Afstudeerscriptie nummer **326**
augustus 1994

Preface

It's all over.
Your mission is a failure,
Your lifestyle's too extreme.
from: The Rocky Horror Picture Show.

This thesis is the result of my graduation project to receive my Master's degree in computer science at the University of Nijmegen. The project, which ran from February 1993 through August 1994, was a collaboration between the Department of Experimental Informatics for Technical Applications (formerly called the Department of Real-Time Systems) of the Faculty of Mathematics and Informatics and the Department of Experimental High Energy Physics of the Faculty of Natural Sciences.

The goal of the project was to build a simulation for the electronics of the muon detectors (formally called the 'muon read-out chambers') of the so-called ATLAS detector being built for the Large Hadron Collider (LHC), which will be the world's largest particle accelerator and is to be constructed at CERN in Geneva, Switserland. The major subgoals were *what* to simulate and *how* to simulate. An additional constraint was that the simulation had to be written in the language C++.

At this point I would like to thank dr. Adriaan König and dr. Theo Schouten, my supervisors at the physics department and the computer science department respectively, for their support, advice, help and – most of all – patience, Ing. Thei Wijnen from the physics department for testing the program, and the boys and girls, either at "De Mark", my habitual pub, in the 'canteen' or elsewhere, whom I annoyed with my habits of drinking coffee, smoking (awfully stinking) cigarettes, consuming alcohol and losing money in card games, all of which in astronomical amounts.

Jack Weeland,
Nijmegen, August 1994.

Contents

List of Figures

List of Tables

Introduction

Short introduction to computer simulations

When designing large real world projects, certain design decisions are not known a priori. Depending on the kind of project, several methods to assist making and evaluating these design decisions can be used. For example, in architecture and construction, the technical feasibility of a design is tested with scaled-down models. In computer science it is common to use prototypes of a program to test and try the user interface and functionality.

Types of computer simulations in electronic engineering

In the world of electronic engineering, designs are usually tested using a computer simulation or down-sized prototypes. In the area of computer simulations, globally three types of simulations can be distinguised [Bratley, e.a. 87].

The first type is simulating the *electronic characteristics* of the design. This is usually done by implementing the design using computer software equivalents of the components and defining the connections between these components. The output of the simulation is compared with the expected, or desired, output of the design on an electronical level. This method is applied for testing integrated circuit (IC) or printed circuit board (PCB) designs. This type of simulation can be seen as a *white box* simulation or a *continuous* simulation, since at every moment in time the state of the system can be determined (i.e. the state changes continuously).

The second type is a simulation of the *behaviour* of the design. Instead of simulating (computing) the electronic signals, which flow between the components in the system, the interaction between the system and the outside world and the interaction between the different parts of the system are simulated. This method of simulating is used when the electronical characteristics of the system are either unimportant, unknown or trivial. This type of simulation is usually classified as a *black box* simulation. It is often a discrete simulation since the state transitions of the system only occur at certain, discrete points in time. These state transitions are called events, hence this type of simulation is commonly referred to as an *event based simulation.* They can be divided in two classes: *synchronous* and *asynchronous* simulations. In synchronous simulations the events occur at fixed, regular intervals whereas in asynchronous simulations the events may occur at any time.

Other application areas where event based simulations are widely spread, are business and social sciences. Examples of these applications are given in [Bratley, e.a. 87] and [Banks, Carson 84].

The third kind of computer simulations is the so-called *hybrid simulation*. In this type of simulation a digital computer controls several low-cost parallel analog processors and it is used for real-time simulations, e.g. the simulation of a rocket engine for the space shuttle.

Verification and validation

To make sure the simulation is good, i.e. the model and the program implementing it represent the real system, it has to be *verified* and *validated* ([Bratley, e.a. 87]).

Verification The stage of verification basically comes to checking if the program operates how the implementor thinks it should be operating. The simulation program has to be checked if it works consistently with the model and if it is free of bugs. The latter can be done using a debugger. Checking its consistency with the model is often a matter of believing since there is no exact method to check the consistency between the program and the model.

Validation The validation of the simulation, i.e. checking that the simulation model is a close enough approximation of the real system, is an even more fuzzy area, since there is no recipe of doing such. Usually it comes to running the program a number of times for long periods to check if 'no strange things' occur.

Although both the process of verification and the process of validation are hard to conduct, they can be facilitated by using the right methods and tools. These tools need to be applied throughout the complete process of analysing and specifying the problem area and designing and implementing a solution for the problem and formality of these tools must be restrained. An informal method might result in a vague model of the problem area, hence making it hard to verify or to validate.

Before the problem area can be formally examined, a description of it will be given which serves as an informal specification. From this description some mathematical properties will be derived and those will be used in the formal stages which will lead to the resulting simulation progam.

- The specification will be made using metric temporal logic. This is a strict and formal method, and it has the advantage that the specification can be mathematical proven and it will therefore facilitate the process of validation. A disadvantage of this method, as it is presented in [Koymans 92], is its incapability to express probabilities. Therefore some extensions to metric temporal logic, that fill in this lapse, will be defined in section 4.2.

- During this process it will appear that the problem lends itself very good for an object oriented approach, since the simulation can be abstracted to a number of objects interacting with each other. It will be an obvious choice therefore to use an object oriented analysis and design method to create a design for the implementation, which in its turn will be made using an object oriented programming language. The tools used are the Object Oriented Analysis and Object Oriented Design methods of Coad and Yourdon in the design stage. The implementation will be made using the programming language C++. The Coad and Yourdon design method has been selected out of a number of available methods since it connects very good to the language C++, as came forth from the comparison of a number of similar methods in [van den Goor 92]. Using the concept of classes and objects in C++ makes it easier to isolate errors in the program, thus making it easier to verify the simulation.

Outline of the thesis

The thesis consists of two parts. Part I covers the path and the decisions which eventually resulted in the simulation program. In this part also a boundary will be set to *what*, i.e. which parts of the problem area, has to be simulated. Part II contains appendices, the bibliography and an index.

The user's guide and the programmer's guide for the simulation program are included only in the full edition [Weeland 94].

Part I

Thesis

Chapter 1

Introduction to part I

1.1 How to simulate the muon read-out chamber electronics of ATLAS

In the simulation of the muon read-out chamber electronics of the ATLAS detector only the *behavior* of these has to be considered. Therefore, an obvious choice for the type of simulation will be an event based simulation as it has been discussed on page 1 of the introduction.

1.2 Outline of part I

In chapter 2 an overview will be given of what the ATLAS detector is. Without too much detail, the physics for which ATLAS is to be build, will be discussed and the boundaries and constraints of the simulation will be set. The mathematical properties that are relevant for the simulation are investigated and described in chapter 3. In chapter 4 the events, that occur within the system, are analyzed and delimited to those that are relevant for the simulation, and finally the events are specified using metric temporal logic. Also a short introduction to metric temporal logic will be given. In chapter 5 the design for the program will be described. Chapter 6 covers the implementation of the events and the implementation of the event list and the associated decisions concerning the algorithms and programming language, which will be the language C++. Finally, in chapter 7 some conclusions and 'hints' for further research are discussed.

Chapter 2

The ATLAS detector

In this chapter an overview will be given of what the ATLAS (**A** **T**oroidal **L**HC **A**pparatu**s**) detector is. The purpose for which the detector will be built, is (from [ATLAS92]):

> "The ATLAS collaboration proposes to build a general purpose proton-proton collider for the Large Hadron Collider (LHC), capable of exploring the new energy regime which will become within reach".

In common English this means that the LHC is capable of accelerating elementary particles, in this case protons, much more than existing particle accelerators. Two of these particles collide at a time with an energy that is high enough to hopefully produce particles which at this moment only exist in theory. The ATLAS detector must be capable to detect these particles. A further explanation will be given in this chapter in order to obtain an idea of the processes which are to be simulated.

In section 2.1 the aspects of the physical processes that are involved will be discussed. A general description of the ATLAS detector will be given in section 2.2. In section 2.3 the muon tracking system will be discussed in more detail, since this is the part of the ATLAS detector which will be simulated. Finally in section 2.4 those parts of the muon tracking system that are relevant to the simulation, will be pointed out.

2.1 Physical background

The LHC will be built to invoke high energetic collisions between two hadrons, in particular two protons. The LHC accelerates protons to an energy level that is above the limits of the existing accelerators, so when two protons collide, the energy of the collision will be high enough to produce, among several kinds of other particles, a Higgs boson and a top quark. The existence of the Higgs boson and the top quark[1] has not been demonstrated experimentally so far since the existing accelerators are not capable of delivering the energy needed for this. The search for the Higgs boson, as well as the top quark, is set as one of the goals for the optimization of the ATLAS detector. The

[1] At the end of April 1994 Fermilab claimed to have found evidence for the top quark, [CDF 94].

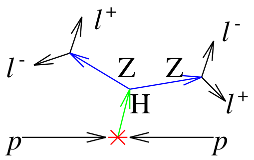


Figure 2.1: *Schematic figure of the proton-proton collision and the subsequent Higgs and Z boson decays.*

search for the top quark, however, is beyond the scope of this thesis and therefore it shall be left for what it is.

The Higgs boson, indicated by the symbol H, can have a number of decays from which the mass of Higgs boson, indicated by $\mathrm{m_H}$, can be derived. The possible decays are mentioned in [ATLAS92], but the most important (for the muon chambers) of the possible decays is: $\mathrm{H} \rightarrow \mathrm{ZZ}^{(*)} \rightarrow \ell^+\ell^-\ell^+\ell^-$ where Z is the so-called Z boson and ℓ is either an electron (indicated by e) or a muon (indicated by the μ symbol). The latter are always produced in either e^+e^- or $\mu^+\mu^-$-pairs. This decay is schematically shown in figure 2.1. More information about some of the physics is given in [ATLAS93a] and [ATLAS92].

2.2 General description of the ATLAS detector

2.2.1 Components of the ATLAS detector

To meet the design goals mentioned in the previous section, the ATLAS detector consists of several units to detect as many particles as possible that might be produced in the proton-proton collisions. The units are:

- The **inner detector**, which provides precision momentum measurements of leptons and identification of electrons and photons as well as some other leptons. The inner detector is surrounded by a superconducting **solenoid coil** which maintains a magnetic field in it.

- **Electromagnetic** and **hadron calorimeters**, which measure the energy of the passing particles. The calorimeters are encapsulated by the so-called **flux return**, which absorbs heavy particles and prevents them from entering the muon chambers and shields the muon chambers from the magnetic field induced by the solenoid coil.

- The **forward calorimeters**. These calorimeters measure the energy of the high rapidity particles.

- The **muon detectors** (consisting of a number of **muon chambers**), which are able to measure the location of passing muon.

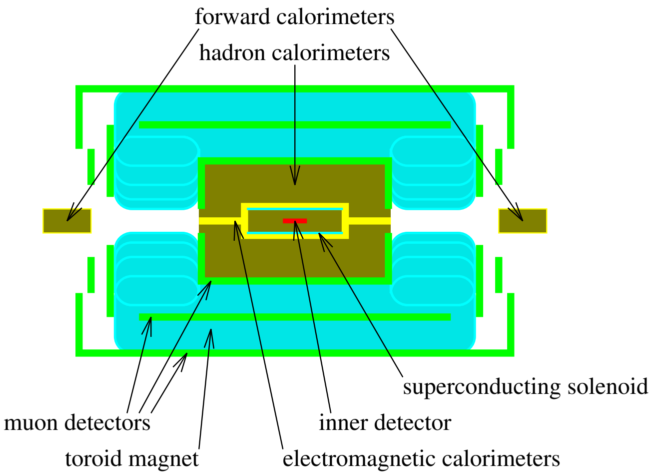


Figure 2.2: *Longitudinal cross section of the ATLAS detector.*

- The muon detectors are placed in magnetic field of about 1 Tesla generated by the superconducting air core **toroid**. The toroid and the muon detectors as such are called the **muon spectrometers** or the **muon tracking system**.

A schematic drawing of the detector design is shown in figure 2.2. The measures of the detector are displayed in figure 2.3.

2.2.2 Global operation of the ATLAS detector

A very important notion of the ATLAS detector is the bunch spacing. The bunch spacing is the time interval between two consecutive proton collisions, or **bunch crossings** (abbreviated to **BX**), which is 25 nanoseconds [ATLAS93b]. Therefore the bunch crossing clock (**BX clock**) is timed at 25 ns or 40 MHz. This clock synchronizes the electronic devices in the detector.

Also important is the trigger that controls the data acquisition. There are three trigger levels, but the most important one for the simulation is the **level 1 trigger**. In general, the trigger is issued if the detectors, among which the inner detector and calorimeters, flag an event to be potentially interesting.

The level 1 trigger has a latency, induced by the cables connected to the trigger control electronics, of 2 microseconds (or less) and this acceptable with regard to making an unambiguous identification of the bunch crossing containing the event of interest. The level 1 trigger has a maximum rate of 100 kHz.

The scope of this thesis does not allow the operation of the detectors to be covered, except for the muon detectors. Those will be discussed in the next section.

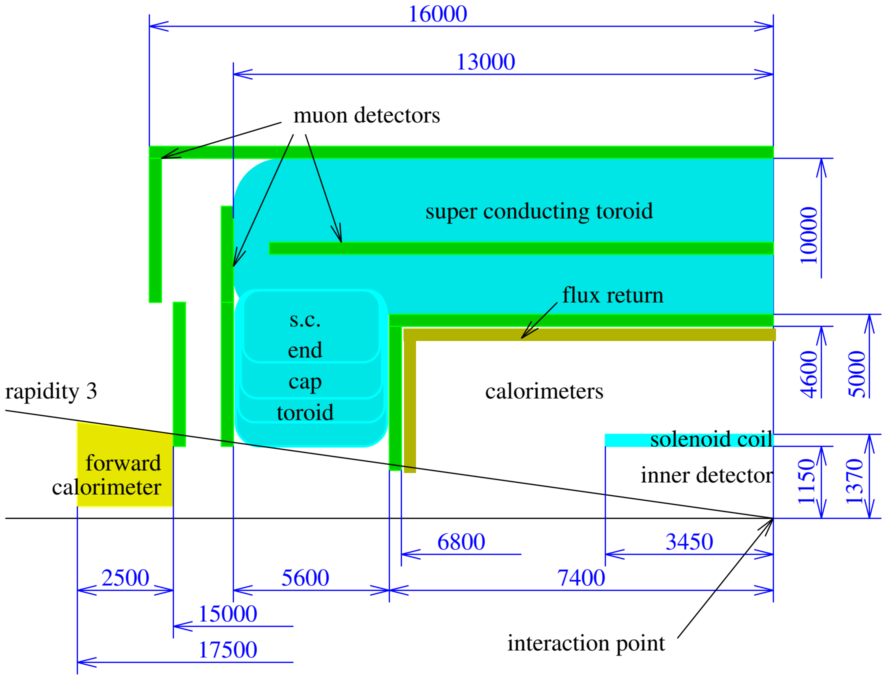


Figure 2.3: *Measures in millimeters within the ATLAS detector.*

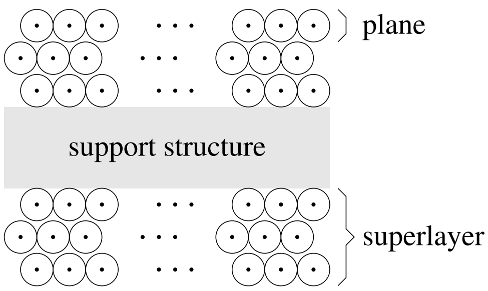


Figure 2.4: *Lay-out of the High-Pressure Drift Tubes muon tracking system.*

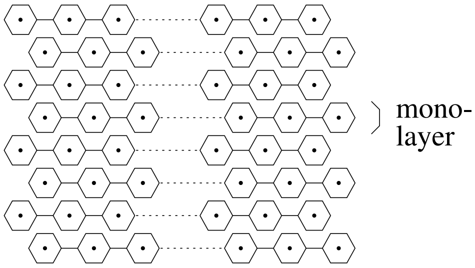


Figure 2.5: *Lay-out of the Honeycomb Strip Chamber muon tracking system.*

2.3 The muon tracking system

The muon tracking system consists of three superlayers of muon detectors. These muon detectors can be regarded as two cooperating sections. The first is the actual detection of a muon when it passes the detector area. The second section is the tagging of the detection with a time stamp. The electronics which perform this task, label the detection with a time stamp and interface to the data acquisition. These sections will be discussed below.

2.3.1 The muon chambers

The actual chamber consists of a 'mattress' of detection cells. At this moment there are three options for these cells, two of which will be discussed, namely the High-Pressure Drift Tubes (HPDT) and the Honeycomb Strip Chambers (HSC, see [Bakker, e.a. 94]).

High-Pressure Drift Tubes; see figure 2.4. The tubes in a HPDT chamber are cylinders filled with gas under pressure. A detection wire is placed in the middle of the tube. An HPDT detector typically consists of two superlayers each constructed of three planes of ninety-six tubes. The main disadvantage of the HPDT is that the coordinates of the wires are not exactly known because the tubes are fixed rather than the wires.

Honeycomb Strip Chambers; see figure 2.5. The HSC are made of a number of monolayers. Each monolayer is made from two folded conductive foils which are joined by welding. The foils are folded in such way that two combined foils make a number of hexagonal tubes. An HSC detector is made of eight monolayers of tubes. As in the HPDT, a detection wire is placed in the middle of each tube, but the HSC have additional copper strips which determine the coordinate along the wire direction. The production process is described in [Bakker, e.a. 94]. The advantage of the HSC is that the wires are fixed and therefore their coordinates are exactly known, but a disadvantage is the non-pressurized gas in the tubes.

The latest development in this area are the **Monitored Drift Tubes** (MDT, which are described in [ATLAS94]), which combines both the aforementioned technologies and some of their advantages. The wires are fixed, so their coordinates are known exactly in contrast to the HPDT, where the tubes are fixed, and the gas in the tubes is pressurized. This is the latest of the technologies and even its name may not yet be final.

The basic idea behind these two types of muon chambers is a tube filled with gas. When a charged particle travels through the tube, it ionizes the gas in the tube. Since the wall of the tube has a negative and the wire a positive potential, the ions drift to the tube's wall and the electrons drift to the wire. When the electrons eventually arrive at wire, they produce a disturbance in the potential, which is called a **hit**. The minimal time it takes to produce a hit is called the **drift time**. The hit then can be detected by the electronics which are attached to the wire and, in case of an HSC, by the electronics connected to the strips on the foil.

The muon detectors have been placed in three superlayers in the longitudinal direction and some layers at the front and the back of the toroid. The toroid maintains a magnetic field of about 1T in the muon chamber area thus causing the (charged) muons to deflect. From the degree of deflection the speed and therefore the energy of the muon can be derived.

However, muons are not the only particles that result in a hit. As a by-product of the physical processes in the detector, a lot of other decays occur within the muon chambers. One of these decays is the decay of a neutron, which causes hits as well and therefore produces at lot of noise on the detection wires. The rate, i.e. the number of events per square centimeter, for the neutron decays is about $1\,000 \frac{\text{Hz}}{\text{cm}^2}$. The hits invoked by a decaying neutron can not be discriminated from a hit by a muon, but the decay products of a neutron decay produce only local hits in one (or sometimes two) tube(s).

2.3.2 The muon read-out electronics

When a disturbance in the potential on the wire is detected, a pre-amplifier and a discriminator generate a digital signal which triggers the electronical devices connected to it to generate a time stamp. This will be collected when a level 1 trigger is issued. The method of collecting these time stamps depends on how the logic in the electronics connected to a wire is organised. Those methods are the following:

1. The signal generated by the discriminator triggers a time to digital converter (abbreviated to **TDC**) to generate a time stamp with a precision of about 1 nanoseconds relative to the bunch crossing clock. This relative time stamp is conjugated with the current BX stamp to form an absolute time stamp. This stamp is stored in a **FIFO** queue. The combination of a wire, a pre-amplifier, a discriminator and a TDC is called a **channel**.

   (a) A **scanner** checks each of the channels to see if an entry in its queue is available. If such an entry is obtainable, it will be put in the local memory of the scanner, which will be called **RAM** (Random Access Memory) for simplicity. The location, the scanner number and the channel number, will be appended to the time stamp. At the next clock tick the scanner will check the next channel. In the sequel this type of scanner will be called the **scanner-by-channel**.

(b) A second alternative for the scanner is one that is very familiar to the one described above. The difference is that this second type, which will be called **scanner-by-entry**, will *not* proceed to the next channel as long as the current channel still has data in its FIFO queue.

The scanners serve 16 or 32 channels, although the exact number of channels needed for proper operation has to come forth from the simulation. A schematic picture of the scanners mentioned above in shown in figure 2.6.

The RAMs are read when a level 1 trigger is issued and the data matching this trigger, i.e. the data that belongs to the same bunch crossing the level 1 trigger belongs to, is sent to the central data processing logic.

2. A variant on the previous scanners is the next one. If in a block of – typically 8 – channels none of the FIFO queues contains any data, all the channels in this block will be skipped. The block is only skipped if the scanner at that time 'points' to the first channel in that block. This type of scanner can use either algorithm 1a or 1b, but the total number of channels for this type of scanner is in the order of 64 or 128. They will be called **blockscanner-by-channel** and **blockscanner-by-entry** respectively.

Not all activity on the wires will be detected. The electric signals on the wire must exceed a certain threshold before the discriminator determines that it is a hit. Moreover, since the signal presented to the discriminator declines slowly, the discriminator is insensitive for another hit as long as the signal on the wire exceeds the threshold. This insensitivity can last for about 100 nanoseconds. This is shown in figures 2.7 and 2.8.

2.4 Area of interest for the simulation

To abstract a usable, small model for the ATLAS muon detectors, certain assumptions will be needed to make. These assumptions are:

- Only a small part of the muon detector will be simulated. This part will be the chamber placed on the far outside of the inner ring of muon detectors. The chamber is highlighted in figure 2.9. The muon rate in this area is about $1.0\frac{\text{Hz}}{\text{cm}^2}$. Selecting a chamber of the inner ring also eliminates the effect of the toroid, so the magnetic field that is maintained in it can be excluded from the simulation.

- The previous point implies that only the muon productions affecting this detector have to be taken into account.

- Only the level 1 trigger will be taken in account with regard to the simulation.

- The data acquisition after a level 1 trigger only has to be rudimentary.

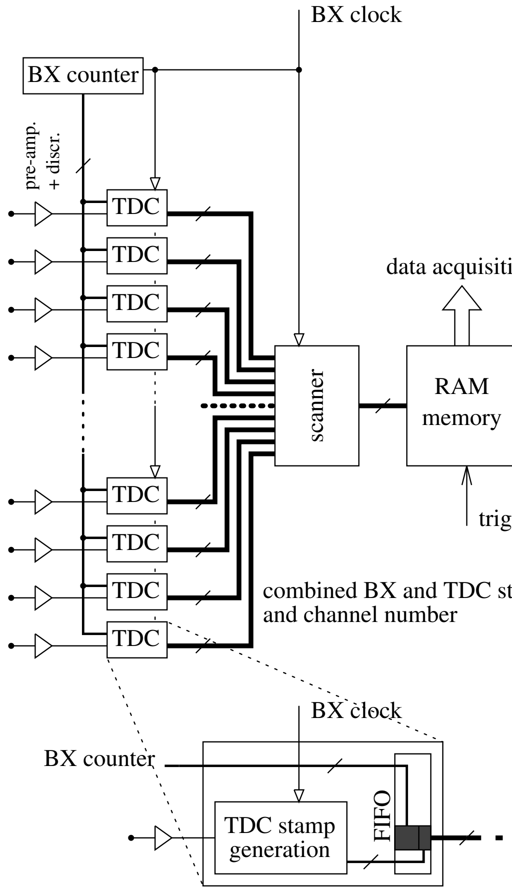


Figure 2.6: *Schematic circuit diagram of the wire read-out electronics with a blown up view of one TDC circuit.*

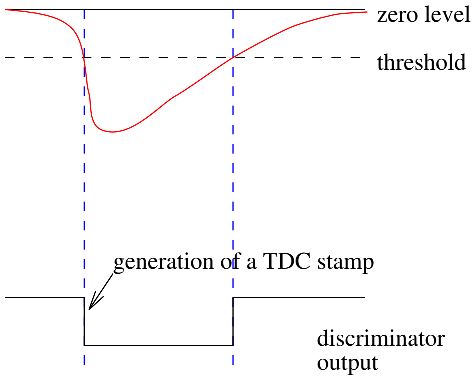


Figure 2.7: *Signal offered to the discriminator and its output with one hit.*

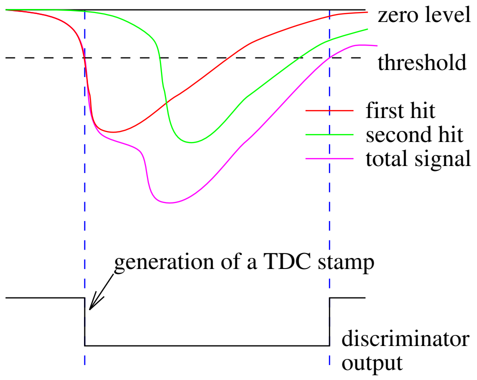


Figure 2.8: *Signal offered to the discriminator and its output with two hits. The signal does not drop below the threshold and the hits will be accounted for as one hit.*

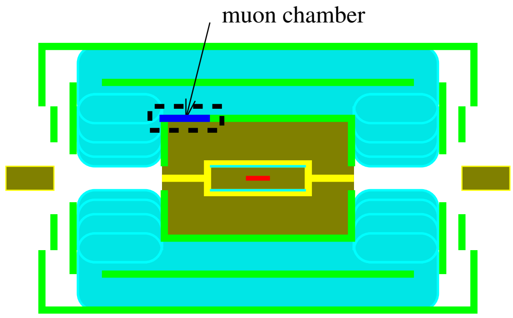


Figure 2.9: *The muon chamber on the outside of the inner ring of detectors which has been chosen to be simulated.*

- Only the muon and the neutron decays will be simulated. Other decays concerning the muon read-out system will be neglected.
- The coordinate along the wire will be left out of the simulation. This way generating a muon track (see section 3.2) will be in a two dimensional plane instead of the three dimensional space.
- The HPDT type of detector will be regarded to as if it had one superlayer consisting of six planes. In this way, the presence of a support structure between the two original superlayers can be disregarded in the simulation.

Chapter 3

Mathematical model of the muon read-out system

Before the simulation can be formalised and implemented, the properties and the behavior of the system to be simulated have to be examined. By such analysis, a mathematical model in which each relevant aspect of the behavior and the properties of the system is included, can be derived. Some parts of this model serve as a basis for the formal specification of the simulation, other parts can be used directly in the implementation.

To analyse the behavior of the ATLAS muon chambers, the next questions have to be examined:

- At what time are the muons detected?
  This question falls apart in three subquestions:
  1. At what time will a muon, that will traverse the selected detector (see section 2.4), be produced in a collision?
  2. How long does it take a muon to travel from the interaction point to a drift tube (*time of flight*)?
  3. How long does detection by the wire take after the gas has been ionized by the muon?
- At what angle did the muon leave the collision?
  This implies the following questions:
  1. Where did the path of the muon cross the detector?
  2. Which drift tube(s) got hit?
  3. Implied by the previous questions: at which distance did the muon pass the wire(s) in the drift tube(s)?

In the remainder of this chapter we only consider the muons which travel through the detector.

