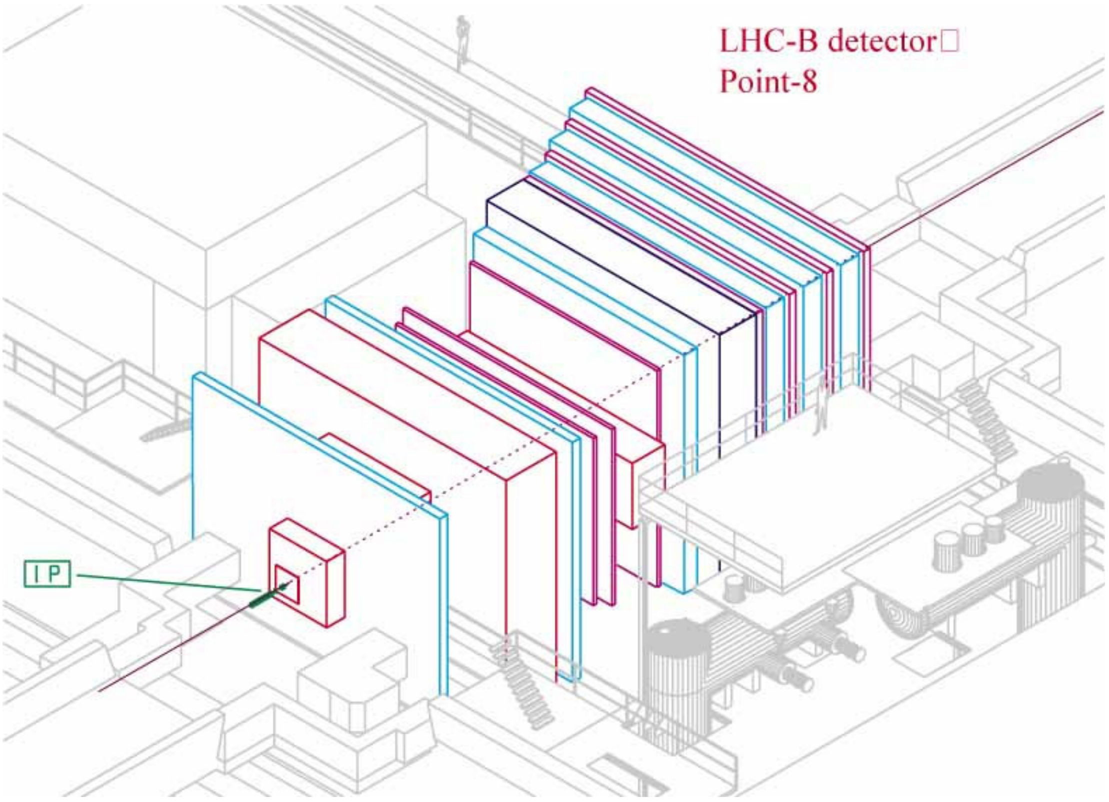
European Organization for Nuclear Research (CERN),
LHCb Collaboration, CERN IT/DB, GEANT4, LCG

Technical University Košice,
Faculty of Electrical Engineering and Informatics,
Department of Computers and Informatics

Object Databases for Detector Description

PhD Thesis

Radovan Chytráček



Abstract

Database systems play very important role in physics data processing for every High Energy Physics (HEP) experiment. HEP community in Large Hadron Collider (LHC) project[2] is in the transition from FORTRAN based software systems to object-oriented (OO) software implemented mostly in C++ and from data records to persistent objects. Along with that a new physics data management issues will arise in order to allow for the transition from sequential files approach towards the data access based on object-oriented selection mechanisms of the required physics contents. In order to conform to object-oriented paradigm, LHC experiments have to intensively invest into the design and development of object data storage systems for both the on-line (data acquisition and real-time processing) and off-line (simulation, reconstruction and analysis) computing environments, e.g. the event data store, detector description database, calibration and alignment database etc.

In the course of this development effort some of the data has begun to be stored in XML based data formats. These new formats bring with them new technological solutions as well as new challenges. One of the challenges the issue of proper serialization of the application data forth and back to XML form. This work presents some real-life use-cases and the corresponding solutions for these are described.

Foreword

This work has been done at the European Laboratory for Nuclear Research (CERN). It has started during my doctoral studentship contract with the CERN LHCb experiment collaboration. After that time, however the work has been continuously evolving during the next couple of years as my involvement in the CERN's LHC research programme was going on.

The work was further gradually extended as I had been assigned to work on more projects and faced various data processing problems and challenges. It has helped me in resolving many of them. It can not be concluded yet that the results, I have achieved, do provide the complete solutions to the problems I was facing but they should at least bring some new light to the research facing the same class of problems I did and provide guidance and inspiration to newcomers in this domain who might meet such challenges along their research path.

I want to remind the reader that the solutions presented in this document might not be the definitive ones nor resolving the problems in place to a full extent. The opposite is true. They often provoke many open questions for which I have not found an answer. There are various reasons for this, beginning with the fact that technology has not reached the required level at the time the problem has appeared or the theoretical understanding at this point in time was not yet mature enough to help in resolution of the problem. On the other hand there are many reasons due to the environment where the solution was developed, like software policies of a collaboration, design choices made far before the problem was known or simply for legacy reasons like maintenance costs. One should not take this as a set of excuses rather they should be accepted as the environment constraints of large software project(s). Many of these conditions apply in general to large scale software designs and developments and must be considered among other influences during software development.

The XML related technologies for data storage have been already briefly introduced in the abstract of this document. The current *state of the affairs* is that they became integral part of many High Energy Physics (HEP) applications today. It was not always like that. At the time of my doctoral studentship the word ***XML*** was not fully understood by many professionals in HEP and not even by many computing scientists. Since that time, the time of the working draft of XML specification 1.0[60], the XML technologies have proven their relevance in resolving many problems of data exchange and application inter-operability not only in the field of World Wide Web (WWW) but they penetrated the other fields of data processing including the HEP applications. The XML family of technologies is being used today even for the GRID[9] computing facilities at CERN and the other institutes involved in GRID technology.

The focus of the work described in this document is a field of detector description which deals with data sets used to describe very complex HEP detector machines, especially the detectors of the LHC experiments currently under construction. Detector description data plays a very important role in the overall data processing chain of HEP applications. It is used by detector simulation systems to populate their geometrical data models as well as by the event reconstruction frameworks and data analysis applications where they are for example used to resolve imperfections in detector geometry caused by construction or to help in analytic calculations to identify the positions of primary vertices during particle collisions.

This work and the results which have been achieved has been almost entirely bound to the XML and related technologies. I have decided to choose a particular style of presenting it to the academic community. The way of presenting the results follows ***demonstration by example*** principle which I believe makes it easier to bring the reader into the problem domain and facilitates the explanations of the presented challenges and the implementations of solutions for these challenges. The examples come as real-life use cases on which I was working during my stay at CERN.

The first use-case presented in this document is the detector description sub-system developed in context of the complex software framework called **Gaudi**[1] which forms the foundation stone of the LHCb[3] and ATLAS[4] experiments data processing systems. I was given the task to design and develop the first prototype of the detector sub-system in the Gaudi framework. This use-case will show how the XML processing can be used as a distributed data store technology and the set of technological challenges, it poses, will be introduced. The set of solutions for most of these will be described together with the driving forces, constraints and some theoretical and practical background behind them.

In the second use-case we will enter the world of HEP detector simulation applications. We will focus on a domain of geometrical data models. This world is not separated from the detector description at all as it might seem at the first look. Although the simulation engines are often developed isolated from the experiment software frameworks they are usually integrated later on into the experiment software and their geometry data models are populated directly from the frameworks' detector description sub-systems. That is not an easy task and in fact it poses rather heavy integration effort to marry two independent complex software systems having often nothing in common sometimes not even the implementation language!.

There are many technical problems to be solved in order to achieve a smooth co-operation of the simulation engine in one hand and the experiment framework in the other. One of these problems is related to the population of geometry data into a simulation geometry data model. It is a common that the geometry data model of the experiment system and that of simulation engine are different in terms of a type system they use. They may have semantically different handling of the geometry primitives and different set of features each of them can support. The experiment framework often needs to use multiple simulation engines which makes this puzzle even more complex, in particular concerning the population of the corresponding data models. This is where XML comes as very useful tool due to one of its primary design goals and it is providing application independent data exchange.