The questions posed above will be answered in sections 3.1 and 3.2. Section 3.1 also covers the Poisson distribution and the generation of Poisson distributed random numbers. In section 3.3 the distribution of the neutrons will be examined.

3.1 Distribution of the muon events

In the simulation the time interval between two muon production events has to be calculated. The mean value for the time interval between two events can be derived from the muon rate. To provide an adequate random number generator, the distributions of the events and the time between two events has to be examined.

3.1.1 Distribution of the muons in the experiment

In order to determine the time interval between two muon productions, it is necessary to know the *distribution* of these productions by examining the number of muons released in a certain fixed time interval. That number is presumed to be distributed with a Poisson distribution. In [Bratley, e.a. 87] (five) criteria are mentioned to determine whether an experiment has a Poisson distribution. These criteria apply to the experiment as follows:

1. The process (the production of muons) is continuous in time. The time interval between two muon productions does not decrease as time increases and the distribution of the productions is stepwise (according the muon rate) in time.
2. Only one muon is detected at a time by the detector. It is possible that more than one muon is produced, but the probability that they hit the same detector equals zero since the impulse of the four leptons (in e^+e^- or $\mu^+\mu^-$-pairs) produced (see section 2.1 and figure 2.1) has to be maintained, which implies the leptons escape into different directions.
3. The number of muon productions in a certain time interval is independent of the number of productions in the past.

3.1.2 Distribution and density of the Poisson distribution

In the previous section it is shown that the *number* of muon productions in a certain time interval has a Poisson distribution. The probability of the number of productions in the interval $(0, t_0)$ can be expressed by the formula

$$P(k) = e^{-\lambda t_0} \frac{(\lambda t_0)^k}{k!} \tag{3.1}$$

where λ is the density (i.e. the probability) of the muon productions in a collision and $P(k)$ is the probability that there are k muon productions in the interval $(0, t_0)$, see [Papoulis 91]. Note that this applies to any interval of length t_0, since the process is continuous in time and independent of the past.

Consider the random point t_1 which is the first point to the right (on the time axes) of the fixed point t_0 (figure 3.1) at which a muon is produced. The random variable $\mathbf{x}$ is defined as the distance between t_0 and t_1. Because t_1 is larger than t_0, $\mathbf{x}$ is a positive number. Then the distribution function $F(x)$ of $\mathbf{x}$ is given by

$$F(x) = 1 - e^{-\lambda x}. \tag{3.2}$$

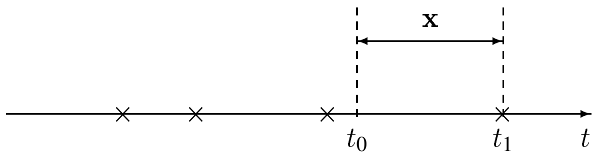


Figure 3.1: *A random point t_1 with a Poisson distribution which is the first event to the right of a fixed point t_0.*

Proof. The distribution function $F(x)$ equals the probability that $\mathbf{x} \leq x$ where x is a specific number. But $\mathbf{x} \leq x$ means that there is at least one point between t_0 and $t_0 + x$. Therefore the probability p_0 that there are *no* points between t_0 and $t_0 + x$ equals $1 - F(x)$. The length of this interval equals x and substituting this in (3.1) yields

$$p_0 = P(k = 0) = e^{-\lambda x} \stackrel{\text{def}}{=} 1 - F(x)$$

and therefore

$$F(x) = 1 - e^{-\lambda x}.$$

□

The derivative of the distribution function, named the density function or *frequency function*, is

$$f(x) = \frac{dF(x)}{dx} = \lambda e^{-\lambda x} \tag{3.3}$$

and this function is an *exponential* function. Therefore, the *time interval* between two muon productions has an *exponential distribution*. This distribution has a mean

$$\mu = \frac{1}{\lambda}. \tag{3.4}$$

Proof. The mean value is defined by

$$\begin{aligned}
\mu &= \int_0^\infty x f(x) dx \\
&= \int_0^\infty \lambda x e^{-\lambda x} dx \\
&\left(\text{rule: } \int u\, dv = uv - \int v\, du\right) \\
&= \left[-x e^{-\lambda x}\right]_0^\infty - \int_0^\infty -e^{-\lambda x} dx
\end{aligned}$$

$$
\begin{aligned}
&\left(\text{use: } \lim_{x\to\infty} -e^{-\lambda x}x = -\lim_{x\to\infty}\frac{x}{e^{\lambda x}} \overset{\text{l'Hôpital}}{=} -\lim_{x\to\infty}\frac{1}{\lambda e^{\lambda x}} = 0\right)\\
&= \; 0 - \int_0^\infty -e^{-\lambda x}dx \\
&= \; -\left[\frac{1}{\lambda}e^{-\lambda x}\right]_0^\infty \\
&= \; \frac{1}{\lambda}
\end{aligned}
$$

□

3.1.3 Generation of random variables

With this knowledge, random variables with an exponential distribution can be generated by inverting the distribution function $F(x)$.

Proof. The distribution function $F(x)$ provides the probability for time intervals $(0, x)$ in which there are no muon productions. This implies the time interval between the productions is 'known' and from that the probability can be derived. On the other hand, if the probability is 'known' (i.e. a 'probability' y is randomly chosen), the related time interval can be derived using the inverse function of $F(x)$, which is:

$$F^{\text{inv}}(y) = \frac{-\ln(1-y)}{\lambda}. \tag{3.5}$$

□

On a computer this calculation can be made easily:

$$X = \frac{-\ln(1-U)}{\lambda}. \tag{3.6}$$

where U is an uniformly distributed random variabele in the interval $(0, 1)$. A further simplification can be made. Since $1 - U$ has the same uniform distribution as U, (3.6) can be transformed into

$$X = \frac{-\ln(U)}{\lambda} = -\ln(U) \cdot \mu. \tag{3.7}$$

A C (or C++) function to calculate (3.7) can be implemented in a staight forward way and is shown in figure 3.2. The standard Unix C-function `drand48()` returns an uniform distributed random number on the interval $(0, 1)$. The argument `mu` is the mean value of the generated numbers and, as is shown in (3.4), $\mu = \frac{1}{\lambda}$. The random number generated has to be checked if it is not too small to prevent the `log` function

```
#include <math.h>
#include <stdlib.h>

double Exponential(const double mu)
{
        double U = drand48();

        if(U > 1e-60) return -log(U) * mu;
        else return 140 * mu;
}
```

Figure 3.2: *C code for the function* `Exponential`.

to give an overflow error. The value 10^{-60} has been taken for the minimum value with $\ln(10^{-60}) \approx -138$ since this is a convenient value for all relevant machine architectures. The value returned for very small U's equals $140 \cdot \mu$ and this is large enough to cover all small random numbers.

This generator has been tested for 1,048,576 numbers and their distrubution is shown in figure 3.3 along with the 'desired' distribution $\lambda e^{-\lambda x}$.

3.1.4 Value of λ and μ in the experiment

In the simulation only one muon detector will be studied. The size of the detector is derived from the following parameters:

l	$:=$	length of a drift tube,	(3.8)
d	$:=$	the diameter of a drift tube,	(3.9)
d_L	$:=$	the distance between the wires of the tubes in two adjacent layers,	(3.10)
d_T	$:=$	the distance between the wires of two adjacent tubes in the same layer,	(3.11)
$\#channels$	$:=$	number of channels per scanner,	(3.12)
N	$:=$	number of scanners in layer,	(3.13)
n	$:=$	the number of layers.	(3.14)

The diameter of the drift tubes[1] in the honeycomb strip chamber (HSC) type of detector is approximated by a circle; see figure 3.4. This approximation has an acceptable error since the muons that are produced and going to hit the selected detector, leave the

[1] In the HSC type detector the tubes are commonly refered to as *cells*. In this thesis however the more general term *tubes* will be used.

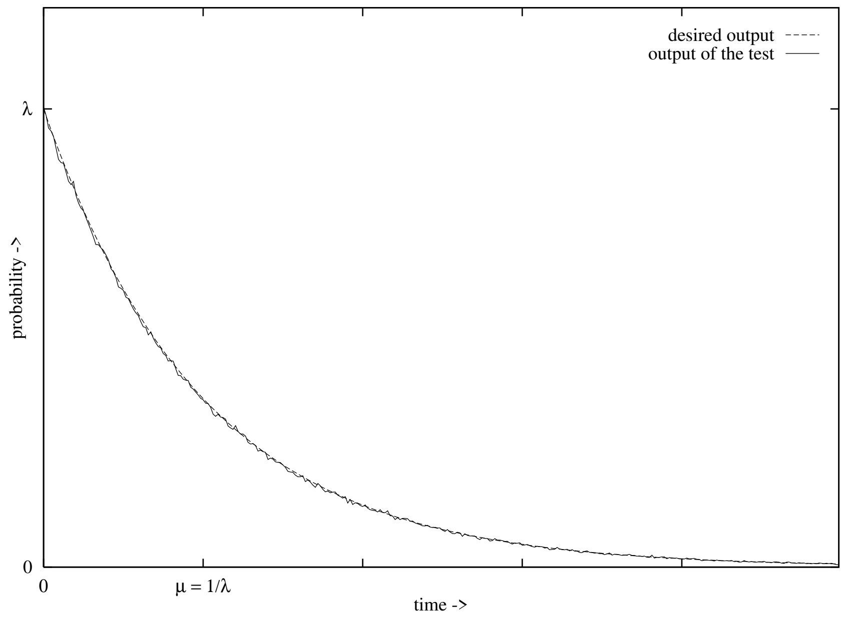


Figure 3.3: *Distribution of 1,048,576 randomly generated numbers and their expected distribution.*

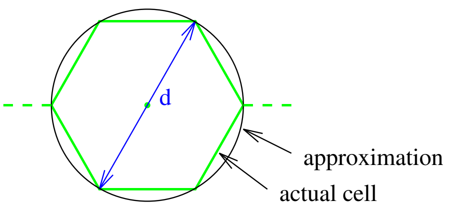


Figure 3.4: *Approximation of the diameter of a cell in the HSC detector by a circle.*

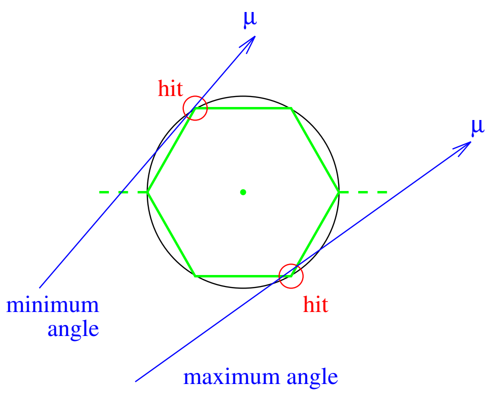


Figure 3.5: *A muon path which intersects the approximation circle, intersects the cell of an HSC detector as well.*

collision under an angle of about 40 degrees. This means that when the muon track intersects the circle, it will intersect the hexagonal cell as well. See figure 3.5.

For the high-pressure drift tubes (HPDT) the following is valid:

$$d = d_T = d_L.$$

The number of tubes in one layer[2] can be derived from (3.13):

$$\#tubes = \#channels \cdot N. \tag{3.15}$$

The symbol $\mathcal{L}$ shall be used to indicate the collection of layers and $\mathcal{T}(l)$ to indentify the tubes in a layer $l \in \mathcal{L}$. For convenience ordinal numbers shall be used for the layers and the tubes in a layer, thus:

$$\begin{aligned} \mathcal{L} &= \mathbb{Z}_{\text{mod } n} && (3.16)\\ \mathcal{T}(l) &= \mathbb{Z}_{\text{mod } \#tubes} \text{ for each } l \in \mathcal{L} && (3.17) \end{aligned}$$

where layer $l = 0$ is the closest to the interaction point (i.e. the most inner layer of tubes) and tube $t = 0$ is the farest (i.e. the most outer) tube in the layer. The notation $\mathcal{T}_l$ shall be used as an abbreviation for $\mathcal{T}(l)$.

To be able to calculate the area of the detector, the width and the length of the detector have to be known. For completeness, the height of the detector shall be noted as well. Since the detector consists of tubes placed side by side, the width of the detector equals the length of a single drift tube (3.8). The length of the detector can be calculated from the number of drift tubes placed in one layer:

$$\begin{aligned} W_d &= l && (3.18)\\ L_d &= \begin{cases} (\#tubes - 1) \cdot d_T + d & \text{if } n = 1 \\ (\#tubes - 1) \cdot d_T + d + \frac{d_T}{2} \\ \quad = \left(\#tubes - \frac{1}{2}\right) \cdot d_T + d & \text{if } n > 1 \end{cases} && (3.19)\\ H_d &= (n - 1) \cdot d_L + d && (3.20) \end{aligned}$$

where W_d is the width of the detector, L_d the length and H_d the height. See figures 3.6 and 3.7.

From (3.18) and (3.19) the area of the detector in cm^2 can be calculated:

$$A = W_d \cdot L_d \tag{3.21}$$

To calculate the mean time interval between two muon productions, the muon rate must be known:

$$\mathcal{R}_\mu := \text{the muon rate in } \tfrac{\text{Hz}}{\text{cm}^2} \tag{3.22}$$

which is a parameter of the simulation. The frequency in Hz of the muon productions can be derived from (3.21) and (3.22):

$$\lambda_\mu = A \cdot \mathcal{R}_\mu. \tag{3.23}$$

[2] An alternative term for *layer* is *plane*.

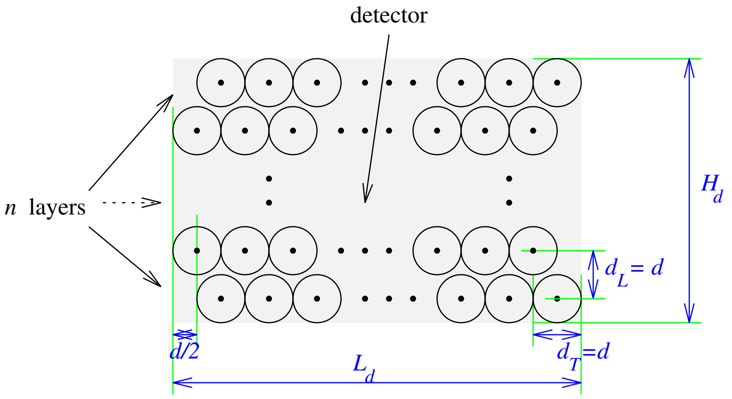


Figure 3.6: *Lay-out of and measures within the HPDT type detector.*

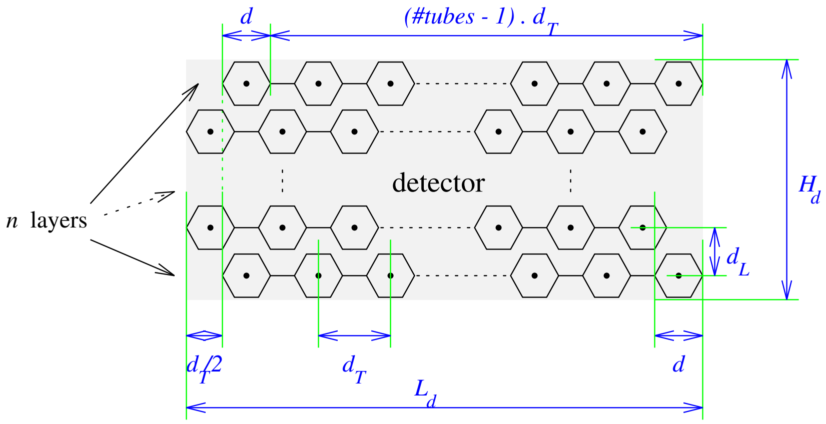


Figure 3.7: *Lay-out of and measures within the HSC type detector.*

The mean time interval between two muon productions in seconds follows from (3.4):

$$\mu_\mu = \frac{1}{\lambda_\mu} = \frac{1}{A\mathcal{R}_\mu}. \tag{3.24}$$

However, in the simulation

$$\mu'_\mu = \mu_\mu \cdot 10^9 \tag{3.25}$$

will be used which indicates the time interval between two muon productions in nanoseconds.

3.2 Calculation of the muon path

After a muon has been produced in a collision, its path has to be calculated. This path is defined by the angle α at which a muon escapes after production in a collision. From this angle α the tube(s) which will be hit can be located and the time it takes the muon to travel from the place of interaction to the detector and how long it will take, after the gas in the tube(s) has been ionized, to be detected by the wire, can be calculated.

3.2.1 Calculation of the angle and travel time

A muon leaves the collision region under a certain angle with respect to the beam direction (the longitudinal axis of the ATLAS detector). Because there is only one detector of interest, as pointed out in section 2.4, picking a spot along the detector will suffice to determine the angle α and to calculate the tubes being hit. This will be done by generating an uniformly distributed random number L'_m. Then the distance from the interaction point to the point where the muon enters the detector is:

$$L_m = L - L'_m \tag{3.26}$$

with

$$L \quad := \quad \text{distance from the interaction point to the edge of the calorimeter area} \tag{3.27}$$

and

$$R \quad := \quad \text{radius of the calorimeter area.} \tag{3.28}$$

To calculate the range of L'_m, α_{max} and L_x have to be defined (see figure 3.9):

$$\begin{aligned} \tan\alpha_{max} &= \frac{H_d}{L_x} = \frac{R + H_d}{L - L_d} \\ L_x &= \frac{H_d}{\tan\alpha_{max}} = \frac{H_d \cdot (L - L_d)}{R + H_d} \end{aligned}$$

thus

$$0 \leq L'_m \leq L_d + L_x = L_d + \frac{H_d \cdot (L - L_d)}{R + H_d}. \tag{3.29}$$

Note that since L'_m is uniformly distributed, it does not matter if L'_m is taken from left to right or from right to left.

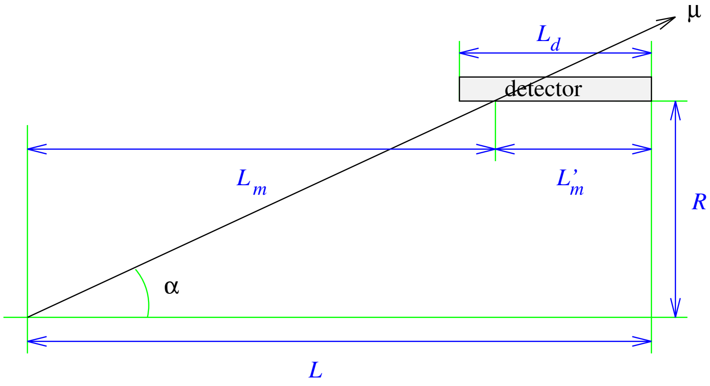


Figure 3.8: *The angle α of the path of a muon.*

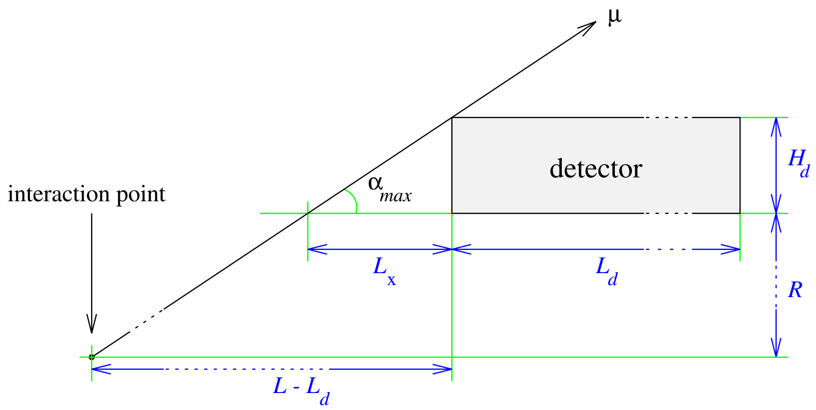


Figure 3.9: *The maximum angle α_{max} of the path of a muon.*

Calculation of the angle α

From (3.26) and (3.28) the angle of the muon path can be calculated (see figure 3.8 for an illustration):

$$\begin{aligned} \tan\alpha &= \frac{R}{L_m} \\ \alpha &= \arctan\frac{R}{L_m}. \end{aligned} \tag{3.30}$$

Calculation of the travel time t_{flight}

From (3.26) and (3.28) (and indirectly from (3.30)) the time it takes a muon to travel from the place of the collision until it enters the detector can be calculated:

$$t_{flight} = \frac{\sqrt{L_m^2 + R^2}}{c} \tag{3.31}$$

where c is the speed of light[3]: $c \approx 3.0 \cdot 10^8 \ \frac{\mathrm{m}}{\mathrm{s}}$.

3.2.2 Calculation of the drift time

In the previous section the time it takes a muon to travel from the place of production to the detector has been calculated. After this, the time interval between ionization of the gas and detection by the wire(s) in the drift tube(s) needs to be calculated. This calculation can be divided in three parts:

- What is the minimum distance at which the muon passes the wire of each drift tube?
- How much time will detection take after the muon ionized the gas?
- How much time did the muon spend in each layer of drift tubes? This time has to be divided in two parts: the time a muon needs to reach to point of minimal distance to a wire and the time after that.

These questions will be answered in the following sections. With regard to the last question the remark need to be made that, of course, the gas which is further than the minimal distance away from the wire is ionised as well. However, this will be compensated for by the electronics connected to the wire and therefore this would not need to be considered in the simulation. See figures 2.7 and 2.8.

Displacement within one layer of drift tubes

Knowing the angle α, the displacement L_e along the axis within one layer of drift tubes is:

$$L_e = \frac{d}{\tan \alpha}. \tag{3.32}$$

This is shown in figure 3.10.

[3] A muon has a speed that is slightly less than that of light. However, this affect is so small that it can be ignored.

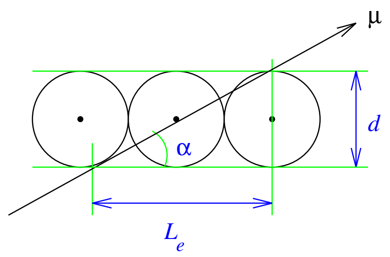


Figure 3.10: *Displacement L_e within one layer of drift tubes.*

Calculation of the drift tubes being hit

To determine which drift tubes have been hit, we have to know where the wires of the drift tubes are situated on the axis. For each drift tube $t \in \mathcal{T}(l)$ and layer $l \in \mathcal{L}$ this is:

$$L(t,l) = \begin{cases} L - \left(d_T \cdot t + \frac{d}{2}\right) & \text{for even } l \\ L - \left(d_T \cdot t + \frac{d_T}{2} + \frac{d}{2}\right) & \text{for odd } l \\ = L - \left(d_T \cdot \left(t + \frac{1}{2}\right) + \frac{d}{2}\right) & \end{cases} \tag{3.33}$$

$$R(t,l) = R + \frac{d}{2} + (d_L \cdot l) \tag{3.34}$$

For each layer l the point where the muon entered (refer to (3.26)) can be calculated:

$$L_m(l) = L_m + l \cdot \frac{d_L}{\tan\alpha} \tag{3.35}$$

From (3.35) the drift tubes that have been passed, but not necessarily hit, by the muon can be derived. The collection of these tubes is defined by $\mathcal{T}_{hit}(l)$:

$$t \in \mathcal{T}_{hit}(l) \quad \text{iff} \quad L_m(l) - \frac{d}{2} \le L(t,l) \le L_m(l) + L_e + \frac{d}{2}. \tag{3.36}$$

for all $t \in \mathcal{T}(l)$ and $l \in \mathcal{L}$. The largest $t \in \mathcal{T}_{hit}(l)$ is:

$$t \le \frac{L_m \cdot (R + l \cdot d_L) - L \cdot R}{R \cdot d_T} - \frac{1}{2} \tag{3.37}$$

which shall be left to the reader to prove.

Calculation of the distances to the wires

The minimum distance from the muon track to the wire is defined by (3.39):

$$L'_e(t,l) = L_m(l) + \frac{L_e}{2} - L(t,l) \tag{3.38}$$

$$L_w(t,l) = \tan\alpha \cdot L'_e(t,l) \tag{3.39}$$

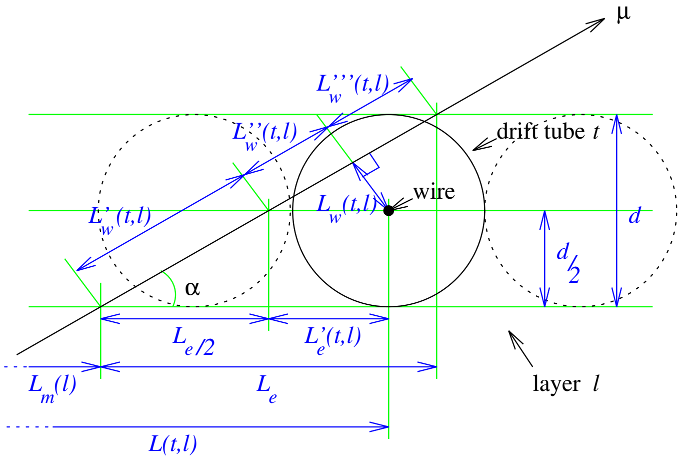


Figure 3.11: *Measures within the drift tube.*

for each tube $t \in \mathcal{T}(l)$ and each layer $l \in \mathcal{L}$. Note that $L'_e(t,l) < 0$ if the muon passes on the left side of the wire and that $L'_e(t,l) > 0$ if the muon passes on the right side. By definition in (3.39), this is also valid for $L_w(t,l)$. These are illustrated in figure 3.11.

Calculation of the time a muon spends in one layer

From the measures below the time a muon spends in one layer of drift tubes can be derived:

$$
\begin{aligned}
L'_w(t,l) &= \sqrt{\left(\frac{L_e}{2}\right)^2 + \left(\frac{d}{2}\right)^2} && (3.40)\\
L''_w(t,l) &= \sqrt{L'_e(t,l)^2 + L_w(t,l)^2} && (3.41)\\
L'''_w(t,l) &= L'_w(t,l) - L''_w(t,l) && (3.42)
\end{aligned}
$$

The time a muon spends within a layer l of drift tubes is:

$$
\begin{aligned}
t_{in\_layer}(l) &= \frac{\sqrt{L_e^2 + d^2}}{c}\\
&= \frac{(L'_w(t,l) + L''_w(t,l) + L'''_w(t,l))}{c}. && (3.43)
\end{aligned}
$$

However $t_{in\_layer}(l)$ is considerably smaller than t_{flight} and $t_{drift}(t,l)$ (see (3.45)) and therefore we will ignore this in the simulation.

Calculation of the drift time

In one of the previous sections the minimum distance at which a muon passes the wire in a drift tube has been calculated. We apply a formula to calculate the drift time suggested by C. Pols[4]:

$$t = a \cdot \frac{x}{r} + b \cdot \left(\frac{x}{r}\right)^2 \tag{3.44}$$

where x is the minimal distance to the center and r the radius of the drift tube. In the HPDT type detector values for a and b have to be chosen such that $a + b = 350$. A good choice will be $a = 100$ and $b = 250$. The formula (3.44) is not very exact, but it is accurate enough for the simulation.

Substituting (3.39) for x, $\frac{d}{2}$ for r and the mentioned values for a and b in (3.44) yields:

$$t_{drift}(t,l) = \begin{cases} 100 \cdot \frac{2\cdot|L_w(t,l)|}{d} + 250 \cdot \left(\frac{2\cdot|L_w(t,l)|}{d}\right)^2 & \text{if } |L_w(t,l)| < \frac{d}{2}, \\ \text{undefined} & \text{otherwise.} \end{cases} \tag{3.45}$$

3.3 Distribution of the neutron decays

Similar to the muon productions, the criteria given in section 3.1.1 apply to the neutron decays too. At the end of section 2.3.1 it is mentioned that the neutron decays only have a local effect. Therefore the neutron distribution shall be regarded on the tube level instead of on the detector level.

The mean time interval μ'_n between two neutron decays in one tube is derived from the following definitions, analogous to (3.21) through (3.25):

$$\mathcal{R}_\mathrm{n} := \text{the neutron decay rate in } \tfrac{\mathrm{Hz}}{\mathrm{cm}^2} \tag{3.46}$$

$$A_\mathrm{tube} = d \cdot l \tag{3.47}$$

$$\lambda_\mathrm{n} = A_\mathrm{tube} \cdot \mathcal{R}_\mathrm{n} \tag{3.48}$$

$$\mu'_\mathrm{n} = \frac{1}{A_\mathrm{tube}\mathcal{R}_\mathrm{n}} \cdot 10^9 \tag{3.49}$$

The products from a decayed neutron have a probability of about 1% to generate a hit in the neighbouring tube as well. This fact, however, will not be taken in account considering the neutron decay rate.

Because of the noisy character of the neutron decays, the drift time will not be considered either. This means that, in the simulation, the neutron decays result in a instantaneous hit.

[4] Private communication, Dr. Ir. C.L.A. Pols.

Chapter 4

Specification of the events in Metric Temporal Logic

To be able to verify and perhaps even validate the simulation, the events, the backbone of the simulation, have to be specified as precisely as possible. Until today, no exact method has been developed to specify events in an event based simulation. It seemed therefore to be a challenge to use metric temporal logic defined in [Koymans 92], which is an extension to modal and temporal logic applied in computer science and philosophy, for the specification.

An introduction to modal, temporal logic and metric temporal logic (MTL) is given in section 4.1. Some extensions to the MTL defined in [Koymans 92] are introduced in section 4.2. In section 4.3, finally, the events that occur within the relevant part of the ATLAS detector – those parts pointed out in section 2.4 – are investigated and specified.

4.1 Modal, temporal and metric temporal logic

In this section a brief overview of modal, temporal and metric temporal logic is given. It will serve as a basis for the other sections of this chapter. A more extensive overview is given in [Koymans 92], chapters 3, 4 and 6. Modal and temporal logic have been used in philosophy for several decades because it is a forceful method to prove propositions. In computer science it is used to prove algorithms and specifications.

Metric temporal logic is an extension to ordinary temporal logic to be able to quantify the time domain. It is defined in [Koymans 92]. The extensions were motivated by a need for a formal specification method for time critical systems.

4.1.1 Modal logic

(Modal) logic starts from a propositional language containing the proposition letters $p, p_1, p_2, \ldots$, $q, q_1, \ldots$, two constants, $\bot$ (falsum) and $\top$ (verum) and the boolean operators $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (if ... then ...) and $\leftrightarrow$ (if and only if). The Greek letters $\varphi, \varphi_1, \ldots$, $\psi, \psi_1, \ldots$, $\chi, \chi_1, \ldots$ denote formulas, which are build from proposition letters or conjunctions of other formulas and the operators mentioned before ([Veldman 84], [Koymans 92]). The subset $\{\rightarrow, \bot\}$ can be taken as a basis for this language. From this the other constant and operators can be derived:

$$\begin{array}{rcl} \top & := & \bot \rightarrow \bot, \\ \neg\varphi & := & \varphi \rightarrow \bot, \\ \varphi \wedge \psi & := & \neg(\varphi \rightarrow \neg\psi), \\ \varphi \vee \psi & := & \neg\varphi \rightarrow \psi, \\ \varphi \leftrightarrow \psi & := & (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi). \end{array}$$

In proof systems the following axiomatization schema of propositional logic is used:

Axiom 4.1 *Modus ponens: to infer ψ from φ and $\varphi \rightarrow \psi$.*

Axiom 4.2 $\varphi \rightarrow (\psi \rightarrow \varphi)$.

Axiom 4.3 $(\varphi \rightarrow (\psi \rightarrow \chi)) \rightarrow ((\varphi \rightarrow \psi) \rightarrow (\varphi \rightarrow \chi))$.

Axiom 4.4 $(\neg\varphi \rightarrow \neg\psi) \rightarrow (\psi \rightarrow \varphi)$.

4.1.2 Temporal logic

To the propositional logic presented in the previous section two modal operators, **L** (necessarily) and **M** (possibly), and four temporal operators, **G** (it is always going to be the case), **F** (eventually; at least once in the future), **H** (it has always been the case) and **P** (at least once in the past), will be added. Before these operators can be formally introduced, the notion of a *model* has to be defined. Again, a more extensive definition is given is [Koymans 92].

Definition 4.1.1 A model is a triple (W, R, V) where W is a (non-empty) set of 'worlds', R is a binary relation on W and V is a valuation on W; it maps proposition letters onto subsets of W (thus giving the set of worlds where the proposition holds). The pair (W, R) is called a frame.

Definition 4.1.2 A modal formula φ holds in a model $\mathcal{M} = (W, R, V)$ at $w \in W$, which is notated as $\mathcal{M}, w \models \varphi$, is defined by recursion:

$$\begin{array}{ll} \mathcal{M}, w \models p & \text{iff } w \in V(p) \text{ (for any proposition letter } p) \\ \mathcal{M}, w \models \bot & \text{for } \textit{no } \mathcal{M} \text{ and } w \\ \mathcal{M}, w \models \varphi \rightarrow \psi & \text{iff } \mathcal{M}, w \models \varphi \Rightarrow \mathcal{M}, w \models \psi \\ \mathcal{M}, w \models \mathbf{L}\,\varphi & \text{iff } \forall w' \in W\ [wRw' \Rightarrow \mathcal{M}, w' \models \varphi] \\ & \\ \mathcal{M} \models \varphi & \text{if } \forall w \in W\ \mathcal{M}, w \models \varphi \end{array}$$

Temporal frames are often called point structures $(T, <)$ where T is the set of 'moments' (points in time) or the *time domain* and $<$ is the 'precedence' or 'earlier' relation. This relation is imposed to be a strict partial order.

Definition 4.1.3 For a temporal formula φ, $\mathcal{M} = (T, <, V)$ and $t \in T$ $\mathcal{M}, t \models \varphi$ is defined in the same way as definition 4.1.2, except that the clause for **L** is replaced by two clauses for **G** and **H**:

$$\begin{array}{lll} \mathcal{M}, t \models \mathbf{G}\,\varphi & \text{iff} & \forall t' \in T\,[t < t' \Rightarrow \mathcal{M}, t' \models \varphi] \\ \mathcal{M}, t \models \mathbf{H}\,\varphi & \text{iff} & \forall t' \in T\,[t' < t \Rightarrow \mathcal{M}, t' \models \varphi] \end{array}$$

Definition 4.1.4 The definitions for **P** and **F**, along with the definitions for **until**, **since**, **D** (different moment), **E** (exists: there is a moment), **A** (always) and **U** (unique moment), are the following:

$$\begin{array}{lll} \mathcal{M}, t \models \mathbf{F}\,\varphi & \text{iff} & \exists t' \in T\,[t < t' \Rightarrow \mathcal{M}, t' \models \varphi] \\ \mathcal{M}, t \models \mathbf{P}\,\varphi & \text{iff} & \exists t' \in T\,[t' < t \Rightarrow \mathcal{M}, t' \models \varphi] \\ & & \\ \mathcal{M}, t \models \varphi_1\ \textbf{until}\ \varphi_2 & \text{iff} & \exists t' \in T\,\Big[t < t'' < t' \text{ and } \mathcal{M}, t' \models \varphi_2 \text{ and} \\ & & \qquad \forall t'' \in T\,[t < t'' < t' \Rightarrow \mathcal{M}, t'' \models \varphi_1]\Big] \\ \mathcal{M}, t \models \varphi_1\ \textbf{since}\ \varphi_2 & \text{iff} & \exists t' \in T\,\Big[t' < t'' < t \text{ and } \mathcal{M}, t' \models \varphi_2 \text{ and} \\ & & \qquad \forall t'' \in T\,[t < t'' < t' \Rightarrow \mathcal{M}, t'' \models \varphi_1]\Big] \\ \mathcal{M}, t \models \mathbf{D}\,\varphi & \text{iff} & \exists t' \in T\,[\neg(t = t') \text{ and } \mathcal{M}, t' \models \varphi] \\ & & \\ \mathbf{E}\,\varphi & := & \varphi \vee \mathbf{D}\,\varphi \\ \mathbf{A}\,\varphi & := & \varphi \wedge \overline{\mathbf{D}}\,\varphi \qquad \text{(see definition 4.1.5)} \\ \mathbf{U}\,\varphi & := & \mathbf{E}(\varphi \wedge \neg\mathbf{D}\,\varphi) \end{array}$$

Definition 4.1.5 Each of these operators has a dual operator, defined as follows:

$$\overline{\mathbf{O}}\,\varphi := \neg\mathbf{O}\neg\varphi.$$

The operators **F** and **G** and the operators **P** and **H** are pairwise duals.

4.1.3 Metric temporal logic

Metric temporal logic is motivated by the need for a formal specification method for time critical systems, such as computer controlled chemical plants, flight control software for airplanes, etcetera. Time critical systems are characterised by *quantitative* timing properties relating occurrences of events ([Koymans 92]). These properties include: minimal and exact (time) distance between an event and its reaction, minimal distance between events, periodicy and bounded response time.

To be able to handle the quantitive temporal properties in the frame $(T, <)$, a measure for the distance between two points in the time domain is needed. Therefore, the distance function d has to be added, where $d(t, t')$ gives a measure how far t and t' are apart in time.

Definition 4.1.6 The distance function d. The range of d is the structure $(\Delta, +, 0)$ of which a definition is given in [Koymans 92], page 105. The distance function is defined by:

1. $d(t,t') = 0 \Leftrightarrow t = t'$,
2. $d(t,t') = d(t',t)$,
3. if $t < t' < t''$ then $d(t,t'') = d(t,t') + d(t',t'')$ and $d(t'',t) = d(t'',t') + d(t',t)$.

The symbol δ is an element in Δ and refers to the distance between two points in time, e.g. $\delta = d(t,t')$. A 'smaller or equal than' relation, indicated by the symbol $\preceq$, and a 'smaller than' relation, indicated by $\prec$, are defined on the elements of Δ. Those relations are defined as

$$
\begin{array}{lcl}
\delta \preceq \delta' & := & \exists \delta'' \left[\delta' = \delta + \delta''\right], \\
\delta \prec \delta' & := & \exists \delta'' \left[\delta'' \neq 0 \text{ and } \delta' = \delta + \delta''\right]].
\end{array}
$$

To impose 'time criticalness' on the unary operators introduced in section 4.1.2, some extensions to these operators need to be made. These extensions are defined in the definitions below.

Definition 4.1.7 The definitions for $\mathbf{F}_\delta$ and $\mathbf{F}_{<\delta}$. The former refers to an exact distance between two events; the latter to the minimal distance between two events.

$$
\begin{array}{lll}
\mathcal{M}, t \models \mathbf{F}_\delta\, \varphi & \text{iff} & \forall t' \in T \left[(t < t' \text{ and } d(t,t') = \delta) \Rightarrow \mathcal{M}, t' \models \varphi\right] \\
\mathbf{F}_{<\delta}\, \varphi & := & \exists \delta' \left[0 \prec \delta' \prec \delta \wedge \mathbf{F}_{\delta'}\, \varphi\right]
\end{array}
$$

The definitions for $\mathbf{P}_\delta$, $\mathbf{P}_{<\delta}$ and similar for those derived from **G**, **H**, **D** and **E**, are analogous.

The definitions above do not take the 'present' into account. To do so, a special operator will be defined, which is called the *reflexive closure*.

Definition 4.1.8 The reflexive closure over the operators **F**, **G**, **P** and **H** is defined as

$$
\begin{array}{rcl}
\dot{\mathbf{F}}\, \varphi & := & \varphi \vee \mathbf{F}\, \varphi \\
\dot{\mathbf{G}}\, \varphi & := & \varphi \wedge \mathbf{G}\, \varphi \\
\dot{\mathbf{P}}\, \varphi & := & \varphi \vee \mathbf{P}\, \varphi \\
\dot{\mathbf{H}}\, \varphi & := & \varphi \wedge \mathbf{H}\, \varphi \\
\\
\dot{\mathbf{D}}\, \varphi & := & \varphi \vee \mathbf{D}\, \varphi \\
 & \equiv & \mathbf{E}\, \varphi
\end{array}
$$

The reflexive closure operator can be applied to the extensions on the operators defined in definition 4.1.7 as well.

4.2 Extensions to Metric Temporal Logic

In this section some operators will be defined that will be needed in the specification of the events in the simulation of ATLAS. In fact, only the Prob-operator, which comes in two 'flavours' and will be defined in definitions 4.2.5 and 4.2.6, will be used in the specification.

Definition 4.2.1 The range operator, $|\ldots|_{t_2}^{t_1}$, limits the time domain of a formula to the interval $[t_1, t_2]$, i.e. a formula φ must hold in that interval.

$$\mathcal{M} \models |\varphi|_{t_2}^{t_1} \text{ iff } \exists t' \in T \left[t_1 \leq t' \leq t_2 \text{ and } \mathcal{M}, t' \models \varphi\right].$$

Definition 4.2.2 The count operator, #, counts the number of occurencies of a formula in the interval $[t_1, t_2]$ of the time domain. It is defined as a recursion.

$$|\#\varphi|_{t_2}^{t_1} = 0 \quad \text{iff} \quad \neg\exists t' \in T \left[t_1 \leq t' < t_2 \text{ and } \mathcal{M}, t' \models \varphi\right], \tag{4.1}$$

$$|\#\varphi|_{t_2}^{t_1} = 1 \quad \text{iff} \quad \mathcal{M}, t_1 \models \varphi, \text{ and } \neg\exists t' \in T \left[t_1 < t' < t_2 \text{ and } \mathcal{M}, t' \models \varphi\right] \tag{4.2}$$

$$\begin{aligned} \exists t' \in T \quad \Big[& t_1 < t' \leq t_2 \text{ and } \mathcal{M}, t' \models \varphi \\ & \underline{\text{and}}\ \neg\exists t'' \in T \left[t_1 < t'' < t' \ \underline{\text{and}}\ \mathcal{M}, t'' \models \varphi\right] \\ & \underline{\text{and}}\ |\#\varphi|_{t_2}^{t_1} = |\#\varphi|_{t'}^{t_1} + |\#\varphi|_{t_2}^{t'} \Big]. \end{aligned} \tag{4.3}$$

In the definition above the formula φ must be discrete, see definition 4.2.4. If (4.1) or (4.2) can be applied, they have precedence over (4.3).

Definition 4.2.3 The count operator can also be defined on the entire time domain.

$$\#\varphi := \lim_{d(t_1,t_2)\to\infty} |\#\varphi|_{t_2}^{t_1}.$$

Definition 4.2.4 The discreteness of a formula in a continuous, i.e. non-discrete, time domain T is defined as follows:

$$\varphi \textit{ is discrete} \quad \text{iff} \quad \neg\exists t_1, t_2 \in T \left[t_1 < t_2 \text{ and } \forall t' \in T \left[t_1 \leq t' \leq t_2 \Rightarrow \mathcal{M}, t' \models \varphi\right]\right]$$

With the definition of the count operator, an operator can be derived which imposes the probability of a certain formula φ. This definition will be given in two steps. The first attempt is a definition using the count operator. The second attempt uses a more algorithmic approach, but it lacks the formality of the first attempt.

Definition 4.2.5 The 'operator' Prob imposes on every validation of a formula φ, a formula ψ having a probability p to be validated as well.

$$\mathcal{M} \models \varphi \to \text{Prob}(p)\psi \quad \text{iff} \quad \frac{\#\psi}{\#\varphi} = p \tag{4.4}$$

Definition 4.2.6 A more algorithmic approach of definition 4.2.5. U is a uniform distributed random variable on the interval $(0, 1)$.

$$\text{Prob}(p)\varphi \quad := \quad \underline{\text{if}}\ U \leq p\ \underline{\text{then}}\ \varphi\ \underline{\text{else}}\ \top\ \underline{\text{endif}} \tag{4.5}$$

Proof. Defintions 4.2.5 and 4.2.6 are equivalent (i.e. (4.4) $\Leftrightarrow$ (4.5)):

($\Leftarrow$) Assume that the left-hand side (4.5) occurs n times. It is trivial that the then-part occurs $p \cdot n$ times (and the else-part occurs $(1-p)\cdot n$ times because the right-hand side of (4.5) must occur n times as well).

Applying axiom 4.2 on the left-hand side of (4.5) yields:

$$\psi \rightarrow \text{Prob}(p)\varphi. \tag{4.6}$$

The left-hand side of (4.5) was assumed to occur n times. Therefore (4.6) occurs n times, which implies that its left-hand side (i.e. the part before the arrow) occurs n times. Therefore:

$$\begin{aligned} \#\psi &= n \\ \#\varphi &= p \cdot n \\ \frac{\#\varphi}{\#\psi} &= \frac{p \cdot n}{n} \\ &= p. \end{aligned}$$

($\Rightarrow$) For the formula φ, the constant $\top$ is filled in. Then the left-hand side of (4.4) can be rewritten to

$$\top \rightarrow \text{Prob}(p)\psi. \tag{4.7}$$

Because of axiom 4.1, (4.7) can be deduced to Prob$(p)\psi$ and this is the left-hand side of (4.5). If the right-hand size of (4.5) is filled in, this yields:

$$\text{Prob}(p)\psi \quad := \quad \underline{\text{if}}\ U \leq p\ \underline{\text{then}}\ \psi\ \underline{\text{else}}\ \top\ \underline{\text{endif}} \tag{4.8}$$

□

4.3 Analysis and specification of the events

Before the events are analyzed and specified, a closer look will be taken to the real-world objects and they will be categorized in a number of sets presented in section 4.3.1. In section 4.3.2 the events occuring in the system will be analyzed and presented. In this process the events will be restricted to those that are relevant for the simulation. Chapter 2 will be used as a guideline during this process. A remark will be made on how to read the specification.

4.3.1 Sets of the real-world objects

To be able to specify the events, the domains of real-world entities, the parameters of the events specified in section 4.3.2, have to be delimited to the following sets:

$$\begin{array}{rcl} T & = & \mathbb{R} \\ & := & \text{the time domain expressed in nanoseconds,} \\[1ex] \mathcal{M} & := & \text{the set of muons,} \\ \mathcal{N} & := & \text{the set of neutrons,} \\ \mathcal{P} & = & \mathcal{M} \cup \mathcal{N} \\ & := & \text{the set of particles,} \\ \mathcal{T} & = & \mathcal{L} \times \mathcal{T}_l = \{\langle l, t\rangle | l \in \mathcal{L} \wedge t \in \mathcal{T}_l\} \\ & := & \text{the set of layer – tube pairs, see (3.16) and (3.17), which} \\ & & \text{shall be used to identify either tubes, wires, channels or} \\ & & \text{FIFO queues,} \\ \mathcal{S} & = & \mathcal{L} \times \mathbb{Z}_{\text{mod } N} \\ & := & \text{the set of layer – scanner pairs,} \\ \mathcal{C} & = & \mathcal{S} \times \mathbb{Z}_{\text{mod } \#channels} \\ & := & \text{the set of scanner – channel pairs,} \\ S & \subset & \mathbb{Q} \\ & := & \text{the set of time stamps, consisting of a BX stamp as the} \\ & & \text{integer part and a TDC stamp as the fraction.} \end{array}$$

4.3.2 Specification of the events

The events specified below refer only to the detector chosen in section 2.4. All time units are expressed in nanoseconds.

Some remarks regarding the specification

- Apart from the interaction between the events, additional constraints have to be specified as well, especially some constraining the behavior of the FIFO queues and the RAM memories. These constraints may seem not to be important for a future implementation, but they are needed to make the specification a solid one. They are necessary to provide facilities to make the specification verifiable, valid and provable. These constraints include:

  No Creation: The events may not be created 'out of the blue', nor may they be duplicated.

  This restriction stems from the subject of [Koymans 92], namely message passing systems. In a way, the FIFO queues are message passing systems as well and for that reason the following restrictions hold for this part of the system as well as they do for any message passing system:

Simultaneous Input: At any moment in time, at most one entry can be inserted into the queue and

Simultaneous Output: At any moment in time, at most one entry from the queue can be delivered to the output.

These restrictions are listed in section 5.2 of [Koymans 92].

- The specification itself is built from several axioms and definitions. An implementation of the system is a valid one if each of the axioms and definitions holds in that implementation, i.e. the axioms and definitions may not be violated. The most appearant of such a violation is the possible overflow of one of the FIFO queues. For example, one of the axioms to be mentioned op page 46 specifies that if an entry is inserted into the queue, it *has* to be delivered to the output somewhere in the future. If the queue overflows, a specific entry will not be delivered to the output since it is either overwritten and therefore no longer available, or it has never been inserted into the queue, depending on the implementation of the FIFO queue. In either way, it is a violation of the axiom and thus this behavior is prohibited from happening.

- To encourage readability of the specification, the axioms are abbreviated to

$$some\_event(e) \rightarrow some\_other\_event(e)$$

instead of the full axiom

$$\exists t \in T \left[\mathcal{M}, t \models \exists e \in \mathcal{E} \left[some\_event(e) \rightarrow some\_other\_event(e)\right]\right]$$

- Unless stated otherwise, different symbols are different objects. For example the statement $some\_event(e) \rightarrow \neg some\_event(e')$ implies $e \neq e'$. Another way to write this is: $(some\_event(e) \rightarrow some\_event(e')) \Leftrightarrow e = e'$.

- The specification will be made in the following steps:

  1. Specification of the physics events.
  2. Specification of the electronics events.
  3. Specification of the events concerning different types of scanners:
     (a) scanner-by-channel,
     (b) scanner-by-entry,
     (c) blockscanner-by-channel,
     (d) blockscanner-by-entry; resulting in a
  4. General specification for all types of scanners.
  5. Specification of the events following a level 1 trigger.

Specification of the physics events

The axioms below specify the existence of the bunch crossings and the collisions that occur on the bunch crossing. Axiom (4.9) specifies strongly that there *is* a bunch crossing and that it is periodic with a cycle time of 25 ns. The parameters $s, s' \in S$ indicate the BX time.

$$\mathbf{E}\ BX(s) \wedge \mathbf{A}(BX(s') \rightarrow \mathbf{F}_{25}\ BX(s'+1)) \tag{4.9}$$

$$BX(s) \leftrightarrow collision$$

The BX time $s \in S$ can be calculated from the real time $t \in T$. To be able to do so, a starting point as to be selected. A convenient starting point will be $t = 0$, so the relation between a BX stamp s and the time t is:

$$s = \left\lfloor \frac{t}{25} \right\rfloor \tag{4.10}$$

and the first bunch crossing occuring at $t = 0$ can be expressed as

$$\mathcal{M}, t \models BX(s) \wedge \neg\mathbf{P}\ BX(s') \quad \text{iff} \quad t = 0$$

If a muon $m \in \mathcal{M}$ is produced, there must have been a collision just before that production. ε is an infinitesimal small number. The other particles that are also produced in the collision (see section 2.1) are not relevant for the simulation and therefore they will be ignored.

$$muon\_production(m) \quad \rightarrow \quad \mathbf{P}_{<\varepsilon}\ collision \tag{4.11}$$

In fact, ε converges to 0 (in practice $\varepsilon = \mathcal{O}(10^{-23})$) and therefore (4.11) can be rewritten to:

$$muon\_production(m) \quad \rightarrow \quad collision$$

Since the collisions occur *only* on a bunch crossing, the event $collision$ shall be left for what it is and (4.11) will be further rewritten to:

$$muon\_production(m) \quad \rightarrow \quad BX(s) \tag{4.12}$$

A muon that has been produced takes a certain time period before it enters the detector. This time period is expressed by t_{flight} (3.31).

$$muon\_production(m) \rightarrow \mathbf{F}_{t_{flight}(m)}\ detector\_enter(m)$$

A muon production has a probability $L1\_prob$ to generate a level 1 trigger in 2 μs. This probability is derived from the level 1 rate, which is about 100kHz. Since the specification only applies to the selected muon detector, one has to take in mind take that a correction factor has to be applied to that probability to compensate for that fact[1]. The level 1 trigger is tagged with the current bunch crossing, indicated by the symbol $s \in S$.

$$muon\_production(m) \rightarrow \mathrm{Prob}(L1\_prob)\ \mathbf{F}_{2000}\ level\_1\_trigger(s)$$

The average time interval between two muon productions is expressed by μ'_μ (3.25). The expression $E(\mu'_\mu)$ returns a random number with an exponential distribution (see section 3.1) with the parameter μ'_μ. Because of (4.12), that interval has to be rounded to a multiple of 25 ns:

$$muon\_production(m) \rightarrow \mathbf{F}_{\left(\left\lceil\frac{E(\mu'_\mu)}{25}\right\rceil\cdot 25\right)}\ muon\_production(m')$$

A certain muon is produced only once and only one muon (that hits the selected detector) is produced at a time (see criterion 2 in section 3.1.1):

$$muon\_production(m) \rightarrow \neg\mathbf{D}\ muon\_production(m) \wedge \neg muon\_production(m')$$

When a muon enters the detector, it ionizes the gas. This $muon\_ionizes\_the\_gas$-event is not interesting and therefore it shall not be specified. It does however produce a hit after t_{drift} nanoseconds (3.45). As indicated on page 31, the time the muon spends in the layer, t_{in_layer} (3.43), is irrelevant so it can be ignored.

$$detector\_enter(m) \rightarrow \forall l \in \mathcal{L}\left[\forall t \in \mathcal{T}_{hit}(l)\left[\dot{\mathbf{F}}_{t_{drift}(m,t,l)}\ hit\_event(m, \langle l, t\rangle)\right]\right]$$

Of course, a muon can only enter the detector if it has been procuded in the past and it will enter the detector only once:

$$detector\_enter(m) \quad \rightarrow \quad \mathbf{P}\ muon\_production(m) \wedge \neg\mathbf{D}\ detector\_enter(m)$$

As indicated in the last paragraph of section 2.3.1, the muons are not the only particles that produce a hit on the wires of the detector. These other particles are the products of a neutron decay and therefore, neutron decays have to be specified (and simulated) as well. The neutron decays have a probability of about 1% to generate a hit on one of the neighbouring wires as well. Hence we need a way to determine which of the (six, see figure 4.1) neighbours is hit. For this purpose, the function `neighbour` will be introduced. Its algorithm is shown in figure 4.2.