The second important issue is related to the native geometry format used by the simulation engines and experiment frameworks. It is very often in the form of source code written in the implementation language of the given simulation engine or framework. Any little change required during geometry setup development, tuning and debugging implies re-compilation and re-linking of libraries and more likely the whole application. This way the round-trip development time is drastically increased and developer's productivity is badly affected. Again, the XML based format can be used in this case to allow updates to geometry data without a need to re-build the whole application. This makes it even easier then before to share geometry data among different applications or allows to develop mission oriented tools for geometry data processing like geometry data editors or visualisation programs. Having the data in a form outside of the implementation language makes it a free choice of programming language or toolkit libraries.

These reasons have led me to the application of XML in this field and to launch a project called **GDML**[2][28]. The GDML use-case is showing a different approach to XML data serialisation and is closer to the theoretical grounds than the XML processing machinery of the Gaudi detector-description system.

1. Named after the famous Catalan architect Antonio Gaudi from Barcelona, the father of the famous El Temple Expiatori de la Sagrada Familia church in Barcelona.
2. Geometry Description Markup Language.

In both cases the software solution(s) will be described and will be accompanied by the relevant theoretical models are either being used by the given solution or they show possible alternatives. On the other hand it may be shown that there are theoretical limits constraining the given implementation or making it impractical.

Acknowledgment

This page is dedicated to the people who I owe my warm “Thank you!” for supporting me in making this work to happen.

At first I want to thank my family for creating the lovely environment in which I could relax and recover after the long working days. Especially I would like to thank my wife Beáta and my daughter Anna for their comprehension and support during all the long years which was invaluable source of positive energy in the critical moments of my work. Without this home environment they have built I would not be able to keep my work going on.

My thanks go as well to my both supervisors Milan Krokavec and Pavel Binko. Pavel was keeping his eye on me during my start at CERN and helped me considerably in the initial phase of my work on Gaudi and had kept me focused on the relevant subjects of my doctoral student duties. The help of my university supervisor Milan Krokavec was always giving me the right guidelines along which I could progress towards the goals of my PhD studies and his advises encouraged me many times to be able to step further in my work.

I can not leave out my colleagues at CERN from LHCb experiment especially the members of the Gaudi development team. I would like to thank the chief architect of Gaudi framework Pere Mato for his guidance in implementing the solutions for Gaudi detector description project. Him and my close colleague Markus Frank helped me to raise my C++ programming skills and component based programming understanding considerably. I want to thank the other LHCb people as well including John Harvey the leader of the computing activities in LHCb experiment for his refreshing unforgettable jokes during morning coffee and Marco Cattaneo for his analytic talent to ask the right questions and keeping others focused on the relevant subject. I want to thank as well Olivier Callot who taught me that nothing is perfect and there is still space for improvement as well as that listening to users is the first item to be kept on the list to become a good and valuable software developer.

As my path at CERN got longer I made nice experience in the Geant4 collaboration when assigned to improve geometry model of their simulation toolkit. I have learned incredible amount of new interesting topics and this work helped me to become more professional software developer. I would like to thank John Apostolakis and Gabriele Cosmo my closest Geant4 colleagues who helped me to absorb the complexities of the simulation environment and geometry modelling. I can not leave out madame Maria Grazia Pia for refreshing discussion we had in our office which were amusing and inspiring at the same time. While working in this group I would like to thank Juergen Knobloch the leader of the CERN IT/API software development group for his advises and encouragement which let me grow in my professional career.

I cannot mention all the great people I have met at CERN and I hope they will forgive me not to see their names explicitly listed here. I want to ensure them that my thanks go to them as well and I let them know it the next time I see them again.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1
Introduction

CERN[1], the European Laboratory for Particle Physics, is currently building a new accelerator, the Large Hadron Collider (LHC)[2]. Scheduled to enter operation in 2007, experiments at the LHC will generate some 5PB of data per year with data rates ranging from 100MB to 1.5GB per second. Data taking is expected to last 15 years or more, leading to a total data sample of some 100PB. Designing a system that can handle such enormous data volumes implies a solution that can theoretically handle at least one order of magnitude more data than is currently foreseen, namely 1EB. For some time, the assumption has been that these new experiments will break with tradition and adopt object-oriented solutions. In order to understand whether object-oriented technology is really suited to the High Energy Physics (HEP) environment, a number of research projects have been established. These are addressing areas such as the development of an OO tool-kit for detector simulation, [10] for example, understanding how object-oriented technologies could be used to address the data management issues involved, and to provide the overall offline environment that is required, including analysis and visualisation tools.

1.1 HEP computing environment

HEP experiments are well known by the fact they gather and process very high volumes of physics data acquired either during on-line data acquisitions done by detectors or by simulation programs. There are other data sources in this environment such as detector description data, alignment[1] data, meta data and many other types of data. The largest portion of the data volumes is event data measured by particle detectors. The second in the row is usually the calibration[2] and slow-control[3] data.

A data-flow analysis has been made of the computing system in context of LHCb experiment[21][26], in order to identify the key tasks and data flows. For example, the estimates of computing requirements are made by analysing existing LHCb simulation, reconstruction and analysis programs, and, in cases where they do not yet exist, by extrapolations from the similar code developed by other experiments. Results show that the total CPU requirements needed for simulation, for Level-2 and Level-3 triggers,

1. Actually the better name would be misalignment data because the data describe imperfections in detector geometry introduced during its assembly.

2. Calibration data are used to suppress the imperfections (such as electronics noise) of a measuring apparatus.

3. The data describe environmental conditions in the experimental area and of the detectors (temperature, pressure or voltage level of power sources)

Table 1.1 Raw data volumes and rates of the LHC experiments[21]

	ATLAS & CMS	**ALICE**	**LHCb**
TIME INTERVAL	**1 MB/event, 100 Hz**	**40 MB/event, 40 Hz**	**100 kB/event, 200 Hz**
1 second	100 MB	1,6 GB	20 MB
1 minute	6 GB	100 GB	1,2 GB
1 hour	360 GB	6 TB	75 GB
1 day	8,6 TB	140 TB	1,7 TB
1 year[a]	1 PB	1 PB	0,2 PB

a. means the amount of useful information collected during 1 year

and for reconstruction and analysis of simulated and real data, amounts to ~ 5x10^6 MIPS. The raw data is written to permanent storage at a rate of 20 MB/s. Expected raw data volume and rates are show in the Table 1.1.

The high volumes of event data are not coming by themselves. They will be produced by the detectors. From the size of the data one can guess the detectors being built will be huge and complex measuring devices. There is a lot of knowledge about the detectors which is required for event data processing and for design and development of the detectors themselves. The detector description knowledge consists of the detectors design and construction documents, geometrical description, hardware characteristics (for example the electronics noise, voltage) etc.

1.2 Generation change (From procedural to Object-Oriented)

It has been already said above that the wind of change is coming to the area of software development in the HEP community. In the mean time the change has gained already its concrete shapes. During a last couple of years the huge Fortran based software systems have been or are being replaced by the modern object-oriented frameworks built by using the well established object-oriented language C++[45]. It did not happen during one night and the effort was enormous and had spread across the whole HEP community from on-line, close to the hardware, groups up to the highest levels of pure physics community working only on the well filtered and prepared data sets holding the information about a desired new particles discovery of which may gain one a Nobel prize.

After initial mistakes and failures trying to build a new generation software for data processing in HEP, the community has learned the hard way how to develop robust and flexible object-oriented software and has learned many of the software engineering practices. During the period, when many of the new projects have raised and fallen, a few of them have established themselves well on the developer and user market. Because the trial and fail period took rather long time the HEP community had enough time to select the ones which best fit their needs and help them to do their job.

When the last generation of OO systems has proven to be strong enough, the next phase has started. The goal was to replace the old Fortran based system, still in use, by the new generation of OO tools. Depending on the maturity of a new tool there were basically two strategies applied across the community, the radical way of stopping usage of the old system and start to use the new one even if it is not complete. The second way was to somehow integrate the old system with the new one and incrementally replace the old modules by the new implementations as soon as the were available.