The neutron decays have, like the muon productions, an exponential distribution with the parameter μ'_{n}, see (3.49) in section 3.3. Of course, one specific neutron cannot decay in different tubes nor at different moments, which is expressed by axiom (4.13).

$$\begin{aligned}
neutron_decay(n, w) \quad &\rightarrow \quad \mathbf{F}_{\leq\delta}\ hit_event(n, w) \\
&\qquad \wedge\ \mathrm{Prob}(0.01)\ \mathbf{F}_{\leq\delta'}\ hit_event(n, \mathtt{neighbour}(w)) \\
neutron_decay(n, w) \quad &\rightarrow \quad \mathbf{F}_{E(\mu'_{\mathrm{n}})}\ neutron_decay(n', w) \\
neutron_decay(n, w) \quad &\rightarrow \quad \neg\mathbf{D}\ neutron_decay(n, w) \\
&\qquad \wedge\ \neg\mathbf{E}\ neutron_decay(n, w')
\end{aligned} \tag{4.13}$$

[1] The compensation factor is the proportion of the area of the selected detector to the area all detectors in the inner ring of muon detectors

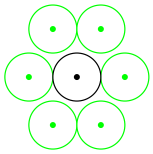


Figure 4.1: *The six neighbours of a drifttube.*

function `neighbour`$(\langle l, t \rangle)$
  [$t' = t - 1, l' = l$
  | $t' = t + 1, l' = l$
  | $l' = l - 1$, if even(l) then $t' = t - 1$ else $t' = t$ end if
  | $l' = l - 1$, if even(l) then $t' = t$ else $t' = t + 1$ end if
  | $l' = l + 1$, if even(l) then $t' = t - 1$ else $t' = t$ end if
  | $l' = l + 1$, if even(l) then $t' = t$ else $t' = t + 1$ end if
  ]
  if $l' \in \mathcal{L}$ and $t' \in \mathcal{T}(l')$ then `neighbour` := $\langle l', t' \rangle$ else `neighbour` := $\bot$ end if
end function

Note: The [*alternative 1* | . . . | *alternative n*] construct represents a non-deterministic choice between one of the alternatives.

Figure 4.2: *The algorithm for the function* `neighbour`.

for tubes $w = \langle l, t\rangle$ and $w' = \langle l', t'\rangle$. The time periods δ and δ' are the drift times of the decay products $n, n' \in \mathcal{N}$. However, the noisy character of the neutron decays justifies taking the values $\delta = \delta' = 0$.

If a particle $p \in \mathcal{P}$ causes a hit on a wire $w \in \mathcal{T}$, this will not always result in a detection. This was made clear in the last paragraph of section 2.3.2.

$$hit\_event(p, w) \wedge \neg insensitive(w) \quad \rightarrow \quad \dot{\mathbf{F}}_{<\varepsilon} detection(w) \tag{4.14}$$

with

$$insensitive(w) \quad := \quad \exists p \in \mathcal{P}\ [\mathbf{P}_{\leq insensitive\_time}\ hit\_event(p, w)]$$

The particle p must be a muon that had entered the detector in the past, or a neutron decay product in either tube w or one of its neighbours.

$$\begin{aligned} hit\_event(p, w) \quad \rightarrow \quad & \Big(\big(p = m \wedge \dot{\mathbf{P}}\ detector\_enter(m)\big) \\ & \quad \vee \\ & \quad \Big(p = n \wedge \dot{\mathbf{P}} \Big(neutron\_decay(n, w) \\ & \qquad\quad \vee\ neutron\_decay\,(n, \mathtt{neighbour}(w))\Big)\Big)\Big) \end{aligned}$$

A particle cannot cause a hit twice, nor can it cause a hit on another wire unless there was a neutron decay in one of the neighbouring tubes. Note that ‘particle’ refers to an individual particle and *not* to a class of particles, e.g. muons.

$$\begin{aligned} hit\_event(p, w) \quad \rightarrow \quad & \neg\mathbf{D}\ hit\_event(p, w) \\ & \wedge \neg\mathbf{E}\,(hit\_event(p, w') \rightarrow \\ & \qquad \neg\,(w' = \mathtt{neighbour}(w) \wedge p = n \\ & \qquad\quad \wedge\ neutron\_decay(n, w'))) \end{aligned}$$

Specification of the electronics events

When a hit on a wire $w = \langle l, t\rangle$ is detected, a time stamp $s \in S$ is placed in the FIFO queue, see section 2.3.2.

$$detection(w) \quad \rightarrow \quad \dot{\mathbf{F}}_{<\varepsilon}\ FIFO\_in\_event(w, s)$$

$$detection(w) \quad \rightarrow \quad \exists p \in \mathcal{P}\Big[\dot{\mathbf{P}}\ hit\_event(p, w)\Big]$$

If a time stamp $s \in S$ is entered in the FIFO queue, the channel $c \in \mathcal{T}$ belonging to the FIFO will be scanned in the future. The function `scanner` converts a layer – tube pair to a layer – scanner pair and is shown in figure 4.3.

$$FIFO\_in\_event(c, s) \quad \rightarrow \quad \mathbf{F}\ scan\_event(\mathtt{scanner}(c), s) \tag{4.15}$$

$$FIFO\_in\_event(c, s) \quad \rightarrow \quad \dot{\mathbf{P}}\ detection(c)$$

<u>function</u> $\texttt{scanner}(\langle l, t\rangle)$
  $\texttt{scanner} := \langle l, l \ \underline{\text{mod}}\ N\rangle$
<u>end function</u>

<u>function</u> $\texttt{scanner}(\texttt{channel}(\langle l, t\rangle))$
  $\texttt{scanner} := \texttt{scanner}(\langle l, t\rangle)$
<u>end function</u>

Figure 4.3: *The polymorphic function* `scanner` *calculates a layer – scanner pair from a layer – tube pair or a scanner – channel pair.*

<u>function</u> $\texttt{channel}(\langle l, t\rangle)$
  $\texttt{channel} := \langle \texttt{scanner}(\langle l, t\rangle)\ , t \ \underline{\text{mod}}\ \#channels\rangle$
<u>end function</u>

Figure 4.4: *The function* `channel` *calculates a scanner – channel pair from a layer – tube pair.*

<u>function</u> $\texttt{succ}(\langle l, t\rangle)$
  $\texttt{succ} := \texttt{channel}(\langle l, t+1\rangle)$
<u>end function</u>

<u>function</u> $\texttt{succ}(c, count)$
  $next := c$
  <u>for</u> $i := 1$ <u>to</u> $count$
  <u>do</u> $next := \texttt{succ}(next)$ <u>end do</u>
  $\texttt{succ} := next$
<u>end function</u>

Figure 4.5: *The polymorphic function* `succ` *returns the next channel relative to the channel passed as the parameter of this function. If that channel is the last channel, the first one will be returned. When the parameter* $count$ *is specified, the* $count$*th successor will be returned. Note that this function is also defined for* $count = 0$.

<u>function</u> $\texttt{pred}(\langle l, t\rangle)$
  $\texttt{pred} := \texttt{channel}(\langle l, t-1\rangle)$
<u>end function</u>

<u>function</u> $\texttt{pred}(c, count)$
  $previous := c$
  <u>for</u> $i := 1$ <u>to</u> $count$
  <u>do</u> $previous := \texttt{pred}(previous)$ <u>end do</u>
  $\texttt{pred} := previous$
<u>end function</u>

Figure 4.6: *The polymorphic function* `pred`*. Analogous to the function* `succ`*.*

The following axioms impose the FIFO behavior on a FIFO queue $f \in \mathcal{T}$:

$$FIFO\_in\_event(f,s) \rightarrow \mathbf{F}\ FIFO\_out\_event(f,s)$$

$$\begin{aligned} FIFO\_in\_event(f,s) \wedge \\ \mathbf{F}\ FIFO\_in\_event(f,s') &\rightarrow \mathbf{F}\,(FIFO\_out\_event(f,s) \\ &\qquad \wedge\, \mathbf{F}\ FIFO\_out\_event(f,s')) \end{aligned}$$

$$FIFO\_out\_event(f,s) \rightarrow \mathbf{P}\ FIFO\_in\_event(f,s)$$

$$\begin{aligned} FIFO\_out\_event(f,s) \wedge \\ \mathbf{F}\ FIFO\_out\_event(f,s') &\rightarrow \mathbf{P}\,(FIFO\_in\_event(f,s) \\ &\qquad \wedge\, \mathbf{F}\ FIFO\_in\_event(f,s')) \end{aligned}$$

Only one time stamp is inserted into or read from the queue at a time and a stamp is only inserted and/or read once:

$$\begin{aligned} FIFO\_in\_event(f,s) &\rightarrow \neg FIFO\_in\_event(f,s') \wedge \neg\mathbf{D}\ FIFO\_in\_event(f,s) \\ FIFO\_out\_event(f,s) &\rightarrow \neg FIFO\_out\_event(f,s') \wedge \neg\mathbf{D}\ FIFO\_out\_event(f,s) \end{aligned}$$

Definition (4.16) checks if a certain FIFO $f \in \mathcal{T}$ contains data. Definition (4.17) specifies the action to be taken if a channel is scanned. If the FIFO contains data, the data will be read from the FIFO and, in a preferably small amout of time ε, transfered into the memory of the scanner (the RAM). The function `channel` is shown in figure 4.4.

$$\begin{aligned} &FIFO\_contains\_data(\texttt{channel}(f)) := \\ &\quad \exists s \in S\,[\mathbf{P}_{\delta} FIFO\_in\_event(f,s) \wedge \neg\mathbf{F}_{<\delta} FIFO\_out\_event(f,s)] \end{aligned} \tag{4.16}$$

$$\begin{aligned} &scan\_channel(\texttt{channel}(f)) := \\ &\quad FIFO\_contains\_data(\texttt{channel}(f)) \rightarrow \\ &\quad (FIFO\_out\_event(f,s) \wedge \mathbf{F}_{<\varepsilon}\ RAM\_in\_event(\texttt{scanner}(f),s)) \end{aligned} \tag{4.17}$$

The only data that can be inserted into the memory is the data that has been read from the FIFO queue f of one of the channels. The events concerning the RAM memories have no special constraints, except for axiom 4.18 which is a timing constraint. The hit represented by the time stamp $s \in S$ must not have been detected more than $2000 - max\_drift\_time$ ns prior to the insertion into the memory to prevent it from not being in the memory if a level 1 trigger forces the memory to be read. Of course, only one entry can be inserted into the memory of scanner $d \in \mathcal{S}$ at a time.

$$RAM\_in\_event(\texttt{scanner}(f),s) \rightarrow \mathbf{P}_{<\varepsilon}\ FIFO\_out\_event(f)$$

$$\begin{aligned} &RAM\_in\_event(\texttt{scanner}(f),s) \rightarrow \\ &\qquad \exists p \in \mathcal{P}\,[\mathbf{P}_{\leq 2000 - max\_drift\_time}\ detection(p,f)] \end{aligned} \tag{4.18}$$

$$RAM\_in\_event(d,s) \rightarrow \neg RAM\_in\_event(d,s')$$

Specification of the scanner-by-channel events

The events concerning the scanner-by-channel (see section 2.3.2, item 1a), are quite straight forward. The time period $scan\_cycle$ is the time it takes to scan one channel, usually the bunch crossing time, i.e. 25 ns.

$$\begin{aligned} scan\_event(\texttt{scanner}(c)) \;&\rightarrow\; \big(scan\_channel(c) \\ &\qquad \wedge \mathbf{F}_{scan\_cycle}\; scan\_event(\texttt{scanner}(c \oplus 1))\big) \end{aligned} \tag{4.19}$$

$$scan\_event(\texttt{scanner}(c)) \;\rightarrow\; \mathbf{P}_{scan\_cycle}\; scan\_event(\texttt{scanner}(c \ominus 1)))$$

The operators $\oplus$ and $\ominus$ are defined as:

$$c \oplus i \;:=\; \texttt{succ}(c, i) \tag{4.20}$$
$$c \ominus i \;:=\; \texttt{pred}(c, i) \tag{4.21}$$

The functions `succ` and `pred` are shown in figures 4.5 and 4.6.

Specification of the scanner-by-entry events

The specification for the scanner-by-entry (see item 1b in section 2.3.2) is slightly more complicated:

$$\begin{aligned} &\big(scan\_event(\texttt{scanner}(c)) \wedge FIFO\_contains\_data(c)\big) \rightarrow \\ &\qquad \big(scan\_channel(c) \\ &\qquad \wedge \mathbf{F}_{scan\_cycle}\; scan\_event(\texttt{scanner}(c))\big) \end{aligned} \tag{4.22}$$

$$\begin{aligned} &\big(scan\_event(\texttt{scanner}(c)) \wedge \neg FIFO\_contains\_data(c)\big) \rightarrow \\ &\qquad \mathbf{F}_{scan\_cycle}\; scan\_event(\texttt{scanner}(c \oplus 1)) \end{aligned} \tag{4.23}$$

The axioms (4.22) and (4.23) can be combined in one axiom with the use of axiom B.2:

$$\begin{aligned} &scan\_event(\texttt{scanner}(c)) \;\rightarrow \quad (4.24) \\ &\Big(\big(FIFO\_contains\_data(c) \;\rightarrow \\ &\quad \big(scan\_channel(c) \wedge \mathbf{F}_{scan\_cycle}\; scan\_event(\texttt{scanner}(c))\big)\big) \\ &\quad \vee \\ &\quad \big(\neg FIFO\_contains\_data(c) \;\rightarrow \\ &\quad \mathbf{F}_{scan\_cycle}\; scan\_event(\texttt{scanner}(c \oplus 1))\big)\Big) \end{aligned} \tag{4.25}$$

Specification of the blockscanner-by-channel events

If the channel c to be scanned is the start of a new block of channels and the FIFO queues of that block are empty, then the next channel to be scanned will be the one that is the first one of the next block. If the block is not empty, the scanner acts as a common scanner-by-channel. The constant CBS is the number of channels that make up a block, or channel block size.

$$\begin{aligned}
&scan_event(\mathtt{scanner}(c)) \;\rightarrow \\
&\quad \Big((can_skip_block(c, CBS) \rightarrow \\
&\qquad\quad \mathbf{F}_{scan_cycle}\ scan_event(\mathtt{scanner}(c \oplus CBS))) \\
&\quad \vee \\
&\quad (\neg can_skip_block(c, CBS) \rightarrow \\
&\qquad\quad (scan_channel(c) \wedge \mathbf{F}_{scan_cycle}\ scan_event(\mathtt{scanner}(c \oplus 1)))) \Big)
\end{aligned} \tag{4.26}$$

with

$$can\_skip\_block(c, CBS) := \left(c \bmod CBS = 0 \wedge \bigwedge_{i=0}^{CBS-1} \neg FIFO\_contains\_data(c \oplus i) \right) \tag{4.27}$$

Specification of the blockscanner-by-entry events

$$\begin{aligned}
&scan_event(\mathtt{scanner}(c)) \;\rightarrow \\
&\quad \Big((can_skip_block(c, CBS) \rightarrow \\
&\qquad\quad \mathbf{F}_{scan_cycle}\ scan_event(\mathtt{scanner}(c \oplus CBS))) \\
&\quad \vee \\
&\quad (\neg can_skip_block(c, CBS) \;\rightarrow \\
&\qquad (FIFO_contains_data(c) \rightarrow \\
&\qquad\quad (scan_channel(c) \wedge \mathbf{F}_{scan_cycle}\ scan_event(\mathtt{scanner}(c)))) \\
&\qquad \vee \\
&\qquad (\neg FIFO_contains_data(c) \rightarrow \\
&\qquad\quad \mathbf{F}_{scan_cycle}\ scan_event(\mathtt{scanner}(c \oplus 1))) \\
&\quad)\Big)
\end{aligned} \tag{4.28}$$

General specification for the scanners

The specification for the different types of scanners, (4.19), (4.25), (4.26) and (4.28) can be united to one general specifiing axiom. The first step to this specification is abstracting the non-block scanners to block scanners with block size $CBS = 1$. The next step is introducing a global constant $algorithm$ indicating the algorithm to be used. It is defined as:

$$algorithm \quad \in \quad \{channel, entry\} \tag{4.29}$$

The general scanner specification then becomes:

$$
\begin{aligned}
& scan\_event(\texttt{scanner}(c)) \;\rightarrow \\
& \quad \Big((can\_skip\_block(c, CBS) \rightarrow \\
& \qquad\quad \mathbf{F}_{scan\_cycle}\ scan\_event(\texttt{scanner}(c \oplus CBS))) \\
& \qquad \vee \\
& \qquad (\neg can\_skip\_block(c, CBS) \;\rightarrow \\
& \qquad\quad (FIFO\_contains\_data(c) \rightarrow \\
& \qquad\qquad (scan\_channel(c) \wedge \\
& \qquad\qquad\quad ((algorithm = channel) \rightarrow \\
& \qquad\qquad\qquad \mathbf{F}_{scan\_cycle}\ scan\_event(\texttt{scanner}(c \oplus 1))) \\
& \qquad\qquad\quad \vee \\
& \qquad\qquad\quad ((algorithm = entry) \rightarrow \\
& \qquad\qquad\qquad \mathbf{F}_{scan\_cycle}\ scan\_event(\texttt{scanner}(c)))) \\
& \qquad\quad \vee \\
& \qquad\quad (\neg FIFO\_contains\_data(c) \rightarrow \\
& \qquad\qquad \mathbf{F}_{scan\_cycle}\ scan\_event(\texttt{scanner}(c \oplus 1))) \\
& \qquad) \Big)
\end{aligned}
\tag{4.30}
$$

Specification of the events after a level 1 trigger

If a level 1 trigger is issued, the RAM memories of the scanners will be read and 2 μs before the trigger, a muon must have been produced.

$$
\begin{aligned}
level\_1\_trigger(s) \quad &\rightarrow \quad read\_RAMs\_event(s) \\
level\_1\_trigger(s) \quad &\rightarrow \quad \mathbf{P}_{2000}\ muon\_production(m)
\end{aligned}
$$

The valid data in the RAM memories, i.e. the data of which the time stamp is maximal 2000 ns and minimal $2000 - max\_drift\_time$ ns prior to the level 1 trigger, is send to some central data acquisition point, which is beyond the scope of the simulation and shall not be specified. The max_drift_time is about 350 ns. Note that s is the BX stamp of the bunch crossing which eventually resulted in the level 1 trigger.

$$
\begin{aligned}
& read\_RAMs\_event(s) \rightarrow \forall d \in \mathcal{S}, s' \in S \\
& \quad \left[\left(\mathbf{P}\ RAM\_in\_event(d, s') \wedge s \leq s' \leq s + \left\lceil \frac{max\_drift\_time}{25} \right\rceil \right) \right. \\
& \qquad \left. \rightarrow RAM\_out\_event(d, s') \right]
\end{aligned}
$$

Some more constraints: the RAM memories are only read when a level 1 trigger is issued and an entry can only be read from the memory if it has been inserted in the past.

$$
read\_RAMs\_event(s) \quad \rightarrow \quad level\_1\_trigger(s)
$$

event	real-world objects involved
BX	BX clock, BX stamp $s \in S$
$muon\_production$	muon $m \in \mathcal{M}$
$detector\_enter$	muon $m \in \mathcal{M}$
$neutron\_decay$	neutron $n \in \mathcal{N}$, tube $w \in \mathcal{T}$
hit_event	particle $p \in \mathcal{P}$, tube $w \in \mathcal{T}$
$detection$	tube $w \in \mathcal{T}$
$FIFO\_in\_event$	FIFO $f \in \mathcal{T}$, time stamp $s \in S$
$FIFO\_out\_event$	FIFO $f \in \mathcal{T}$, time stamp $s \in S$
$scan\_event$	scanner $d \in \mathcal{S}$, channel $c \in \mathcal{C}$, $(algorithm)$
RAM_in_event	scanner $d \in \mathcal{S}$, channel $c \in \mathcal{C}$, time stamp $s \in S$
RAM_out_event	scanner $d \in \mathcal{S}$, time stamp $s \in S$
$read\_RAMs\_event$	time stamp $s \in S$
$level\_1\_trigger$	time stamp $s \in S$

Table 4.1: *Summary of the (relevant) events occuring within the ATLAS detector and the real-world objects involved with these events.*

$$RAM\_out\_event(d, s) \quad \rightarrow \quad \mathbf{P}_{\leq 2000}\, RAM\_in\_event(d, s)$$

Note that entries inserted into a RAM memory are not to be read per se.

Chapter 5

Design of the simulation program

In this chapter, the events and objects that are relevant for the simulation will be designed. Because the simulation is to be written in an object oriented language, it is obvious to use an object oriented method for the design. Until today, quite a lot of such analysis and design methods have been developed. Six of those are described and compared in [van den Goor 92]. A method that connects rather good to the language C++ is the Object Oriented Analysis and Object Oriented Design method of Coad and Yourdon, described in [Coad, Yourdon 91a] and [Coad, Yourdon 91b]. It uses a clear graphical notation which will be briefly introduced in section 5.1.

The two phases of the design will be applied to the ATLAS muon chamber read-out electronics simulation in the sections 5.2 and 5.3.

5.1 Introduction to OOA/OOD

5.1.1 History and motivation

The Coad and Yourdon method makes a rigid distinction between the analysis and design phase ([Coad, Yourdon 91a, Coad, Yourdon 91b]). Historically the step from a problem to an implementation was made only with one design step in between utilizing functional decomposition methods or, even when the distinction between analysis and design was made, the analysis only served as input to the design. These methods were aimed at procedural languages like Fortran and Cobol, etc. The method of functional designing however does not connect very much to the way humans think. Object orientedness connects much better because the world, and therefore often the problem that is to be realized, consists of objects and their interaction. For this reason several object oriented design methods have been developed. However, the step from the problem to a design is still a fairly large step and errors are easily made because the problem described by a design is poorly understood. A thorough analysis of the problem area *and* a proper feed-back between the analysis and the design enables the designer to understand the problem better and therefore the resulting design will improve.

5.1.2 Approach and notation

The object oriented analysis consists of five major activities ([Coad, Yourdon 91a]):

1. **Finding the classes and objects.** A class is a collection of objects that share the same structure, properties and behavior.

2. **Identifying the structures.** There are two structures:

   (a) **Generalization-Specialization relationships between classes.** This is also know under the term *inheritance*. A class can be a specialization of one other class, e.g. a taxi is a kind of car, but a class can also be a specialization of several classes, e.g. a taxi driver is a person *and* an employee of the taxi company. The latter case is called *multiple inheritance*.

   (b) **Whole-Part relationships between objects.** An object can be decomposed into other objects with possibly specific amounts or ranges. There are basically three variations of the whole-part relationship:

      i. **Assembly-Parts.** A car has wheels, an engine, etc.
      ii. **Container-Contents.** A garage contains a number of cars.
      iii. **Collection-Members.** A taxi company employs a number of taxi drivers.

3. **Identifying subjects.** A subject divides the problem area in more understandable parts. E.g. subjects that can be identified in the taxi company are the organization, personnel and the vehicles.

4. **Defining attributes.** The attributes of a class are the data members (or the status information) which can be different for each object in the class. An **instance connection** is made between two objects if they need each other to fulfill their responsibilities. E.g. a taxi driver works at a taxi company.

5. **Defining services.** A service defines the behavior that is expected from an object. Objects communicate through **message connections**, e.g. the taxi company sends a driver somewhere to pick up a client. Services are described using a service chart. See figure 5.2 for the notation used in these charts.

The five activities are not sequential. They can be carried out in any order. The graphical notation of the concepts mentioned above are shown in figure 5.1. The objects in the diagram can be divided into five layers, analogous to the five activities, which can be presented separately as if they were printed on transfers. These five layers are:

1. the subject layer,

2. the class&object[1] layer,

3. the structure layer,

[1]Class&object is short for ‘a class and the objects in that class’.

4. the attribute layer,
5. the service layer.

The object oriented design phase adds four different components to the five layers. These components are ([Coad, Yourdon 91b, van den Goor 92]):

1. **The problem domain component.** The object oriented analysis is carried into the object oriented design and some improvements are made. The design is checked if previously designed classes exist that can be reused and the design is adapted for the target language and – if necessary – the target operating system.
2. **The human interaction component.** The users of the system are classified and their tasks and needs are examined. The user interaction is tested using a prototype of the (user interface of the) program.
3. **The task management component.** A check is made if the system needs special tasks. The types of systems that need tasks include multi-user and multi-processor systems. These tasks should not be designed if they are not necessary since it increases the complexity of the system significantly.
4. **The data management component.** The database management system is chosen and the databases are designed.

The four steps of the OOD phase merely concern with information (database) systems and therefore their use for the ATLAS muon read-out chamber electronics simulator will only be rudimentary.

5.2 Object oriented analysis

5.2.1 The subjects

In the previous chapters it has become clear that there are globally two types of class&objects in the simulation. The first type are the events and the eventlist. This can be divided even further in the eventlist and the types of events, i.e. the different kinds of events specified in chapter 4. The second type are the real-world objects described in chapter 2. A fourth subject will be added and it contains the environment in which the simulation runs. For a great deal this ‘environment’ is identifiable with the user interface. The collapsed subject layer is shown in figure 5.3.

5.2.2 The eventlist

The first class&object one can find in the subject ’Eventlist and events’ is the eventlist. The actual list of the event list is some list structure discussed below.

ListStructure. The list structure has to be sorted on a key: the time stamp of the events. A non-sorted list will not be a good choice because the events are handled in time order and finding the entry with the smallest stamp in such a list is rather time

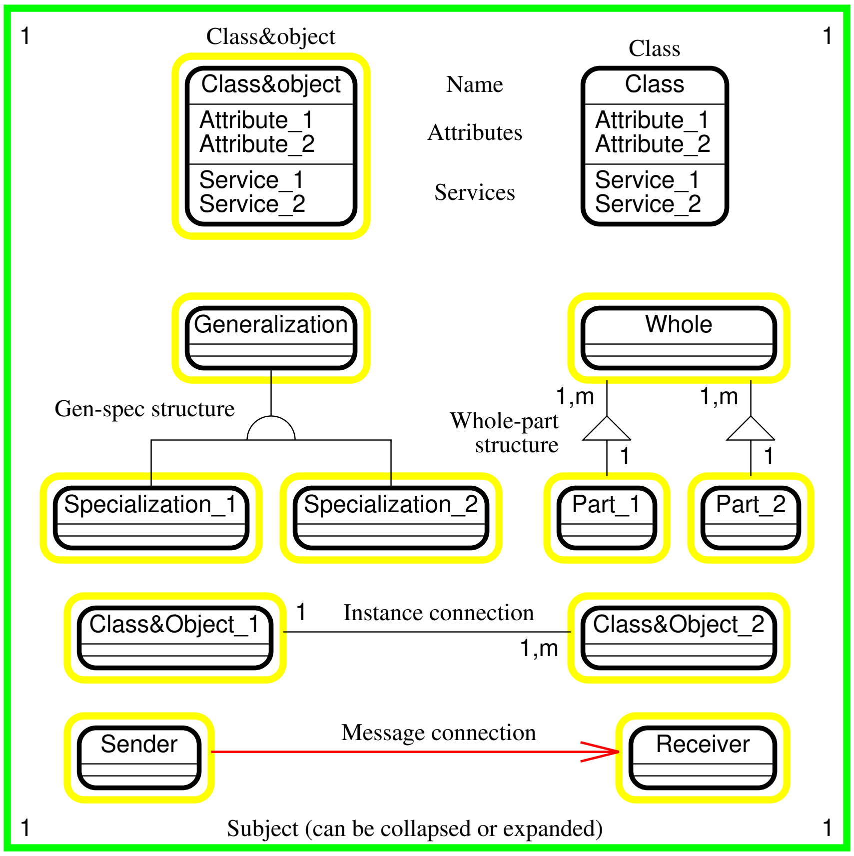


Figure 5.1: *Graphical notation of the OOA/OOD method.*

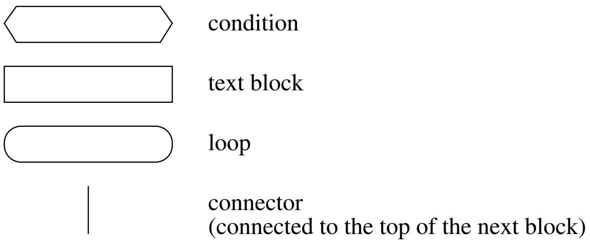


Figure 5.2: *Notation for service charts.*

Figures 5.1 and 5.2 are reproduced from [Coad, Yourdon 91a].

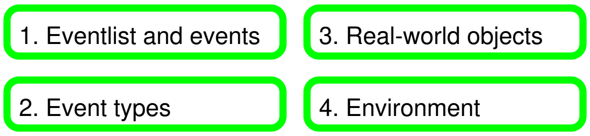


Figure 5.3: *The subject layer.*

consuming. Alternatives for this list structure are a linked list or a binary tree. A linked list is only suitable for small amounts of entries in the list since its complexity is $\mathcal{O}(n)$ where n is the number of entries in the list. The complexity of a binary tree is much better, namely $\mathcal{O}({}^2\log n)$, but in a worst case situation the complexity grows to $\mathcal{O}(n)$ as well, since the tree then behaves like a linked list with a lot of overhead. The probability of this happening however is very small. A solution for this problem is the AVL tree, which is explained in appendix A. This is a special type of binary tree that keeps the tree structure in balance. Tests however must show which type of structure is the most suitable for the ATLAS simulation program[2].

The list structure must provide services to insert and remove entries, to get the entry with the smallest key (the first node in the structure) and a test if the structure is empty. Each node in the list structure has its own key and node data. This node data will be the event. Each list structure is part of one eventlist.

EventList. The class&object 'eventlist' consists of one list structure as mentioned above. It must maintain the 'current time' as well as some status information (indicating whether the event loop is running, halted or has encountered a problem). The eventlist class&object has to provide services to run the event loop, to retrieve and handle an event in the list and to submit new ones with either an absolute time stamp (the time at which an event is to be handled) or a relative time interval (i.e. the event must be handled a specific amount of time units from the current time). The event retrieved and handled will always be the one with the smallest time stamp. The eventlist polls the user interface if the user wants to interrupt the event loop.

The service chart for the service 'run event loop' is shown in figure 5.5. The numbers in parentheses refer to the message connections shown in the OOA diagram for the subject 'eventlist' in figure 5.4. The types of events will be discussed in the section below.

Event. An event has a time stamp as attribute and offers a service to handle the event. An event is associated with one type of event, but a specific type of event can be instanciated by a number of events. The event maintains a whole-part relationship with the list structure. An event can be entered only once into the list but the list can contain zero or more events.

[2]The results of the tests are discussed in section 7.1.

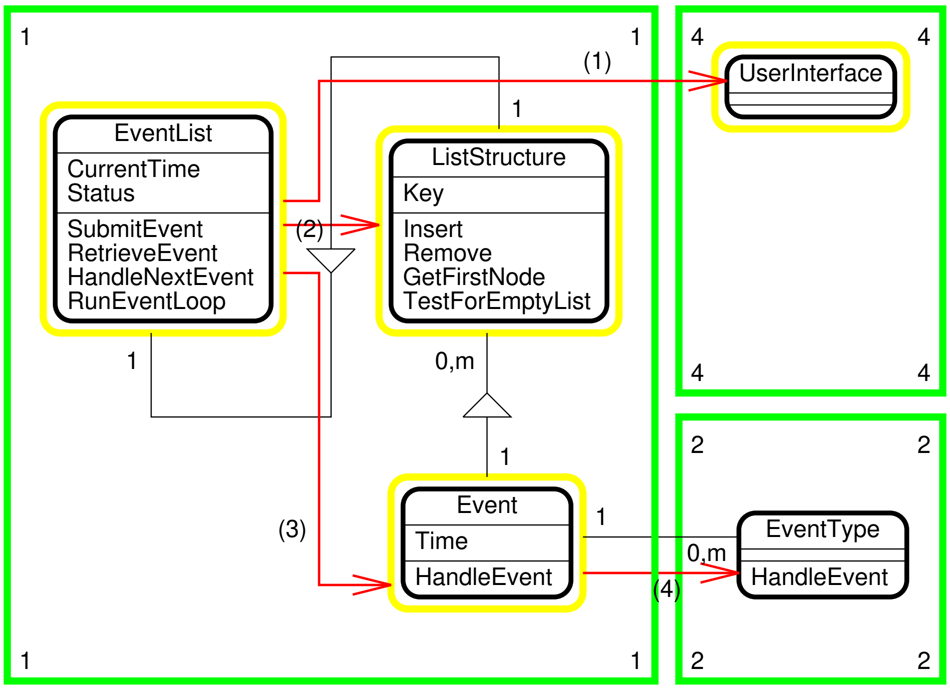


Figure 5.4: *OOA diagram of the eventlist, events and event types.*

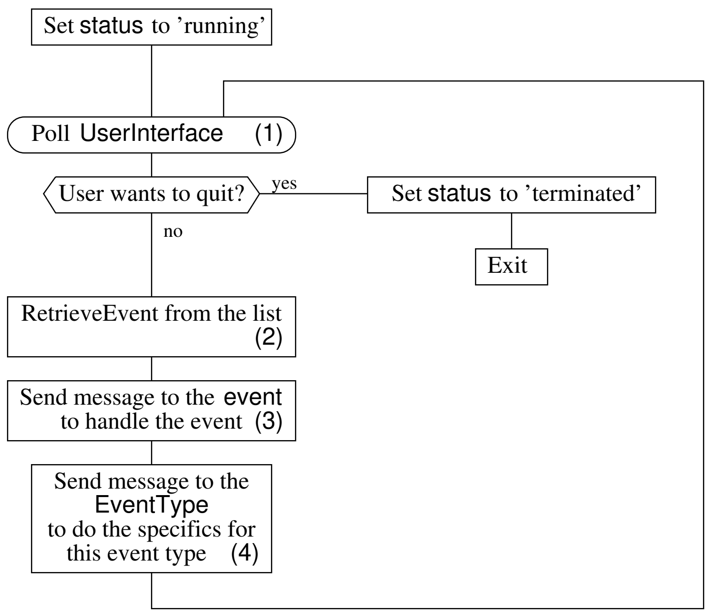


Figure 5.5: *Class&ojbect 'eventlist': the service chart for the service 'run event loop'.*

5.2.3 The events and real-world objects

The class&objects in the subject 'event types' are those events listed in table 4.1. The following class&objects can be identified: 'BX event', 'muon production event', 'muon enters detector event', 'neutron decay event', 'hit event', 'detection event', 'FIFO in event', 'FIFO out event', 'scan event', 'RAM in event', 'RAM out event', 'read RAMs event' and 'level 1 trigger event'. These class&objects will be discussed in section 5.2.5.

Furthermore the real-world objects involved with these events can be identified and they are represented by the class&objects 'muon', 'neutron' and 'particle' as well as 'tube', 'FIFO', 'layer', 'channel', 'scanner' and 'time stamp'. A class&object 'detector' can be identified implicitely too. The class&objects representing the real-world objects will be described in section 5.2.4 and are shown in figure 5.8; the class&objects representing the events are depicted in figure 5.12. These two diagrams are combined and the interactions between the class&objects in these two subjects are shown in figure 5.13.

Some abstractions

Some of the class&objects mentioned above can be filtered out because they can be abstracted to other, more convenient class&objects.

- The event BX only imposes some timing constraints. Therefore this event will not be regarded as an event type, but it will be abstracted to a real-world object and its representation class&object 'BX clock'. For the same reason the class&object 'BX event' shall not be modeled in this analysis. The class&object 'BX clock' will be discussed below.

- Each mention of a 'channel' in chapter 4 in the specification of the events concerning the scanners, actually refers to the FIFO queue of a channel. For this reason the class&object 'channel' will not be part of the design. Its responsibilities will be shared between the class&objects 'FIFO' and 'tube'. Neither will the class&object 'time stamp' be identified as a separate class. In fact, a time stamp only exist as an entry in either the FIFO queues or the RAM memories, so a new class&object 'internal register', which holds that entry, shall be introduced.

Possible other abstractions will made during the analysis of either the events resulting from the specification or the real-world objects.

5.2.4 The class&objects representing the real-world objects

Particle, Neutron, Muon. The class&object 'particle' is a pure abstract class. It does not have any attributes nor will it provide services nor will if have objects. The class&object 'neutron', which is short for 'neutron decay product', does not have any attributes or provide any services either. The class&object 'muon' has the attribute 'L_m', see (3.26), and offers the service 't_{flight}' which calculates the time it takes this specific muon to reach the detector, see (3.31). The class&objects 'neutron' and 'muon' are specializations of the class&object 'particle'.

InternalRegister. The class&object 'internal register' represents a memory location in a FIFO queue or a RAM memory. Although the memory entries in the RAMs contain more data than the entries in the FIFO queue (namely the scanner and channel number from which the data was read), both these entries will be abstracted to and represented by the same class&object. The 'internal register' has two attributes, the data entry and the type of data. The latter serves for debugging purposes and contains information and a representation of the type of particle which resulted in a hit, which in its turn resulted in placing a time stamp in the queue. This representation will not necessarily be the actual object 'muon' or 'neutron', but an identification of either one will suffice.

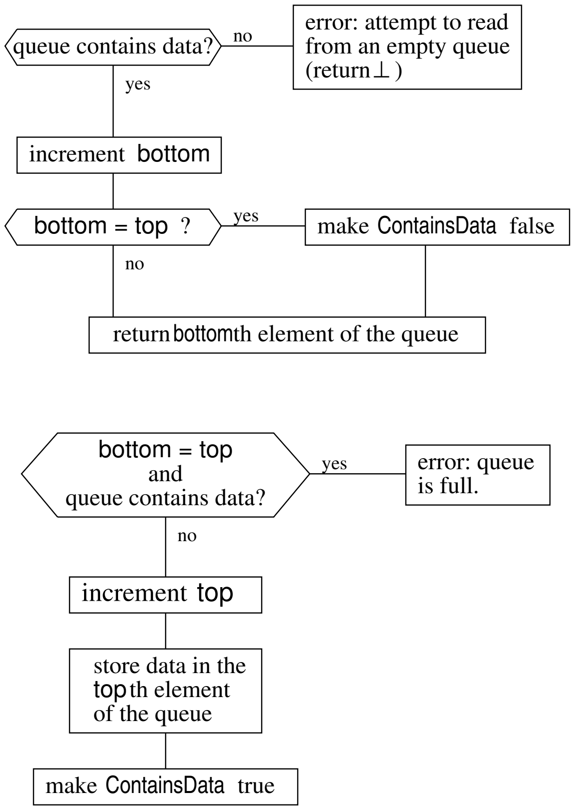
RAM. The class&object 'RAM' contains a number of memory entries, represented by the class&object 'internal register'. The number of these entries is indicated by a constant number represented by the name 'RAM size'. The 'RAM' offers the services 'get' and 'put' to read and write entries in the memory. The service 'number in RAM' returns the number of valid entries in the RAM. The service 'location of the smallest entry' returns the memory location which contains the smallest, i.e. oldest time stamp. The service is used by the service 'put' to find an – invalidated – entry which can be overwritten.

The definition of a valid entry is an entry whoms time stamp is younger than a set boundary: the maximum RAM life time. The class&object 'RAM' does not have any attributes.

FIFO. Analogously to the class&object 'RAM', the class&object 'FIFO' contains a number of memory entries represented by 'internal register' as well. The number of such entries is represented by the name 'FIFO size'. The class&object 'FIFO' has the attributes 'top' and 'bottom' which indicate the data element that was last and the element that was first entered respectively. The attribute 'contains data' specifies if the queue indeed contains data. The purpose of these attributes will be made clear in the service charts for the services 'put' and 'get' in figure 5.6. Besides these, 'FIFO' also offers the service 'number in FIFO'.

Scanner. The class&object 'scanner' contains exactly one 'RAM' (and each 'RAM' belongs to precisely one 'scanner'). The 'scanner' has as an attribute its ID (an integer number starting with 0 for the scanner in the bottom layer at the end of the detector, i.e. the most distant scanner from the interaction point, and that number is incremented in one layer, continuing at the end of the next layer and so on). It offers the service 'scan' which does the actual scanning. This service is explained below. The scanner has a connection to a channel, which is represented by the instance connection to the class&object 'FIFO'.

The class&object 'scanner' has message connections with the class&objects 'FIFO' and 'RAM'. The service 'scan' sends a message to the 'FIFO' to get a data entry from the queue. The service 'scan' then sends a message to the 'RAM' to store that data entry into the memory, using its service 'put' described above. The algorithm to use for this service is indicated by the attribute 'algorithm', see (4.29). The service chart for



(a)

(b)

Figure 5.6: *Class&object 'FIFO': service charts for 'get'* (a) *and 'put'* (b).

the service 'scan' is shown in figure 5.7.

Tube. The class&object 'tube' contains exactly one 'FIFO'. The 'tube' has an attribute 'tube number' which is the relative tube number in the layer. The attribute 'last hit' indicates the time at which the tube got its last hit. The time is read through the message connection 5 from the class&object 'eventlist'. It provides the services 'L' and 'R' which calculate $L(t, l)$ and $R(t, l)$ (see (3.33) and (3.34)) respectively. The service 't drift' calculates the drift time (3.45) of a particle, hence the instance connection with the class&object 'particle'. It also provides the service 'neighbour' which non-deterministically chooses one of the six neighbours of the tube, see the algorithm in figure 4.2.

Layer. A layer consists of $\#tubes$ tubes and N scanners. This is represented by the whole-part relationships between the class&object 'layer' and the class&objects 'tube' and 'scanner' respectively. One attribute will be defined: the 'layer number'.

Detector. The class&object 'detector' contains n layers. This is indicated by the whole-part relationship to the class&object 'layer'. It provides one service 'in T hit' which checks through message connections 7 and 8 if a tube is in $\mathcal{T}_{hit}$, see definitions

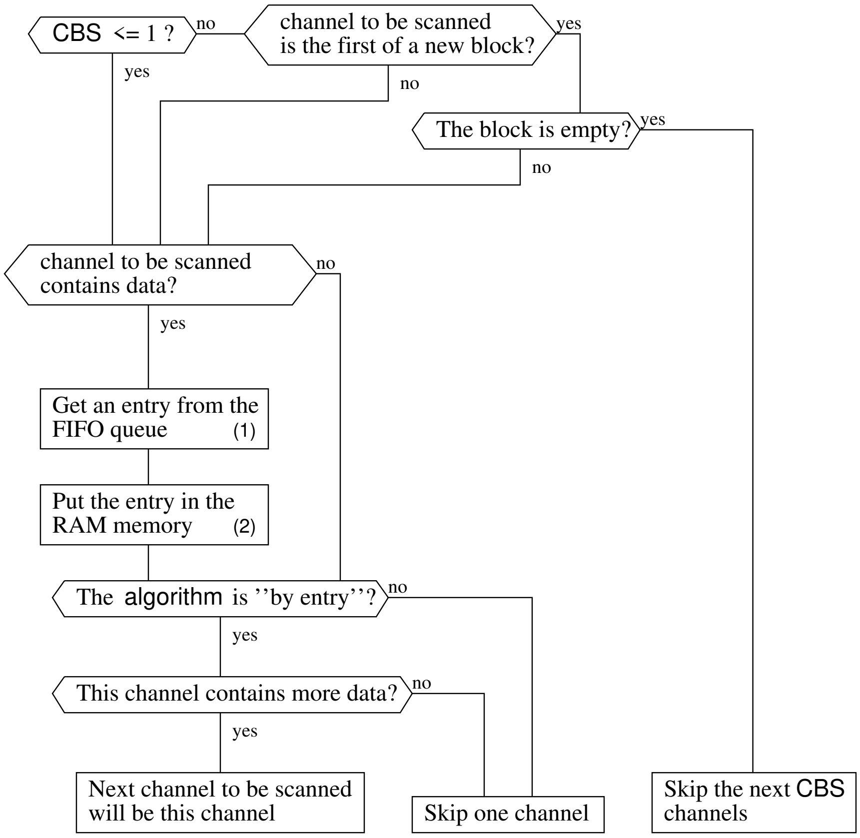


Figure 5.7: *Class&object 'scanner': service chart for the service 'scan'.*

(3.36) and (3.37).

The class&object representing the clock mechanism

BX_Clock. The class&object 'BX clock' has only one service: 'read BX'. This service sends a message to the class&object 'eventlist' to get the current time and derives the current BX from that by taking the floor from that time divided by the bunch spacing, refer to (4.10).

TDC_Clock. The class&object 'TDC clock' also has only one service: 'read TDC'. This service sends a message to the class&object 'eventlist' to get the current time and derives the current TDC bin from that. The time is taken modulo the bunch spacing. From the remaining fraction the TDC bin is calculated.

$$TDC\ bin \quad = \quad \left\lfloor \frac{current\ time \bmod bunch\ spacing}{bunch\ spacing} \cdot TDC\ bins\ per\ BX \right\rfloor \qquad (5.1)$$

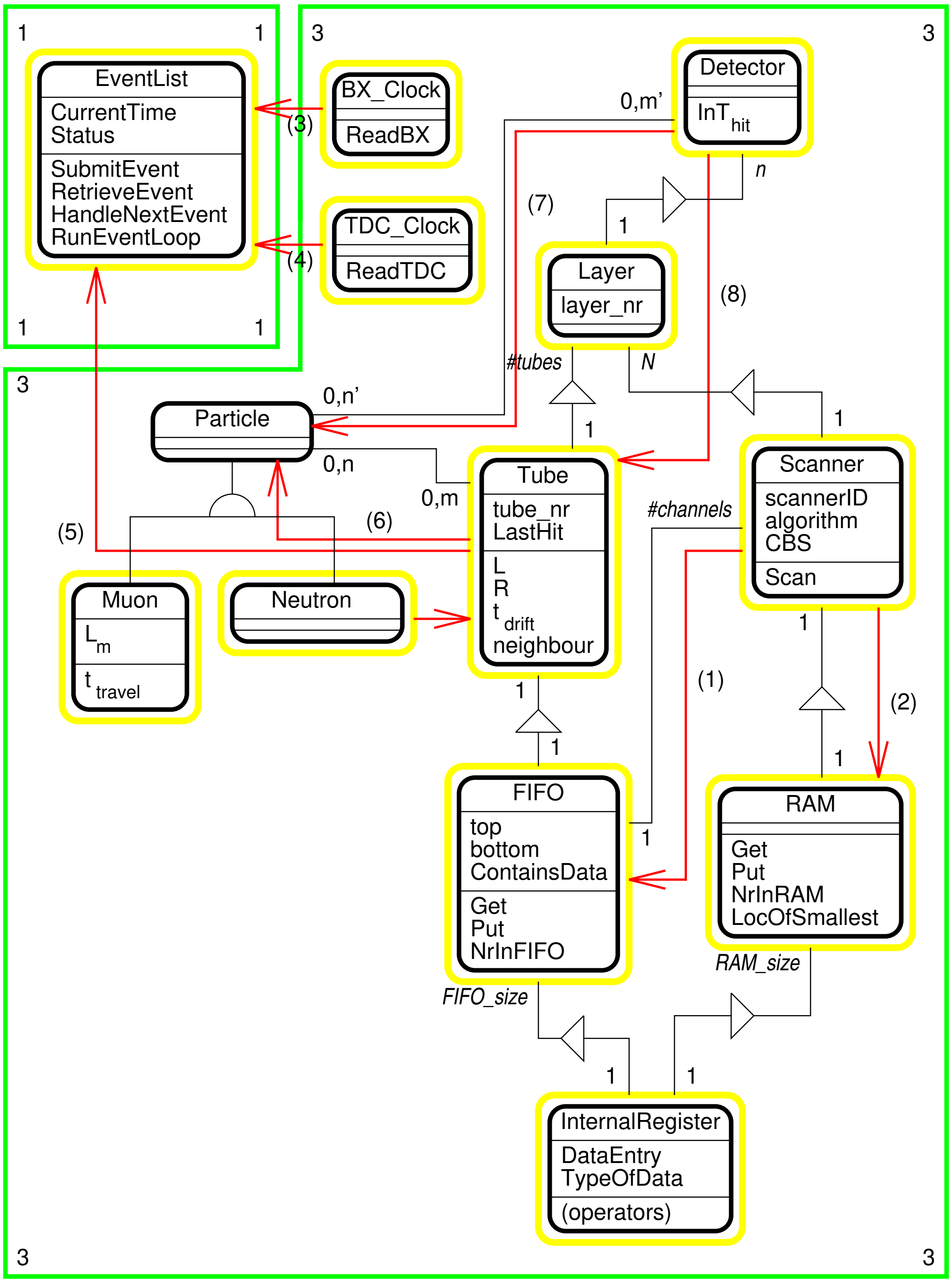


Figure 5.8: *OOA diagram for the real-world objects.*

5.2.5 The class&objects representing the events

EventType. An abstract class 'event type' will be introduced. It is the generalization of each of the types of events. It has no attributes, but it provides the service 'handle event'. This service will be inherited by each of the specilizations of this class and functionality is added to the service to do the tasks that are specific for the type of event.

MuonProductionEvent. The class&object 'muon production event' is the representation of the event $muon\_production$. It has an instance connection (and a message connection because this class&object creates the 'muon') to the real-world object 'muon' representing the muon that is produced. Of course, it inherits the service 'handle event' from the class&object 'event type'. This service submits a new 'muon production event' and with a certain possibility a 'level 1 trigger event'. The time interval at which the newly submitted 'muon production event' will be handled is an exponentially distributed random number with mean μ'_μ. The probability that a 'level 1 trigger event' will be submitted is $L1\_prob$. These parameters are discussed in the previous chapter(s). The 'muon production event' also submits a 'muon enters detector event'.

DetectorEnterEvent. The class&object 'muon enters detector event' is derived from the event $detector\_enter$. It has an instance connection with the real-world object 'muon'. This 'muon' is the same object as the 'muon' created by 'muon production event' which submitted this 'muon enters detector event'. For each tube that is hit by the 'muon', the service 'handle event' will submit a 'hit event' to the event list. For this reason a message connection exists to the class&object 'detector' to determine which tubes are hit.

NeutronDecayEvent. The class&object 'neutron decay event' represents the event $neutron\_decay$. It has an instance connection to the class&object 'neutron', which represents the decay product(s) of the decaying neutron, and an instance connection with the class&object 'tube', indicating in which tube the neutron decayed. The service 'handle event' submits a 'hit event' for that tube and, with a probability of 1%, a 'hit event' for one the neighbouring tubes. This neighbour shall be chosen in a non-deterministic way, see the algorithm shown in figure 4.2, which is indicated by the message connection to the class&object 'tube'. Finally a new 'neutron decay event' will be submitted for the same tube at a random time interval which has a exponential distribution with mean μ'_{n}.

HitEvent. The event hit_event is represented by the class&object 'hit event'. This class&object has an instance connection to the class&object 'tube', indicating the tube in which a particle caused a hit. It has the attribute 'insensitive time' which represents the width of a pulse (in ns) induced by one hit. See figure 5.9. The service 'handle event' updates the attribute 'last hit' of the class&object 'tube', for which a message connection exists to the class&object 'tube'. Usually it also submits a 'detection event', unless a hit is followed by another hit with an intermediate time interval of less than the time period indicated by the attribute 'insensitive time'. In this case the second hit will not result in a detection and the service 'handle event' will *not* submit a 'detection event'.

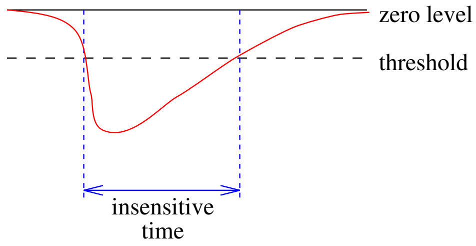


Figure 5.9: *Class&object 'hit event': the attribute 'insensitive time' represents the width of (one) pulse induced by a hit.*

DetectionEvent. The class&object 'detection event' represents the event $detection$. Its service 'handle event' submits a 'FIFO input event'. It has the attribute 'kind of particle', which is intended for debugging purposes (see the description of the class&object 'internal register').

FIFO_InEvent. The event $FIFO\_in\_event$ is represented by the class&object 'FIFO input event'. It has the attribute 'kind of particle' which is set by the class&object 'detection event' and it indicates what kind of particle resulted in the detection of a hit. The service 'handle event' reads the current BX time and TDC bin, hence there is a message connection to the class&object 'BX clock' and to the class&object 'TDC clock', and it stores the time stamp in the FIFO queue, which is represented by an instance and a message connection to the class&object 'FIFO'.

FIFO_OutEvent, RAM_InEvent. The events $FIFO\_out\_event$ and RAM_in_event have become obsolete with the existence of the service 'scan' of the class&object 'scanner', which performs the tasks of these two events (see figure 5.10).

ScanEvent. The class&object 'scan event', which represents the event $scan\_event$, has a message connection (and an instance connection) with the class&object 'scanner' to use its service 'scan'. It has the attribute 'channel' which indicates the channel scanned most recently. The service 'handle event' sends a message to the class&object 'scanner' which then executes the service 'scan'. The reply on the message is a modification of the attribute 'channel'. See the service chart in figure 5.7. A new 'scan event' will be submitted at a time interval of 25 ns.

Level_1_TriggerEvent. The class&object 'level 1 trigger event' represents the event $level\_1\_trigger$. It has the attribute 'BX stamp' which indicates the bunch crossing to which the level 1 trigger belongs, i.e. the bunch crossing at which this 'level 1 trigger event' was submitted to the 'eventlist'. For this reason there is a message connection to the class&object 'BX clock' which reads the BX time when an 'level 1 trigger event' object is created. The service 'handle event' submits a 'RAM output event', which in fact is the combination of the events RAM_out_event and $read\_RAMs\_event$ from the specification.

RAM_OutEvent. The class&object 'RAM output event' has the attributes 'BX

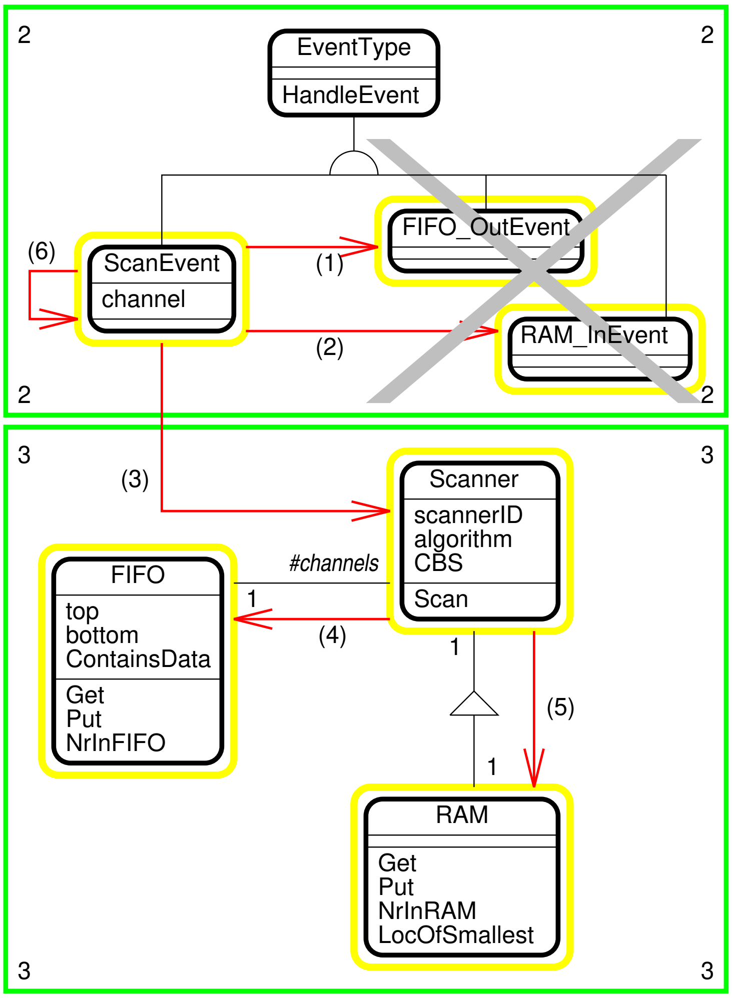


Figure 5.10: *OOA diagram for the class&objects 'scan event', 'FIFO output event' and 'RAM input event'. The last two class&objects have become obsolete since the scan procedure is performed by the service 'scan' of the class&object 'scanner' (message connection 3 replaces message connections 1 and 2).*

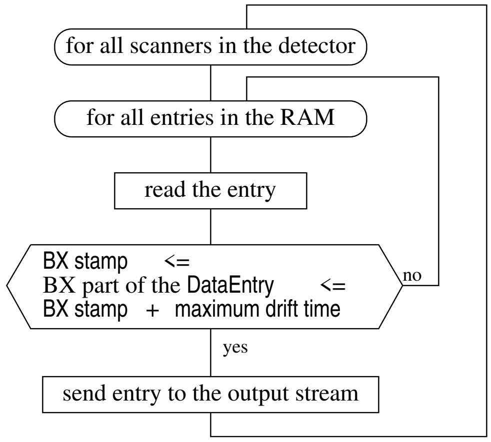


Figure 5.11: *Class&object 'RAM output event': service chart for the service 'read RAMs'. The class&object 'output stream' resides in the subject 'environment'.*

stamp' and 'maximum drift time'. It represents the events RAM_out_event and $read\_RAMs\_event$. Besides the inherited service 'handle event', the class&object offers the service 'read RAMs', which uses an instance connection and a message connection to the class&object 'detector'. These connections are with that class&object since the service 'read RAMs' polls *all* scanners in the detector. The operation of the service 'read RAMs' is explained in the service chart in figure 5.11.

5.2.6 The class&objects of the environment

Before the class&objects of the environment are analyzed, a survey will be made *what* to expect from the environment. The following aspects are involved:

- A view on the current fill of the FIFO queues. This will only be applicable when a graphical display is used.
- A clear view of the statistics. To achieve this, statistics need to be gathered of the maximum fills of the FIFO queues and the RAM memories as well as the number of data entities in the output stream when a level 1 trigger is issued. The FIFO and RAM statistics have to be recorded locally (i.e. per FIFO and RAM) and globally.
- The ability to display the maximum fills of the FIFO queues and the RAM memories and the maximum number of data entities in the output stream.
- The ability to collect the statistics in a file.
- The ability to collect the data entries in the output stream after a level 1 trigger.

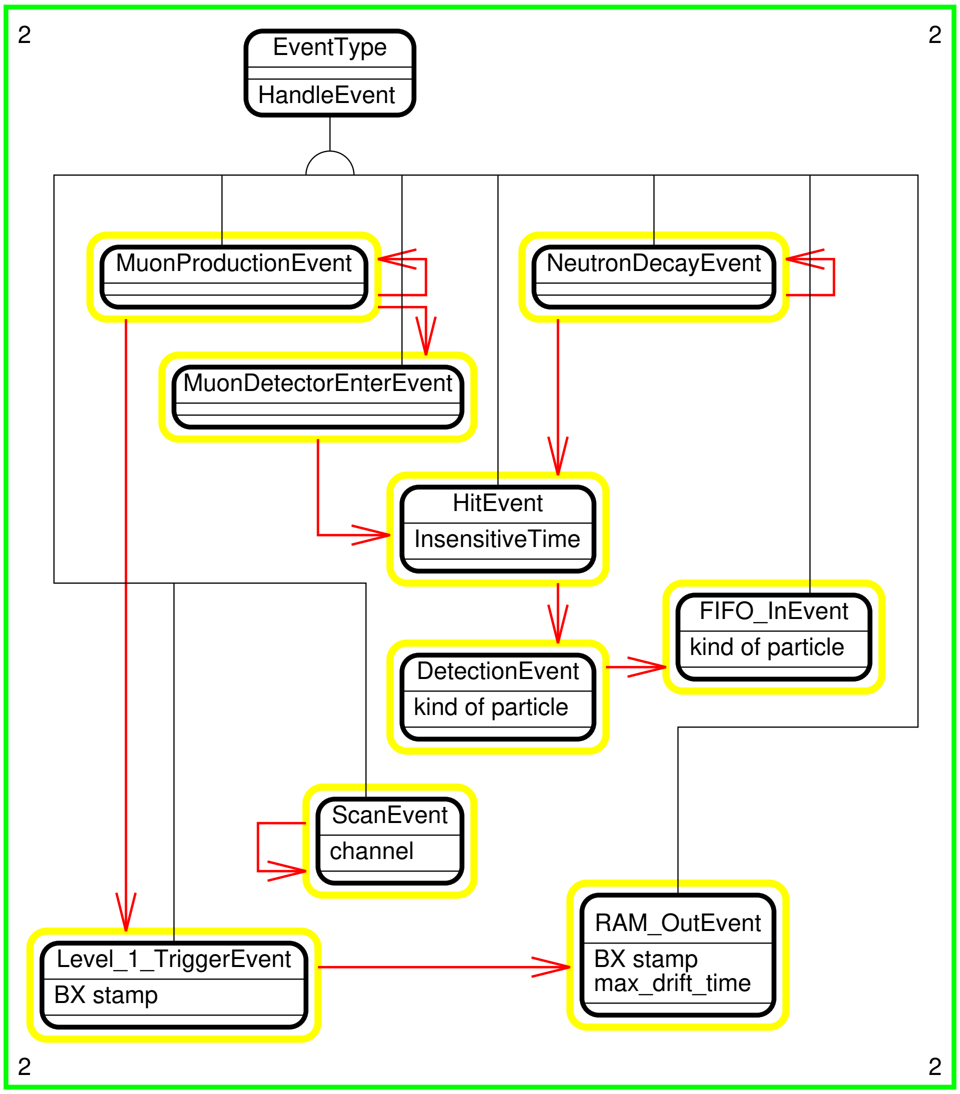


Figure 5.12: *OOA diagram for the event types.*

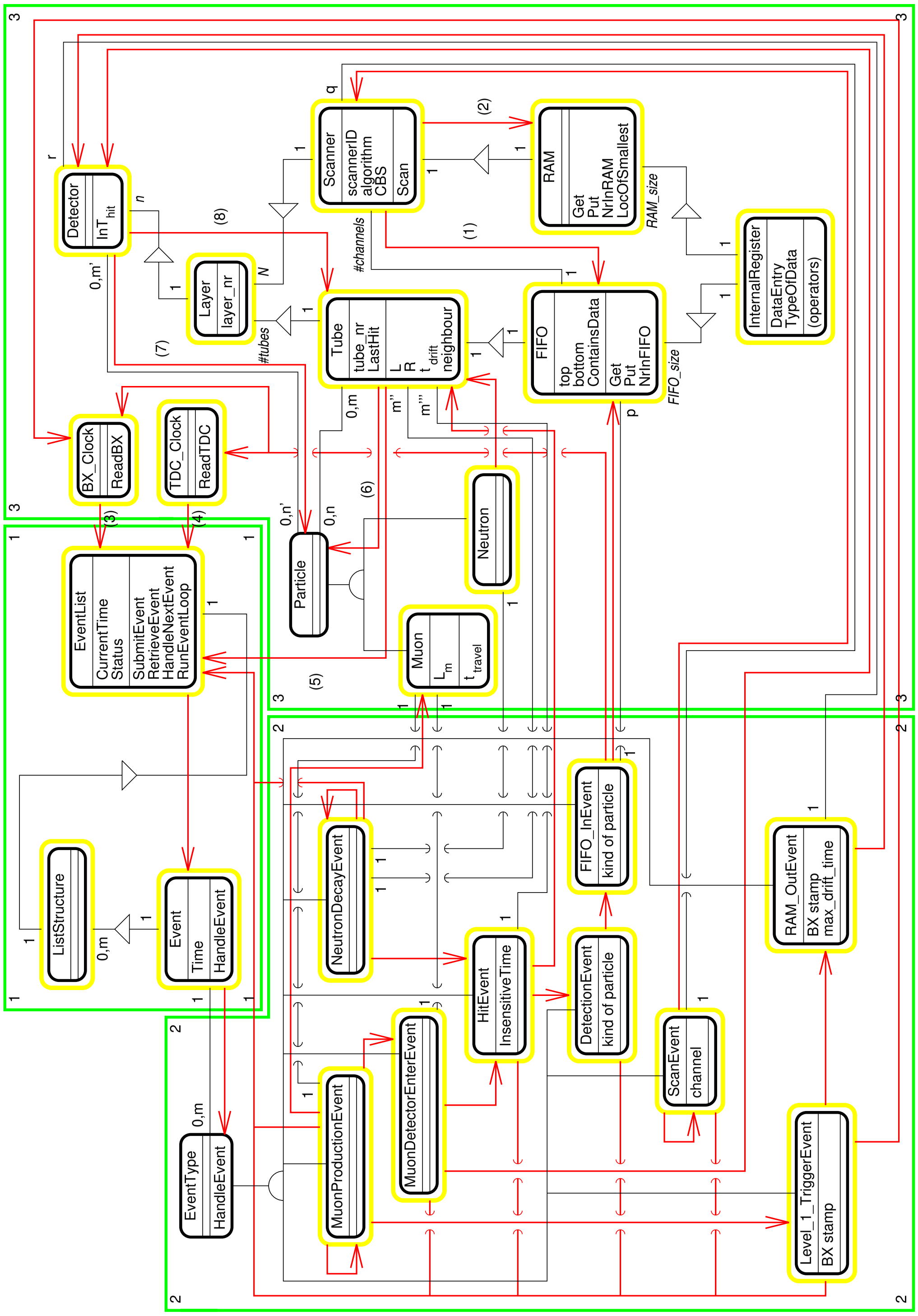


Figure 5.13: *The complete OOA diagram of the subjects 'eventlist', 'event types' and 'real-world objects' and the interactions between the class&objects in those subjects.*

- The ability to fully customize every possible parameter of the simulation. This ranges from the size of the detector, the muon and neutron rates to the sizes of the memories.
- Progress display; the ability to display the current (BX) time.
- The ability to terminate the event loop.

Class&objects

In analysing the aforementioned requirements, the following class&objects can be identified:

- A screen display, represented by the class&object 'screen'.
- A manner to intercept user events, represented by the class&object 'user events'.
- A method to set the parameters of the simulation. The class&object 'parameters' will be introduced for this purpose.
- The collection of the statistics will be taken care of by the class&object 'statistics'. It has three parts: 'FIFO statistics', 'RAM statistics' and 'level 1 statistics'.
- An 'event' which displays the current time at regular intervals. For this task the class&object 'timer event' will be introduced.

UserInterface. The class&objects 'screen' and 'user events' can be seen as a specialization of the class&object 'user interface'. This will be an abstract class without attributes or services. It was briefly introduced in figure 5.4.

Screen. The class&object 'screen' has the attributes 'X size' and 'Y size', indicating the physical size of the screen in pixels and the attributes 'X virtual' and 'Y virtual', indicating the size of the screen in virtual coordinates. These virtual coordinates will be used by the simulation program. It has the services 'Xv2p', 'Yv2p', 'Xp2v' and 'Yp2v', which are abbreviations for the virtual to physical and vice versa conversions.

To draw objects on the screen, the services 'draw line', 'move to', 'draw rectangle', 'draw text' as well as 'set fill style', 'set line style' and 'set colour' are offered.

The class&object 'FIFO' has a message connection to the class&object 'screen' to be able to display its current fill.

UserEvents. The class&object 'user events' intercepts the events (not to be mistaken for the events that are handled in the simulation) resulting from intervention by the user. Usually this is a key the user pressed on the keyboard or a button on the mouse. It offers the service 'get user event'.

Parameters. The class&object 'parameters' offers no services, but it has the following attributes:

Location and size of the detector: 'L', L in (3.27); 'R', R in (3.28); 'tube length', l in (3.8); 'tube diameter', d in (3.9); 'wire distance', d_T in (3.11); 'layer distance',

d_L in (3.10); 'number of channels', $\#channels$ in (3.12); 'number of scanners' per layer, N in (3.13); 'number of layers', n in (3.14).

Muon and neutron decay rates: 'muon rate', 'neutron decay rate' and the derived attributes 'muon distribution' (μ'_μ in (3.25)) and 'neutron distribution' (μ'_{n} in (3.49)), as well as the probability that a neutron decay product produces a hit in a neighbouring tube: 'neighbour hit'.

Drift times: 't_drift a' and 't_drift b' representing a and b in (3.44).

Timing and clock parameters: 'bunch spacing', 'TDC bins per BX', 'level 1 latency', usually 2000 ns, and the 'insensitive time' of the discriminator (the width of the pulse shown in figure 5.9) as well as 'level 1 probability': $L1\_prob$.

Scanners, channels and memories: 'scanner algorithm', 'channel block size' (CBS), 'FIFO size', 'RAM size' and the number of bunch crossings a scanner needs to scan one channel: 'scan time'.

File options: To turn on or off the different file options the following attributes are provided: 'log file', to write a log of all produced muons; 'RAM output', to write the entries in the RAM memories after a level 1 trigger; 'FIFO statistics', 'RAM statistics' and 'Level 1 statistics', to write the files with the statistical information.

Display options: To turn on or off the display of the current time (the attribute 'display time') and the display of the latest statistics (the attribute 'display statistics').

Statistics. The class&object 'statistics' has one service: 'get maxima'. The service chart is shown in figure 5.14.

FIFO_Statistics. The class&object 'FIFO statistics' has the attributes 'maximum fill' and 'overflow detected', which indicates – yes or no – if an overflow is detected in the FIFO queue. It offers the services 'update' and 'write'. The class&object 'FIFO' has an instance connection and a message connection to this class&object which is used by the 'FIFO' to update the statistics (through the service 'update'). The service 'write' writes the (current) statistics to a file.

RAM_Statistics. The class&object 'RAM statistics' has the attributes 'maximum fill' and 'overflow detected'; similar to the class&object 'FIFO statistics'. It offers the services 'update' and 'write'. The class&object 'RAM' has an instance connection and a message connection to this class&object.

Level_1_Statistics. The class&object 'level 1 statistics' has the attribute 'maximum words' indicating the maximum words in the output stream after a level 1 trigger. It offers the service 'update' and 'write'. The class&object 'RAM output event' has a message connection to this class&object, using the service 'update' to update the attribute 'maximum words'. The class&object 'level 1 trigger event' must signal the class&object 'level 1 statistics' that a new sequence of output has started so a distinction can be made between the subsequent messages from the class&object 'RAM output event'.

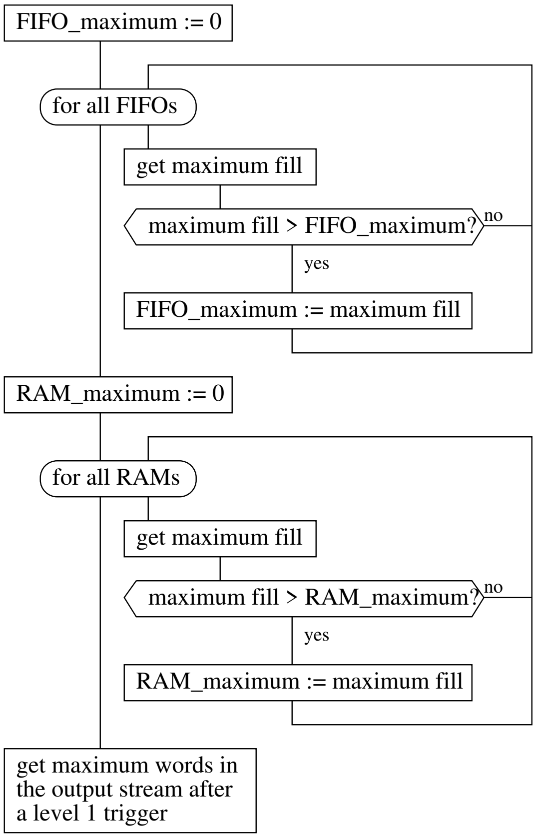


Figure 5.14: *Class&object 'statistics': the service chart for the service 'get maxima'.*

TimerEvent. The class&object 'timer event' is a specialization of the class 'event type'. For this reason, this class&object will be part of the subject 'event types' and *not* be a part of the subject 'environment'. It has the attribute 'interval' indicating the interval between the updates of the time display. The inherited service 'handle event' displays the time and submits a new 'timer event' at an interval indicated by the attribute. It has a message connection to the class&object 'BX clock' to read the current BX time and a message connection to the class&object 'screen' to display the current BX time.

5.3 Object oriented design

5.3.1 The problem domain component

Improvements with regard to the analysis

The analysis in the previous section can be improved on several points. These points are:

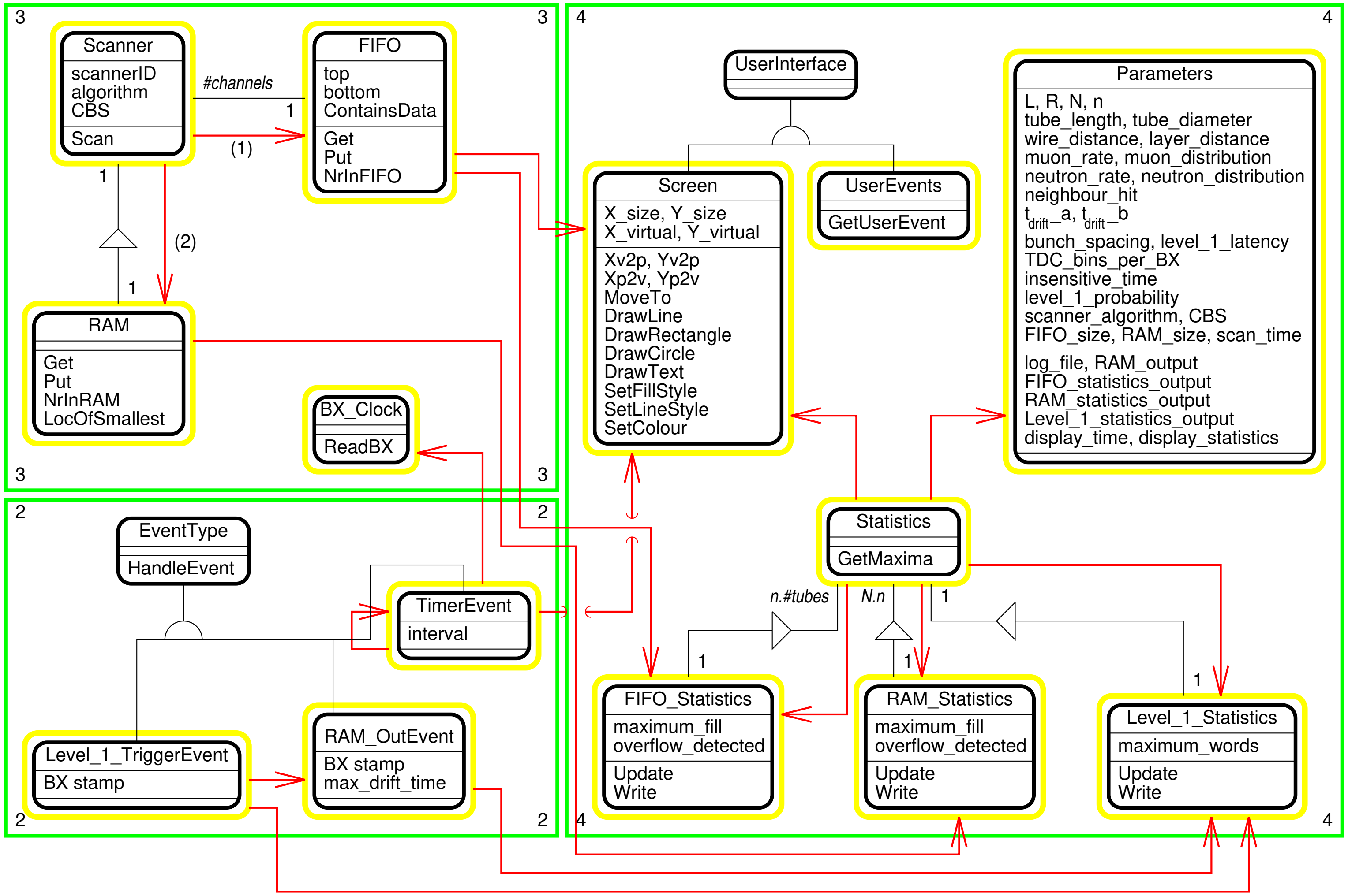


Figure 5.15: *OOA diagram for the environmental objects.*

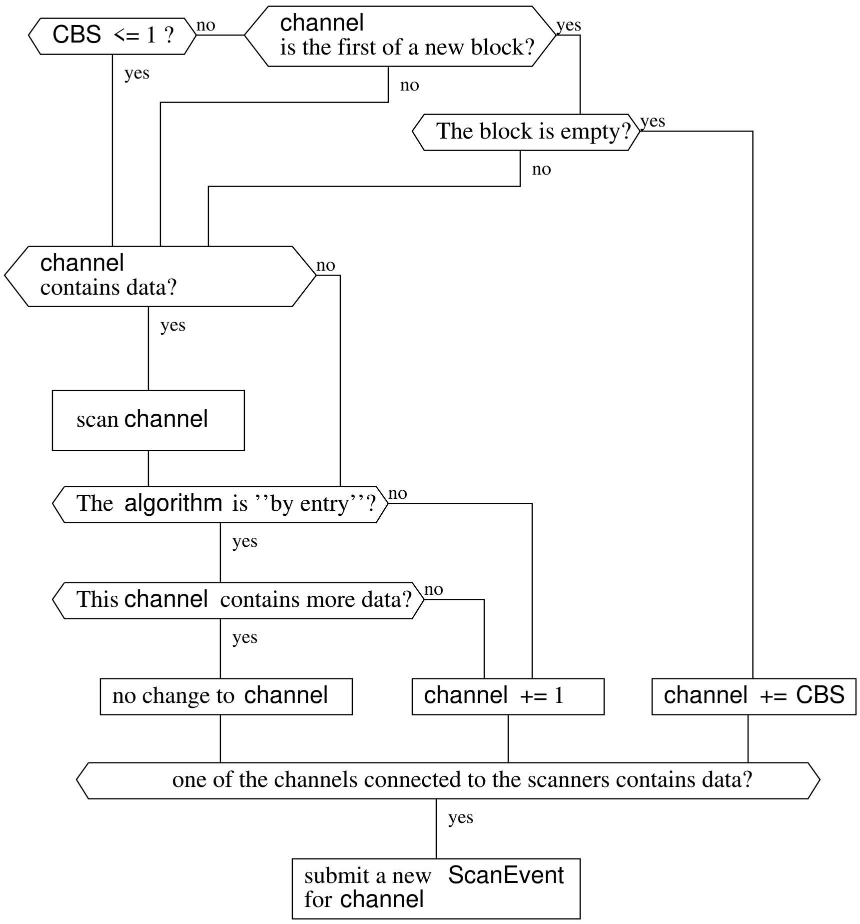


Figure 5.16: *Class&object 'scan event': service chart for the service 'handle event'.*

- A 'scan event' is submitted to the eventlist at every 25 ns. This could be very time consuming, and especially when the queues are empty this is a waste of computer time.

  A solution to this problem is to introduce a scheduling mechanism for the scanners so they will only be activated when one of the queues receives data (i.e. a 'FIFO in event' is handled for that queue). For this, the next class&objects need to be changed:

  ScanEvent. The meaning of the attribute 'channel' will slightly change. It indicates the channel that *will* be scanned. Also the service 'handle event will be modified. The actions taken by the service 'scan' of the class&object 'scanner' in the object oriented analysis will now be performed by the service 'handle event'. Its service chart is shown in figure 5.16.

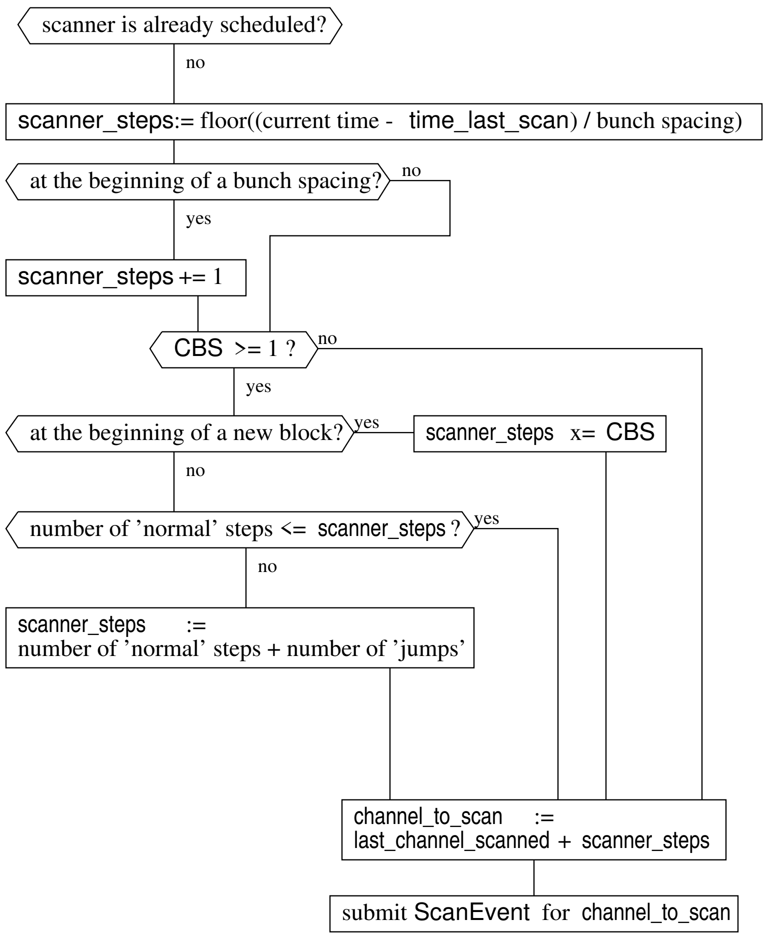


Figure 5.17: *Class&object 'scanner': service chart for the additional service 'schedule scanner'.*

Scanner. The service 'scan' will be stripped to perform only the actual scanning (i.e. to get a data entry from the FIFO queue and put it in the RAM memory). Two attributes will be added: 'time last scan', which indicates the time (which is read from the class&object 'eventlist') at which the last scan action was performed, and 'channel last scanned', indicating *which* was the last channel scanned. One service will be added: 'schedule scanner'. The class&object 'FIFO in event' sends a message to this service so, when in rest, the scanner will be scheduled again. The service chart for 'schedule scanner' is shown in figure 5.17. The scanner will remain scheduled as long as one of the queues in the channels connected to it contains data.

FIFO_InEvent. The service 'handle event' will be slightly modified and sends a message to the service 'schedule scanner'.

- The class&object ’neutron’ is not really necessary since if performs no tasks of any interest nor of any importance. Therefore it shall be removed from the design. The class&object ’neutron decay event’ sends a message directly to the class&object ’tube’ to determine the neighbour, if any, that might be affected by a specific neutron decay.

- The attribute ’insensitive time’ appearing in the class&object ’hit event’ always has the same value as the attribute in the class&object ’parameters’. Therefore it shall be removed from the class&object ’hit event’ and be replaced by a message connection to the class&object ’parameters’.

- The class&objects ’hit event’ and ’detection event’ will be combined to one class&object ’detection event’. Its service ’handle event’ will check if a channel is sensitive and if so, a ’FIFO input event’ will be submitted to the eventlist.

- The class&object ’user events’ represents in practice the same physical device as the class&object ’screen’. Therefore, that class&object will be incorporated in the class&object ’screen’. The ‘need’ for a common generalization ’user interface’ disappears as well so this class&object can be removed from the design too.

Reuse of existing class&objects

The only class&object in the analysis which is likely to be created previously, is the class&object ’list structure’. As discussed before, a usuable implementation is the binary tree (either a generic binary tree or a special type of tree, like the AVL tree). It will be useful to check if an existing implementation can be used in the simulation program.

Adaption to the target language C++

No special actions will be taken to adapt the design for the target language, since each of the concepts of the OOA/OOD method is present in the language C++, see [van den Goor 92]. The concepts in the OOA/OOD method and their equivalents in the language C++ are listed in table 5.1. However, there will be one modification made to the class&object ’parameters’. Since the attributes in this class&object are used throughout the simulation program, they will be implemented as global variables, thus preventing excessive references to the class&object ’parameters’. The class&object will *not* be removed from the design for reasons discussed in the next section.

Order of creation of the objects

The ensure correct execution of the resulting simulation program, the order of creation of the objects has to be specified:

1. The creation of (one) object of the class ’parameters’. The actual parameters will be read from the command-line (i.e. the values for the parameters will be entered

Concept in OOA/OOD	Equivalent in C++
class	class (or struct)
class&object	class (or struct) and instantiations of that class
class name	class identifier
attribute	data member
service	method (function member)
gen-spec structure	inheritance (including multiple inheritance)
generalization	base class
specialization	derived class
whole-part structure	pointer(s)
message connection	function call
instance connection	pointer(s)

Table 5.1: *The concepts in the Object Oriented Analysis/Object Oriented Design method of Coad and Yourdon and the equivalents of those concepts in C++.*

by the user when starting the program), although default values will have to be provided for the convenience of the user.

2. The creation of (one) object of the class 'screen'.

3. The creation of (one) object of the class 'detector'. This in its turn will create the objects of the classes 'layer', 'tube', 'scanner', 'FIFO', 'RAM' and 'internal register'.

4. The start of the simulation, beginning with the creation of one 'muon production event' at the beginning of the time scale (i.e. time = 0 ns) and for each tube one 'neutron decay event' at random intervals from the start of the time scale.

5.3.2 The human interaction component

This part of the design will only the rudimentary, since the only interaction with the user are changes the user wishes to make in the default parameters and a way to terminate the simulation program.

To make the entry of parameters more comfortable, the use of a parameters file will be introduced, including a default parameter file which will always be read. The parameter file can be used to set different types of detectors, or to make persistent changes to the default values.

The program will be run on different types of hardware. This mainly affects the class&object 'screen'. For this reason, some specializations of that class&object will be made, with regard to the target hardware.

In the hardware, the distinction will be made between personal computers running MS-Dos and workstations running the Unix™ operating system or similar. On both types of hardware a generic, text based TTY screen will be available. On MS-Dos machines the program must be able to run using the different types of video hardware[3]. On Unix™ based machines the program must be equiped to run using the X Windows system.

For this reason, the class&object 'screen' has three specilizations. These are 'TTY screen', 'BGI screen' and 'X screen'. The class&object 'BGI screen' has the additional attribute 'type of display', indicating the specific display adapter (e.g. in the Borland Graphics Interface library drivers for CGA, EGA, VGA, Hercules video adapters etc. are available).

5.3.3 The data management component

The simulation will produce a number of output files, all of which will be normal, readable text files. The following files can be generated:

1. Files containing statistical information:
   (a) Statistical information about the fill degrees of the FIFO queues.
   (b) Statistical information about the fill degrees of the RAM memories.
   (c) Statistical information about the number of words (data entries retrieved from the RAM memories) after a level 1 trigger event.
   (d) A log file which will be written when the program is terminated: the on-exit log file. This file will contain the global maxima of the fill degrees of the FIFO queues and the RAM memories and the maximum number of words in the output stream after a level 1 trigger. The file will also contain the parameter settings under which the program was run.
2. A file which collects the output stream after a level 1 trigger. The data entries that are retrieved from the RAM memories when a trigger is issued will be stored in this file.
3. A log file to which the characteristics of a muon production are written.

5.3.4 Additional constraints in the human interaction component

The simulation program must provide the following, additional features:

- Every possible setting of the program must be allowed to be changed through the command-line options.
- For debugging purposes, an option must exist to show the scanners graphically.

[3]The Borland C++ compiler is shipped with libraries which handle the different types of graphical displays using the so-called Borland Graphics Interface (BGI).

5.3.5 The task management component

Because the X Windows system (as well as other windows systems) maintain their own event loop, problems might appear with the event loop which of the simulation. To solve these problems, two solutions can be provided.

The first solution is to catch the X windows events within the event loop of the simulation. However, the risk exists that this can be time consuming and the complexity of the design can grow enormously. Therefore this solution is not feasable.

The second solution is the introduction of a separate display driver. This program will communicate with the simulation program through Unix sockets (interprocess communication channels). This option also creates room to implement window manager specific display drivers, each of which having the look and feel of the environment in which it is run. The display driver will be executed by the simulation program so this fact will be transparent for the user. To handle this feature, two attributes will be added to the class&object 'X screen' which are needed to set up a communication channel through a socket, namely 'port number' and 'hostname'. The service 'make connection' makes a connection to the display driver program.

The details of communications through a socket are discussed in chapter 6 which handles the implementation.

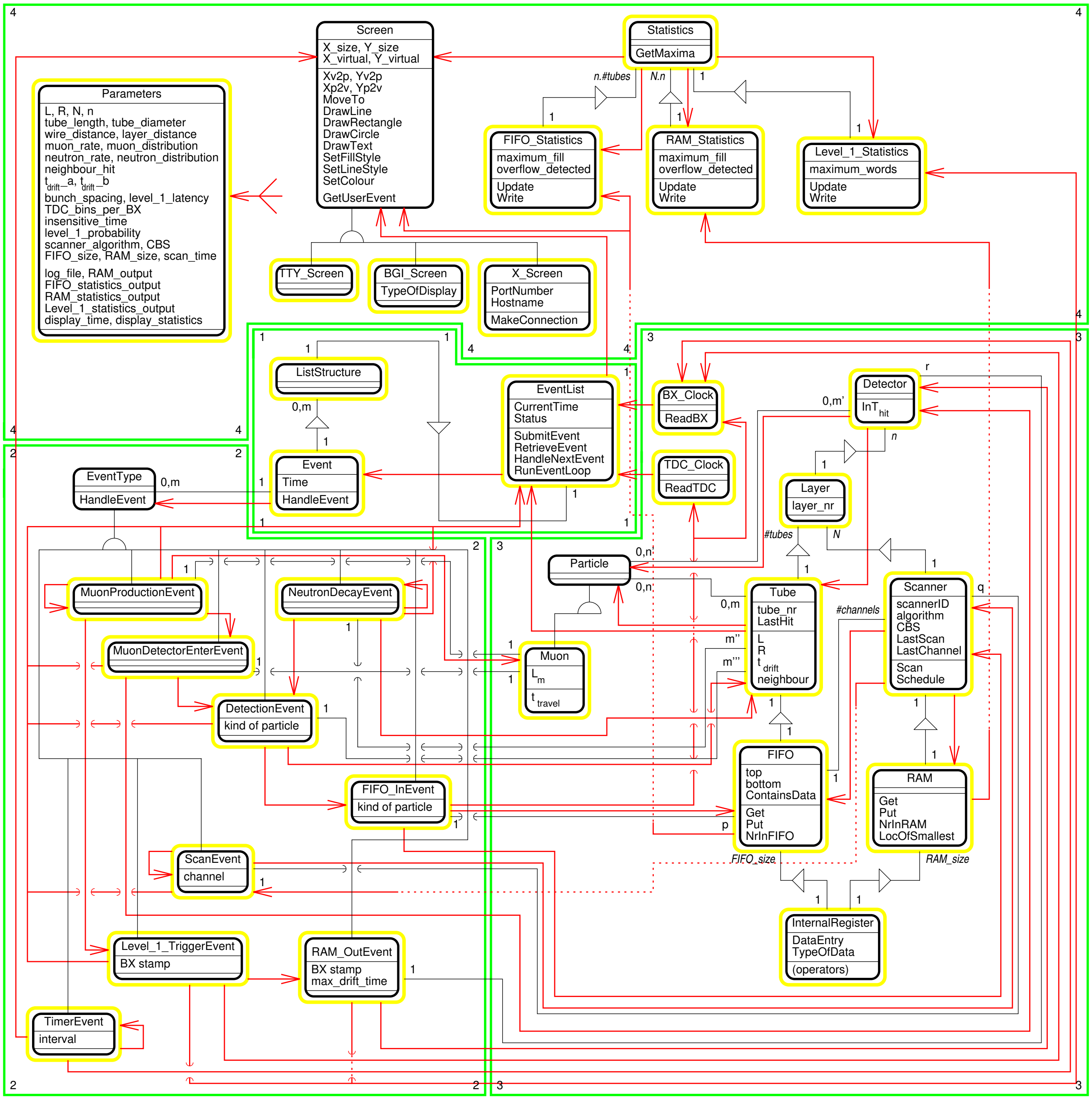


Figure 5.18: *Diagram of the Object Oriented Design.*

Chapter 6

Implementation

This chapter covers some of the 'highlights' of the implementation. A more detailed coverage of the implementation is given in part III, the programmer's guide.

The topics discussed in this chapter are the eventlist and the event types in section 6.2 and the concept of operator overloading in C++ is handled in sections 6.3 and 6.4. The details about communications through a socket (which are needed for the graphical display under the X Windows system) are discussed in section 6.5.

6.1 The function 'main'

Like in the language C, in C++ the function 'main' is the first function called when a program is started[1]. The function 'main' of the simulation program handles the initialization of the objects which are part of the simulation and starts the simulation itself. It is shown in figure 6.1.

The first object that is created, according to the order listed in section 5.3.1, is the eventlist, which is declared as a global variable and hence its instantiation is not shown in figure 6.1. The second object that will be created is one object of the class 'Parameters'. The constructor of this class reads the configuration file and the command-line options (in that order). The call to the function 'InitScreen' results in the creation of an object of one of the classes that is derived from the class 'Screen'. By default, an object of the class 'TTY_Screen' will be created, but the user can specify through the command-line options if an object of the class 'BGI_Screen' or 'X_Screen' needs to be created. Finally, the call to the function 'StartSimulation' results in the creation of an object of the class 'Detector' (and in the creation of objects of the classes 'Layer', 'Tube', etc., see section 5.3.1), the creation of an object 'MuonProductionEvent', the creation of a number of objects of the class 'NeutronDecayEvent' and the creation of an object of the class 'TimerEvent' if the user wants the time displayed on the screen.

[1]For C++ this function 'main' will *not* necessarily be the first function that is called. If objects of a class are declared globally, their constructors will be called first. A constructor is a special type of function which is called when an object of a class is instantiated.

```
int main(int argc, char *argv[])
{
        int return_value;
        cout << '\n' << argv[0] << ", version:  " << prog_version
             << " of " << prog_date << '\n' << '\n';

        Parameters *parameters = new Parameters(argc, argv);
        // Sets up all program parameters.

        InitScreen(terminal, display_options,
                   screen_width, screen_height, &Redraw);
        // Initialises the screen display.

        StartSimulation();
        eventlist→AddFunctions(screen_callbacks);

        eventlist→EventLoop();

        // delete the following objects in the right order:
        delete screen;
        delete parameters;
        delete statistics;

        return return_value;
}
```

Figure 6.1: *The function 'main'.*

After the objects have been created, the call-back functions for the display driver (which handle the user events) are added and the event loop is started.

6.2 The eventlist and the event types

In the definition of the class 'EventList', which is shown in figure 6.2, some differences from the design can be observed. Besides a few changes in the names, the attribute 'current time' has been renamed to 'last_event' and the service 'retrieve event' is represented by the method 'Retrieve', some 'attributes' (in C++ called 'data members') and 'services' (in C++ called 'methods') have been added.

The service 'submit event' has been split in three seperate methods. The first one is 'SubmitAbs' which submits an event to the eventlist with an absolute time stamp. The second version submits an event with a relative time stamp and the third submits an event at the current time. The last two versions make use of the method 'SubmitAbs' and the data member 'last_event', the way it is shown in figure 6.2.

```
class EventList
{
        private:
                loop_status status;
                long events_handled;
                        // counts the number of events handled
                time_type last_event;
                Binary_Tree *list;
                FUNCTION *functions;
                        // list of functions to be called in the loop
        public:
                EventList();
                ~EventList();

                inline time_type Time();
                inline loop_status Status();
                inline void AddFunctions(FUNCTION *functions_to_call);
                inline FUNCTION *GetFunctions() const;

                inline void TerminateLoop()
                {
                        status = TERMINATE;
                }

                void SubmitAbs(const time_type, EventType*);
                inline void Submit(const time_type relative_stamp,
                                        EventType *event)
                // submits an event with a time stamp relative to Time()
                {
                        SubmitAbs(last_event + relative_stamp, event);
                }

                inline void Submit(EventType *event)
                // submits an event with time stap Time()
                {
                        SubmitAbs(last_event, event);
                }

                Event *Retrieve();
                BOOLEAN HandleNextEvent();

                loop_status EventLoop();
};
```

Figure 6.2: *Class definition for the class 'EventList'.*

```
class Event
{
        private:
                time_type timestamp;
                EventType *kind_of_event;
        public:
                Event(const time_type stamp, EventType *event)
                {
                        timestamp = stamp;
                        kind_of_event = event;
                }

                virtual ~Event()
                {
                        delete kind_of_event;
                }

                inline time_type Stamp() const
                {
                        return timestamp;
                }

                inline EventType *Type() const
                {
                        return kind_of_event;
                }

                inline void HandleEvent()
                {
                        kind_of_event→HandleEvent();
                }

                void* operator new(size_t);
                        // overloaded new-operator; uses the free_list of unused
                        // nodes
                void operator delete(void*);
                        // overloaded delete-operator; place the deleted node in
                        // the unused nodes cache
};
```

Figure 6.3: *Class definition for the class 'Event'.*

```
loop_status EventList::EventLoop()
// retrieves and handles events in an infinite loop.
// returns status when loop ends (for some reason).
{
        if(status == NO_STATUS) status = RUNNING;
        while(status == RUNNING)
        {
                if(!HandleNextEvent())
                {
                        cerr << "\nOops...  no more events?\n";
                        status = EXCEPTION;
                }
                else if(functions) for(int f = 0; functions[f]; f++)
                        (*functions[f])();
        }
        return status;
}
```

Figure 6.4: *The method 'EventLoop' of the class 'EventList'.*

```
BOOLEAN EventList::HandleNextEvent()
// retrieves the first event in the queue and handles it.
{
        Event *this_event = Retrieve();

        if(this_event == NULL) return FALSE;
        else
        {
                this_event->HandleEvent();
                delete this_event;
        }
        events_handled++;
        return TRUE;
}
```

Figure 6.5: *The method 'HandleNextEvent' of the class 'EventList'.*

6.2.1 The methods 'EventLoop' and 'HandleNextEvent'

The method 'EventLoop', which is the equivalent for the service 'run event loop' from the design, is shown in figure 6.4 and it has virtually the same structure as the service chart shown in figure 5.5. The events are retrieved from the list and handled by the methode 'HandleNextEvent', which is shown in figure 6.5. The event loop does not end unless it is terminated by a call to the method 'TerminateLoop' or when an error occurs (usually because one or more time stamps in the list have been corrupted, for whichever which reason). The basic ideas behind the event loop are from [Vermeulen 92].

The method 'HandleNextEvent' calls the method 'HandleEvent' of the class 'Event'. This method is highlighted in figure 6.3. After the event has been handled, the event is deleted. To speed up the creation and destruction of objects of the class 'Event', the operators 'new' and 'delete' have been overloaded, i.e. these operators are reimplemented. The motivation and implementation of this fact are discussed in section 6.3.

6.2.2 The event types

The class definition for the class 'EventType' is shown in figure 6.6. The method 'HandleEvent' is a pure virtual method (indicated by the '= 0' behind the function declaration) as the class 'EventType' is an abstract class. The class serves as a base for the derived classes. An example of such a derived class is the event type 'MuonProductionEvent' which is shown in figure 6.7.

Calls to the (pure virtual) method 'HandleEvent' of the class 'EventType' are translated run-time to a call the the method 'HandleEvent' of the appropriate derived class. The C++ compiler inserts code to handle this translation into the program executable.

6.3 Overloaded operators 'new' and 'delete'

To save time and space, the allocation and de-allocation of objects of a known size can be handled more efficiently when the programmer implements new versions of the operators 'new' and 'delete' (see [Stroustrup 91], page 177). The principle is simple. When an array of objects is allocated, these objects will be placed in a contiguous memory block unlike allocating seperate objects, in which case the objects are aligned to a word boundary (thus leaving unused 'gaps' between the objects). This is visualized in figure 6.8. The allocation of new objects, using the reimplemented 'new'-operator, is only a matter of moving pointers, thus leaving out the presumably time consuming actions the system defined 'new'-operator takes.

The 'new'-operator, shown in figure 6.10, checks if there are any objects available in the list of free objects (called 'free list' for short). If this is the case, a pointer to this object will be returned and the object is removed from the free list. If the free list is empty, a new block of 'nr_new_objects' objects will be allocated and each object within this block is linked to the next one through a pointer which should preferably be part of the structure to be allocated. In the case of the example this field is simply called 'next'. This operation in visualized in figure 6.9.

```
class EventType
{
        public:
                virtual void HandleEvent() = 0;
};
```

Figure 6.6: *Class definition for the class 'EventType'.*

```
class Muon_Production_Event: public EventType
{
        private: Muon *muon;
        public: virtual void HandleEvent();
};

void Muon_Production_Event::HandleEvent()
{
        if(!(muon = new Muon)) OutOfMemory("Muon");
        time_type next_production = ceil(Exponential(muon_distribution)
                                                        / BX_clock_cycle)
                                                    * BX_clock_cycle;
        // round time to the next BX clock tick.
        time_type t_travel = s2ns(sqrt(muon→L_m() * muon→L_m()
                                                      + R * R) / c);

        eventlist→Submit(t_travel, new Muon_Detector_Enter_Event(muon));
        if(TryChance(level_1_chance / detector_area_ratio))
                eventlist→Submit(level_1_trigger, new Level_1_Trigger_Event);

        // generate next muon production:
        eventlist→Submit(next_production, new Muon_Production_Event);
}
```

Figure 6.7: *Class definition for the class 'MuonProductionEvent' as an example of a class derived from 'EventType'.*

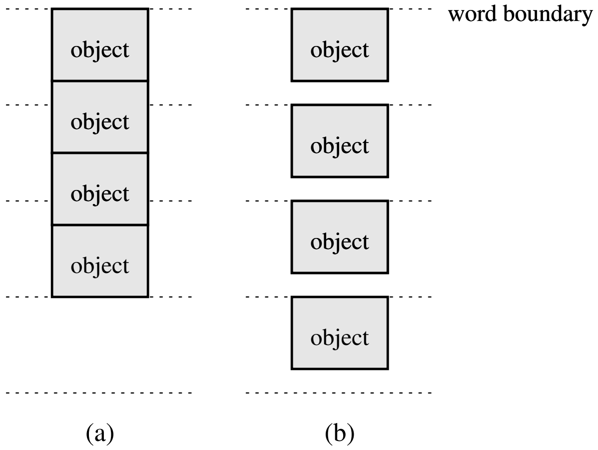


Figure 6.8: *The allocation in the memory of four objects as a contiguous array* (a) *or as separate objects* (b).

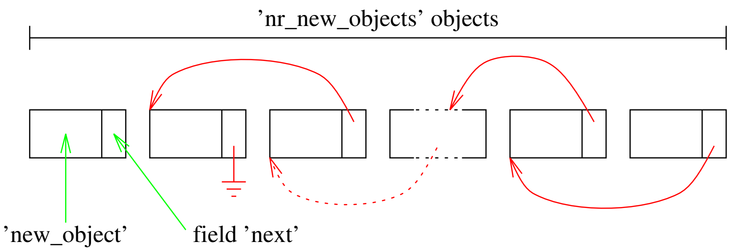


Figure 6.9: *Visualization of the operation of the overloaded 'new'-operator.*

The 'delete'-operator is trivial. It simply links the de-allocated object to the free list. The 'delete'-operator is shown in figure 6.11.

The typecasts in the implementation of the 'new'-operator are needed since straight forward allocation of an array of objects would result in an infinite recursion. The typecast in the 'delete'-operator is needed since the actual object no longer exists. Note that the operators can only be used for single objects and not for arrays of objects.

6.4 The structure 'internal_registers'

The structure 'internal_registers' implements the memory locations in the FIFO queues and the RAM memories. It demonstrates the concept of operator overloading in C++. Its listing is displayed in figure 6.12.

Objects of the structure can be added, subtracted and compared with other objects of the structure. It offers three constructors and one assignment operator, which are

```
void* SomeClass::operator new(size_t)
{
        register SomeClass *new_object = free_list;

        if(new_object) free_list = (SomeClass*)new_object→next;
        else
        {
                SomeClass *new_objects
                = (SomeClass*)new char[sizeof(SomeClass) * nr_new_objects];

                if(new_objects == NULL)
                {
                        cerr ≪ "Fatal error:  out of memory.\n"
                        exit(1);
                }
                else
                {
                        for(new_object = free_list =
                                &new_objects[nr_new_objects - 1];
                        new_objects < new_object;
                        new_object--)
                                new_object→next
                                = (SomeObject*)(new_object-1);
                        (new_object + 1)→next = NULL;
                }

        }
        return new_object;
}
```

Figure 6.10: *Overloaded 'new'-operator.*

```
void SomeClass::operator delete(void *object, size_t)
{
        ((SomeClass*)object)→next = free_list;
        free_list = (SomeObject*)object;
}
```

Figure 6.11: *Overloaded 'delete'-operator.*

discussed below. The numbers refer to the numbers in figure 6.12.

1. The first constructor is used when a new object of the structure 'internal_registers' is declared without assigning a value to it, as in:

```
internal_registers p;
```

2. The second constructor is used when a new object is declared and the value of an existing object is assigned to it. It is called the 'copy constructor'.

```
internal_registers q = p;
internal_registers r(p);
```

3. The third type of constructor is used when a new object is declared and a value is assigned to it through integer parameters:

```
internal_registers r1(3);
internal_registers r2(BX_stamp, TDC_stamp, scanner, channel);
```

   The first value, the BX stamp, is mandatory; for the other parameters default values are specified. The declaration of the object 'r1' is the same as the following declaration:

```
internal_registers r1(3, 0, 0, 0, 0);
```

4. The assignment operator is similar to the copy constructor. The difference is that the assignment operator is used when left-hand object or the assignment already exist:

```
r1 = p;
```

It is also possible to assign an integer value to an object of the structure 'internal_registers', like in the following example:

```
p = BX_clock→Read();
```

This assignment however is not as straight forward as it may look. First the constructor number 3 is called because of an implicit type cast of the integer to an 'internal_registers'. The second step is to call the assignment operator (number 4) to assign the temporary object to the object 'p'.

The same mechanism is used when an 'internal_registers' is *compared* with an integer. In the example below, the integer 0 is converted first to an temporary object of the structure 'internal_registers' before the comparison operator is called:

```
if( p == 0 ) cout << "p is empty!\n";
```

6.5 Communication through sockets

A Unix socket is an endpoint for communication through an interprocess communication channel (IPC). Sockets allow processes to 'speak' to each other. A socket is described

```
struct internal_registers
{
        int tag_ID;
        int scanner_channel;
        long BX_stamp;
        int TDC_stamp;

        inline int tag() const;
        inline void SetTag(const int &tag);
        inline int scanner() const;
        inline void SetScanner(const int &scanner);
        inline int channel() const;
        inline void SetChannel(const int &channel);
        inline long BX() const;
        inline void SetBX(const long &BX);
        inline int TDC() const;
        inline void SetTDC(const int &TDC);

        inline internal_registers(const internal_registers &element)        // (2)
        // copy constructor
        {
                SetBX(element.BX());
                SetTDC(element.TDC());
                SetScanner(element.scanner());
                SetChannel(element.channel());
                SetTag(element.tag());
        }

        inline internal_registers()                                         // (1)
        {
                scanner_channel = BX_stamp = TDC_stamp = tag_ID = 0;
        }

        inline internal_registers(const long &BX,
                        const int &TDC = 0,
                        const int &sc = 0, const int &ch = 0,
                        const int &tag = 0)                                 // (3)
        {
                SetBX(BX);
                SetTDC(TDC);
                SetScanner(sc);
                SetChannel(ch);
                SetTag(tag);
        }
```

Figure 6.12: *Structure definition for the struct 'internal_registers'.*

```
        inline internal_registers& operator=(const internal_registers &element)  // (4)
        // assignment
        {
                SetBX(element.BX());
                SetTDC(element.TDC());
                SetScanner(element.scanner());
                SetChannel(element.channel());
                SetTag(element.tag());
                return *this;
        }

        friend internal_registers operator+(const internal_registers&,
                                            const internal_registers&);
        friend internal_registers operator-(const internal_registers&,
                                            const internal_registers&);
        friend BOOLEAN operator==(const internal_registers&,
                                  const internal_registers&);
        friend BOOLEAN operator<(const internal_registers&,
                                 const internal_registers&);
        friend BOOLEAN operator≤(const internal_registers&,
                                 const internal_registers&);
        friend BOOLEAN operator>(const internal_registers&,
                                 const internal_registers&);
        friend BOOLEAN operator≥(const internal_registers&,
                                 const internal_registers&);
};


inline internal_registers operator+(const internal_registers &element1,
                                    const internal_registers &element2)
{
        long BX_stamp = element1.BX() + element2.BX();
        int TDC_stamp = element1.TDC() + element2.TDC();
        if(TDC_stamp > TDC_ticks_per_BX)
        {
                TDC_stamp -= TDC_ticks_per_BX;
                BX_stamp++;
        }
        return internal_registers(BX_stamp, TDC_stamp,
                                  element1.scanner(), element1.channel(),
                                  element1.tag());
}
```

Figure 6.12: continued. *Structure definition for the struct 'internal_registers'.*

```
inline internal_registers operator-(const internal_registers &element1,
                                    const internal_registers &element2)
{
        long BX_stamp = element1.BX() - element2.BX();
        int TDC_stamp = element1.TDC() - element2.TDC();
        if(TDC_stamp < 0)
        {
                TDC_stamp += TDC_ticks_per_BX;
                BX_stamp--;
        }
        return internal_registers(BX_stamp, TDC_stamp,
                                  element1.scanner(), element1.channel(),
                                  element1.tag());
}

inline BOOLEAN operator==(const internal_registers& element1,
                          const internal_registers& element2)
{
        return (element1.BX() == element2.BX()) &&
               (element1.TDC() == element2.TDC()) &&
               (element1.scanner_channel == element2.scanner_channel) &&
               (element1.tag() == element2.tag());
}

inline BOOLEAN operator≠(const internal_registers& element1,
                         const internal_registers& element2)
{
        return !(element1 == element2);
}

inline BOOLEAN operator<(const internal_registers& element1,
                         const internal_registers& element2)
{
        return element1.BX() == element2.BX()
               ? element1.TDC() == element2.TDC()
                 ? element1.scanner_channel < element2.scanner_channel
                 : element1.TDC() < element2.TDC()
               : element1.BX() < element2.BX();
}
```

Figure 6.12: continued. *Structure definition for the struct 'internal_registers'.*

```
inline BOOLEAN operator≤(const internal_registers& element1,
                         const internal_registers& element2)
{
        return element1 < element2 || element1 == element2;
}

inline BOOLEAN operator>(const internal_registers& element1,
                        const internal_registers& element2)
{
        return !(element1 ≤ element2);
}

inline BOOLEAN operator≥(const internal_registers& element1,
                         const internal_registers& element2)
{
        return !(element1 < element2);
}
```

Figure 6.12: continued. *Structure definition for the struct 'internal_registers'.*

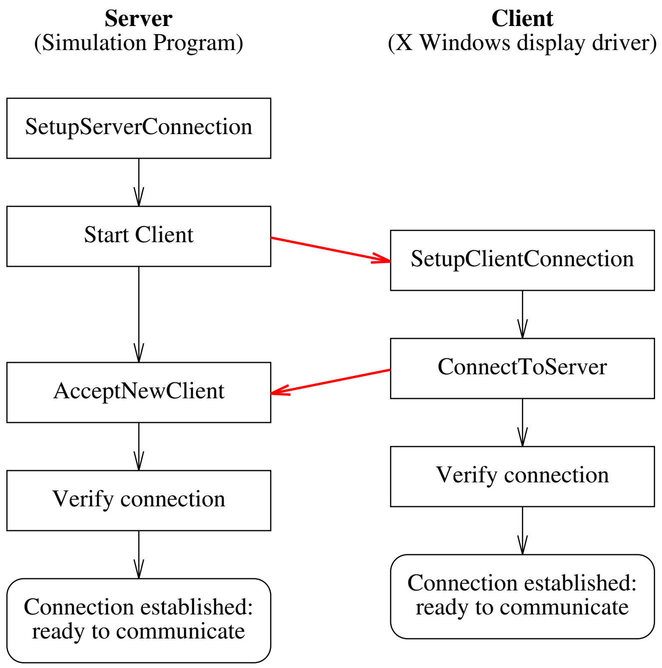


Figure 6.13: *The steps in setting up a client – server communication channel.*

by a host name and a port number. In the simulation program this concept is used to let the simulation communicate with the X Windows display driver. This is called 'client – server communication' since one of the programs (in this case the X Windows driver) serves as a client, which passively listens to what the server, the simulation program, tells it to do.

Before the communication can take place, the socket has to be 'installed'. An oftenly used analogy is that of the telephone, like in [Frost 90]. The process of setting up the channel uses a slightly modified version of the communication procedures by Louis Vuurpijl, see [Vuurpijl 92]. The following steps are involved (see also figure 6.13):

1. The first step is to install the socket. The server calls the function 'SetupServer-Connection' to do this task. This function basically does two actions:

   (a) It creates a socket.

   (b) It binds the socket to an address (i.e. the host name and a port number).

   To be able to run multiple incarnations of the simulation program, the function 'SetupServerConnection' looks for a free port number. The first free port will be returned. To avoid intervening with other programs that use sockets as well, a lower and an upper bound for the port number have to be specified. The function will only look for free ports within this range.

2. The second step is to start the X Windows display driver program, which runs in parallel with the simulation. Since the port number is obtained dynamically, it has to be provided to the display driver through a command-line option. Evidently the display driver has to understand and interpret this command-line option.

3. The third step of setting up the channel consists of two processes run in parallel:

   (a) The simulation program waits until the display driver reports its presence and ability to communicate. This task is performed by the function 'AcceptNewClient'.

   (b) The display driver sets up its own connection using the function 'SetupClientConnection' and 'ConnectToServer'.

   When the connection between the client and server is established, some dummy data is sent over the channel to verify that there is in fact a connection.

The connection is now established and communications over the channel can take place.

Chapter 7

Conclusions, results and hints for further research

7.1 Test runs with different types of list structures

Test runs have been made with the simulation program to acquire an insight in how the different list structures (namely a linked list, a generic binary tree and an AVL binary tree) behave. The results of the test runs are shown in figure 7.1.

The most obvious conclusion that can be drawn from the results is that the benefits of the binary trees only come apparent when the number of entries in the list exceeds 2 000. This means that for the simulation running under the default settings, in which case the average number of entries is about 1 050, the linked list is the most efficient implementation for the list.

7.2 A result from the simulation

The simulation program showed its use already. Since the number of entries in the FIFO queues and the RAM memories appeared to be higher than expected, a different kind of 'scanner' needed to be developed. This type is documented in [Marchioro 94] and its diagram is shown in figure 7.2. In this type of scanner, each channel has two registers. The second register will only be used if the first one is filled. The channels are not scanned in order, but as soon as one of the channels has data available, the data will be put in the circular buffer. When a level 1 trigger is issued, the measurements stored in the buffer are passed to the read-out FIFO and sent to the data processing logic. This device typically has 32 channels.

In the simulation, this scanner can be abstracted to a 'normal' scanner (the one mentioned under item 1 in section 2.3.2) with one channel having a two entries deep FIFO queue. The maximum fill of the circular buffer then can be derived from the maximum fill of the RAM memories (which in this case are connected to one channel) times 32.

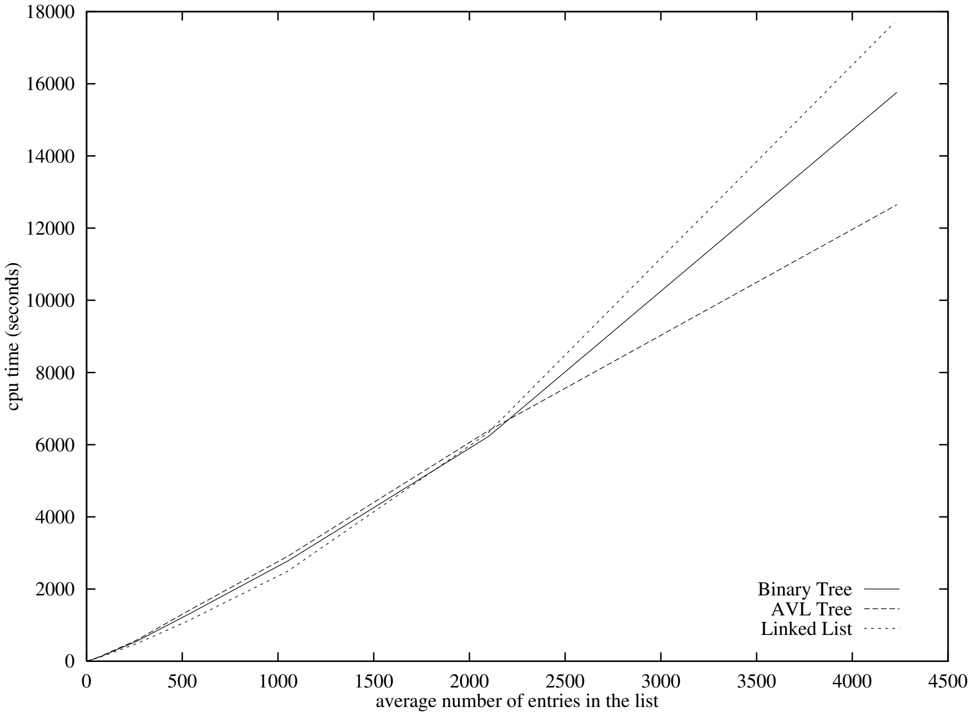


Figure 7.1: *Results of test runs with different types of list structures at a muon rate of* $1.0\frac{\mathrm{Hz}}{\mathrm{cm}^2}$ *and a neutron rate of* $1\,000.0\frac{\mathrm{Hz}}{\mathrm{cm}^2}$. *The x-axis shows the average number of events in the list; the y-axis shows the CPU time in seconds needed on a Hewlett Packard 9000 model 715/75 workstation.*

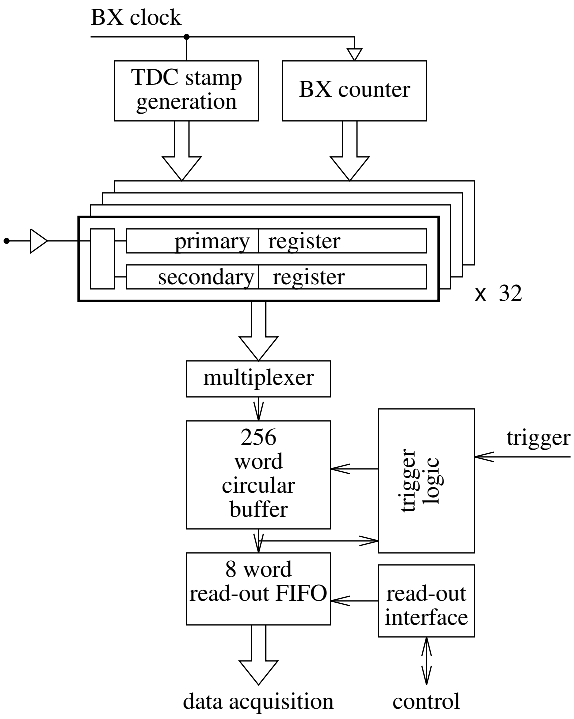


Figure 7.2: *Simplified schematic circuit diagram of the 32 channel, 19 bit, 0.5 ns resolution TDC.*

7.3 Conclusions

- Logic, and especially metric temporal logic, provides a rather fair mathematical basis for the specification of the events within an event based simulation. Its only, but most apparent, lack is that logic has no possibilities to express probabilities. In chapter 4 a first try has been made to add this feature, but it would need further research to make it a good, formal definition.
- The events specified using metric temporal logic can be transformed into C++ classes nearly without any changes, hence this transformation lends itself to be done by automation. Using mathematical tools, it would be possible to prove the specification and to transform it into a more efficient form using logical deduction and inferences. For other mathematical specification methods such proving tools already exist.
- The concept of inheritance in object oriented design and programming is very well suited for event based simulations. In procedural programming languages the programmer would have had to made the distinctions between the different types of events, in object oriented languages like C++ the *compiler* accounts for this task.

7.4 Further research

Although this thesis handles a stand-alone, confined project which ended with the publication of this thesis, some points can be distinguished that can be examined at a closer look.

- An study can be made on how to reconstruct a muon track from the data that is obtained from the RAM memories. Since the simulation program has the ability to produce this output as well as to log the actual muon tracks, reconstruction algorithms can be put to the test if they perform their tasks properly. Since the data from the simulation program approaches the real data that will be delivered by the ATLAS detector, it would probably provide a better test set for those algorithms than data generated by (other) Monte Carlo simulations.
- An extension to metric temporal logic so it will be able to express probabilities. If this extension can be formally defined, logic will be a very good formal method to specify events in an event based simulation.
- Tools – which to my knowledge do not exist at this moment – could be created to prove a specification made in metric temporal logic and to transform it into an executable program.

Part II

Appendices, Bibliography and Index

Appendix A

The AVL binary tree

To keep the access times for operations on a binary tree within limits, even in the worst case, it will be needed to maintain the tree in balance. A method to keep a tree in balance is to reduce the difference in depth between the left and right subtrees.

A tree T is called k-*balanced* if for each node n the following is valid:

$$|\text{depth}\,(\text{right}\,(n)) - \text{depth}\,(\text{left}\,(n))| \leq k$$

The term depth (right (n)) $-$ depth (left (n)) is called the balance factor of the node n and is printed as balfac(n).

In the case that $k = 1$, a tree is balanced if it is 1-balanced. This type of binary tree is called the AVL tree after its two 'inventors' Adel'son-Vel'skii and Landis, [van der Weide 89]. In section A.1 the insert operation on an AVL tree shall be discussed. The remove operation is described in section A.2 and the operation of rebalancing the tree – the perhaps most important operation in the AVL tree since is distinguishes the AVL tree from an ordinary binary tree – will be shown in section A.3.

A.1 The insert operation

The insert operation for AVL trees works similar to that of generic binary trees. If an element is inserted into the tree, its key is compared with the key of the node (starting with the root of the tree). If the key of the new element is smaller[1] than the key of the node, the insert operation will be continued in the left son; if it is larger the insertion will be continued in the right son. When the insert algorithm reaches the bottom of the tree – i.e. the son in which direction the insertion would have been continued does not exist – a new node is created and is made the new son of the last visited node in the chosen direction.

[1]In the AVL tree used for the simulation program, a particular key must be allowed to be present in the tree more than once (since more than one event may occur at the same point in time). In a common AVL tree, the new element is rejected since it is already present in the tree, but in the simulation program the new element will only be rejected if its 'user data' is equal to that of the node in the tree. If these are inequal, an arbitrary son will be chosen – in the case of the simulation program the right son.

However, the insert operation may cause the tree to become out of balance. To be able to detect this, the insert operation will be given an extra boolean parameter 'deeper' which will be assigned the value `TRUE` if a new node has been added to the tree. If this is indeed the case, a check has to be made if the (sub)tree has become unbalanced and depending on that special action may need to be taken. The next cases can be distinguished [van der Weide 89]:

- Both subtrees were equally deep before the addition of the new node.

  The tree has not become unbalanced because of the addition. The balance factor was 0 and will become – depending on the direction in which the new node was added – 1 or -1. The tree however has become deeper which will be reported to the parent node through the parameter 'deeper'.

- The subtree in which the addition took place, was undeeper.

  Again the tree has not become unbalanced. The balance factor of the root of the subtree will be 0.

- The subtree in which the addition took place, was already deeper.

  The (sub)tree is no longer in balance. Its balance factor has become 2 or -2. So the tree needs to be reorganized. This however will be prosponed till section A.3.

To be able to administrate the balance factor of a particular node quickly, a field will be added to each node which holds its balance factor (which is either *balanced*, *left* or *right*).

A.2 The remove operation

In contrast to the insert operation – in which all modifications to the tree took place at the bottom of it – the remove operator may modify the tree at each arbitrary level. It is however not allowed to 'just cut away' a node in tree – removing a node would leave its two subtrees dangling 'somewhere'. Therefore some special actions need to be taken to fill the gap that would be created. The easiest solution is to replace the removed node by either the largest element in the left son or the smallest element in the right son, but of course one has to take the balance factor of the node in mind, since removing an element in the undeepest subtree would bring the tree unwantedly out of balance. Therefore the following criteria have to be checked:

- The left son is empty.

  The node that will be remove can now easily be replace by (the root of) its right subtree. See figure A.1. The subtree however has be come undeeper, which will be reported to the parent node through a parameter 'undeeper' (similar to the parameter 'deeper' in the insert operation).

- The right son is empty.

  The left son can now replace the removed node, see the previous criterium.

- One of the sons in deeper than the other son.

  If the right son is deeper than the left son, the node with the smallest key in the right son replaces the node that is removed. Than that node will be removed from the right son. If the right son has become undeeper, than the subtree will be in balance, otherwise the balance factor will remain unchanged. This is shown in figure A.2. The parent node will be notified that this subtree has become undeeper.

  If the left son is deeper than the right son, the node with the largest key in the left son will be 'moved up'. The principle is the same as above.

- The node is in balance.

  An arbitrary directory can be chosen. If that direction is left, the node with the largest key in the left son will replace the node that is removed. If the left son has become undeeper, the balance factor will be adjusted to indicate that the balance is now to the right. The subtree as a whole has *not* become undeeper, which will be reported to the parent node, so the parameter 'undeeper' will be assigned the value `FALSE`. See figure A.3.

The remove operation as such searches the node that needs the be removed using the usual method to search a binary tree. If it has found the node, it will be removed using the algorithm mentioned above. The parameter 'undeeper' will be passed to the parent node. The parent node checkes the parameter 'undeeper' and takes the proper actions. The following cases can be distinguished:

- The subtree was in balance.

  The balance factor will be adjusted (if the element was removed in the left son, the balance will now be to the right). The subtree as a whole however has not become undeeper, which will be reported to its own parent node.

- The subtree in which the element was removed, was deeper.

  This node will now be balanced, but the subtree as a whole has become undeeper.

- The subtree in which the element was removed, was already undeeper.

  The subtree now has a balance factor of -2 or 2, meaning that this subtree has to be reorganized. This will be discussed in section A.3.

A.3 Rebalancing the tree

If a subtree has become out of balance, it will need to be reorganized. This process of reorganizing is called 'rebalancing'. Assume that the node that needs to be rebalanced is the node p and that its right son, the tree T_2 is deeper than the left son T_1. The subtree T_2 has a root q. Now three cases can be distinguised, depending on the balance factor of the node q:

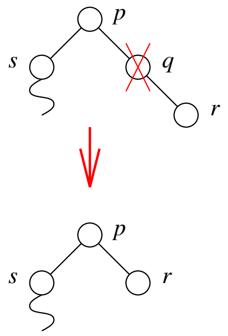


Figure A.1: *Remove operation on an AVL tree. The node q, that will be removed, has an empty left son.*

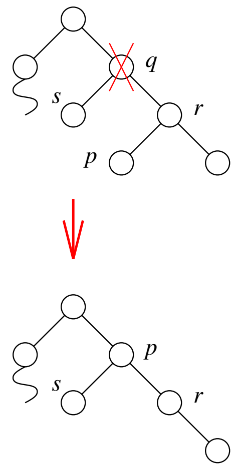


Figure A.2: *Removal of a node* q *which has one son that is undeeper than the other.*

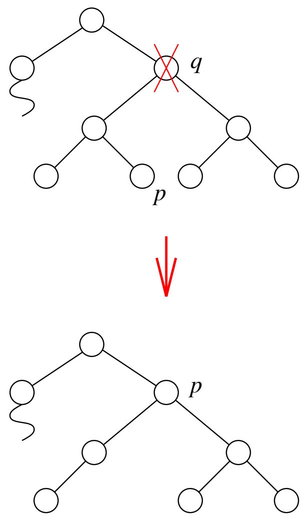


Figure A.3: *Removal of a node* q *which is in balance*

balfac(r) before	balfac(p) after	balfac(q) after
-1	0	$+1$
0	0	0
$+1$	-1	0

Table A.1: *Correlation between the balance factors before and after a double rotation.*

- balfac(q) $= 0$ or balfac(q) $= +1$

  This situation is shown in the top half of figure A.4. If the tree needs to be rebalance, the subtree T_{22} must be hung higher in the tree. The result of the operation is shown in the bottom half of the same figure. This operation is called a 'single rotation'. The depth of the subtree might have changed (in case balfac(q) $= +1$) which needs to be reported to the parent of the (rebalanced) subtree. In the case balfac(q) $= 0$ the balance factor of the (new) right son has changed and will be -1 (the opposite of the balance factor of the node p). In the other case the right son will be balanced.

- balfac(q) $= -1$

  If the balance factor of the node q is the opposite of that of node p, a single rotation would not work since the tree would still be out of balance. Since the left subtree T_{21} of q has a depth greater than 1, the situation can be observed at greater detail. The root of the subtree T_{21} is called r and it has the sons T_{211} and T_{212}. This is shown in figure A.5. The operation is called a 'double rotation'. The new balance factors of the nodes p and q after the rotation depend on the balance factor of r before. The new values are shown in table A.1. The depth of the tree however has not changed, so none of the nodes in the higher levels in the tree has to taken special actions.

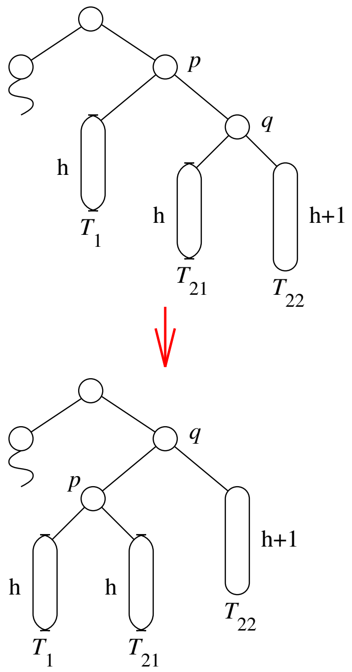


Figure A.4: *Single rotation on an AVL tree.*

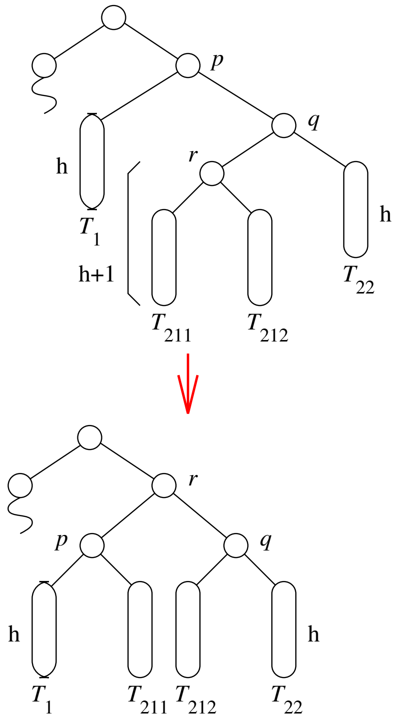


Figure A.5: *Double rotation on an AVL tree.*

Appendix B

Logical inferences

In this appendix, some axioms are derived that are used in chapter 4. The rules in these inferences are described in [Veldman 84].

Axiom B.1 *To infer* $(a \wedge b) \to c$ *from* $a \to (b \to c)$*:*

$$\dfrac{\dfrac{\dfrac{a \wedge b^{(1)}}{b} \qquad \dfrac{\dfrac{a \wedge b^{(1)}}{a} \qquad a \to (b \to c)}{b \to c}}{c}}{(a \wedge b) \to c}\ (1)\textit{: give up } a \wedge b$$

Axiom B.2 *To infer* $a \to (b \to c)$ *from* $(a \wedge b) \to c$*:*

$$\dfrac{\dfrac{\dfrac{\dfrac{a^{(2)} \quad b^{(1)}}{a \wedge b} \qquad (a \wedge b) \to c}{c}}{b \to c}\ (1)\textit{: give up } b}{a \to (b \to c)}\ (2)\textit{: give up } a$$

Bibliography

[ATLAS92] ATLAS. *ATLAS: Letter of Intent for a General-Purpose pp Experiment at the Large Hadron Collider at CERN*. CERN, October 1, 1992. CERN/LHCC/92-4, LHCC/I 2. Available on the CERN WWW server.

[ATLAS93a] ATLAS. *B Physics with the ATLAS Experiment at LHC*. CERN/LHCC/93-53, October 15, 1993. This document is available from the CERN WWW server.

[ATLAS93b] ATLAS. *Impact of LHC bunch spacing scheme on the ATLAS detector and its performance*, January 25, 1993. Available on the CERN WWW server.

[ATLAS94] ATLAS. *Progress Report on ATLAS Milestones*, chapter 3.2. CERN, May 23 1994. Available on the CERN WWW server.

[Bakker, e.a. 94] F. Bakker et al. *Honeycomb Strip Chambers for the ATLAS Muon Spectrometer*. NIKHEF-H, Amsterdam; University of Amsterdam; University of Nijmegen; MSU, Moscow, Russia and IHEP, Protvino, Russia, January 19, 1994. Available on the NIKHEF WWW server.

[Banks, Carson 84] Jerry Banks and John S. Carson. *Discrete-event System Simulation*. Prentice Hall, Englewood Cliffs, New Jersey, 1984. Covers the same subjects as [Bratley, e.a. 87].

[Bratley, e.a. 87] Paul Bratley, Bennett L. Fox, and Linus E. Schrage. *A Guide to Simulation*. Springer-Verlag, New York, Berlin, Heidelberg, second edition, 1987.

[Brockett, Levine 84] Patrick Brockett and Arnold Levine. *Statistics and Probability and their applications*. Saunders College Publishing, 1984.

[CDF 94] CDF. *Evidence for Top Quark Production in $\bar{p}p$ Collisons at $\sqrt{s}$ = 1.8 TeV*, April 22 1994. FERMILAB-PUB-94/097-E, CDF/PUB/TOP/PUBLIC/2561. Available on the Fermilab WWW server.

[Coad, Yourdon 91a] Peter Coad and Edward Yourdon. *Object-oriented Analysis*. Prentice Hall, Englewood, New Jersey, 2nd edition, 1991.

[Coad, Yourdon 91b] Peter Coad and Edward Yourdon. *Object-oriented Design*. Yourdon Press, Englewood Cliffs, New Jersey, 1991.

[Frost 90] Jim Frost. *BSD Sockets: A Quick and Dirty Primer*, 1990.

[König 92] A. König. *Muon Chamber Read-Out for ATLAS*. Overhead presentation for meetings of the ATLAS collaboration, November 1992.

[Koymans 89] R.L.C. Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*. PhD thesis, Technische Universiteit Eindhoven, 1989.

[Koymans 92] Ron Koymans. *Specifying Message Passing and Time-Critical Systems with Temporal Logic*, volume 651 of *Lecture Notes in Computer Science*. Springer-Verlag, 1992. Updated and extended version of [Koymans 89].

[Marchioro 94] A. Marchioro. *32 Channel, 19 bit, 0.5 ns resolution TDC with On-Chip Buffering and Trigger Matching*. CERN ECP-MIC, 1994.

[Papoulis 91] Athanasios Papoulis. *Probability, Random Variables and Stochastic Processes*. Electrical & Electronic Engineering Series. McGraw-Hill International Editions, third edition, 1991.

[Stroustrup 91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesly Publishing Company, AT&T Bell Laboratories, Murray Hill, New Jersey, U.S.A., second edition, 1991.

[van den Goor 92] G.P.M. van den Goor. *Formalization and Comparison of Six Object Oriented Analysis and Design Methods*. Master's thesis, University of Nijmegen, Department of Information Systems, March 1992.

[van der Weide 89] Th.P. van der Weide. *Van Datastructuur tot Informatiesysteem*. Vakgroep Informatiesystemen, Faculteit der Wiskunde en Informatica, Katholieke Universiteit Nijmegen, Spring 1989. Lecture notes for the course "From Data Structure to Information System" at the University of Nijmegen. In Dutch.

[Veldman 84] Wim Veldman. *Introduktie tot de Logica*. Mathematisch Instituut, Fakulteit der Wiskunde en Natuurwetenschappen, Katholieke Universiteit Nijmegen, Toeinooiveld, 6525 ED Nijmegen, Autumn 1984. Lecture notes for the course "Introduction to Logic" at the University of Nijmegen. In Dutch.

[Vermeulen 92] Jos Vermeulen. *Discrete Event Simulation and Modelling in C++*. Unpublished paper, December 1992.

[Vuurpijl 92] L.G. Vuurpijl. *Convis, Action-Oriented Control and Visualization of Neural Networks, Version 1.0.* Technical report, University of Nijmegen, Faculty of Mathematics and Informatics, Toernooiveld 1, 6525 ED Nijmegen, The Netherlands, December 1992.

[Weeland 94] J.M. Weeland. *Discrete Simulation of "Muon Chamber Read-Out" electronics for LHC/ATLAS*. Master's thesis: full edition, University of Nijmegen, Department of Experimental Informatics for Technical Applications, August 1994.

Note: In those items where ATLAS is mentioned as the author, in fact the ATLAS Collaboration must be read and the Collider Detector at Fermilab Collaboration must be read instead of CDF.

Index

– U –

– V –

– W –

– X –