1.3 The LHCb software infrastructure

The LHC computing challenges strongly influenced the evolution of software development of each of the LHC experiments as described above. In the case of the LHCb experiment the foundation stone of its software infrastructure is based on the architecture centric approach as a way of creating resilient software framework. The software architecture, called GAUDI[30], covers event data processing applications in all processing stages from the high level triggers in the on-line system to the final physics analysis. It is very important that this architecture can work in different environments and configurations. The implementation of this architecture is the robust, object-oriented, framework called Gaudi[11]. The implementation strategy chosen by LHCb is the latter approach from the two described in the previous section. It is based on incremental releases and relying on the architectural patterns identifying set of functional components which are set up at the beginning from the available code base. The functionality of the components is wrapped by well chosen abstract interfaces to de-couple them and keep whole system very flexible. At first this allows to build whole system starting from a minimal set of available functionality and add more features as they get available and the already existing clients of these interfaces thus continue to work without noticing any difference in behaviour.

There is one design choice of GAUDI architecture which makes it an outstanding one among its competitors. The GAUDI architecture maintains separate and distinct description of the ***transient***[1] and ***persistent***[2] representations of the data objects[31]. One of the motivations for this approach has been the requirement for a multi-technology persistency solution such that the best adapted technology can be used for each category of data. The main advantage of this design is that the choice of persistency technologies that does not need to be fixed at the beginning and it allows to switch technologies later if needed or even use more of them in parallel.

1.4 Role of the detector description in the LHCb architecture

The LHC detectors are complex devices. They are being built in order to allow measurements of nuclear particles' collisions being accelerated to very high energetic levels. These collisions will be source of huge amount of raw even data but there is already, before this happens, some other types of data that will be collected or produced. The types of data we talk about here is the data describing the detector apparatus itself, its dimensions, materials, various (environment) conditions data and geometrical data used for simulations of these detectors.

Although the data which belong to detector description are not at the orders of magnitude of the event data but it is still a considerable volume it occupies. All the LHC detectors will consist of millions of

1. The transient data structures live only during the time the program which constructed them is running. Once the application is closed these data structures cease to exist, they are destroyed unless they have been made persistent.

2. The persistent data structures can survive the shutdown of the application which has created them and typically are used later by an instance of the same or a different program which reads the persistified data structures from some storage media.

different construction parts and supports, electronic modules as well as hundreds thousands of cables and connectors. Having this information available is not only for the good of logistics and construction purposes but considerable portion of the data is used by physics applications performing simulation, reconstruction and analysis tasks. For example it is very important to know the exact layout of cables to connectors mappings as these are important for electronic calibration jobs as well in cases of debugging and recovery procedures.

The detector description system poses a certain technological challenge to have it implemented right from the beginning. Unlike the event data where the data sources and data flow is planned carefully the sources of detector data do appear more or less in random places and time during periods the detector simulation, construction and run. The data come from various sources like slow controls, simulations, CAD systems and various construction data bases. They are stored various forms and formats like CAD drawings and construction plans, data bases of cable mappings or even simulation geometries in source code form. In addition to the data mentioned so far there is another type of data produced by detector expert groups which are used then later during the reconstruction to optimize this task.

This makes the design of detector description system more complex. What is important however is to provide a common interfaces for access to detector description data. The GAUDI architecture provides such mechanisms which allow to design and implement such a solution[32].

1.5 Think twice before you cut? No, simulate instead!

No one can start to build such detectors from day to day. The design process for these complex devices starts far before the experiment itself may exist! The design of a detector and its functionality depends strongly on the physics processes it is supposed to measure. To figure out the best estimation and provide some candidate solution the number of various simulations must happen in order to obtain some results for decision making about the future detector. And this is usually only the first round of many more to come. This process is incremental with many iterations and loops of verifications at various levels. The important fact is that simulations are done almost at each iteration again and again. It is simply a ***must have*** software component in the high energy physics experiment software machinery.

There is a number of various simulation engines and Monte Carlo generators used across the whole community. Some of them are very focused into a very specific physics processes and then there are, so called, general purpose simulation frameworks which offer much broader spectrum of physics processes and features. Very often the latter ones offer an interface to the specific ones if such a functionality is needed but the implementation inside the framework is not efficient or does not fully cover the required area of physics.

Clearly, the general purpose simulation systems provide more features and many of them are targeting the big collaborations. One of the features is that the simulations jobs can be run in so called mass production mode. Basically this means that the whole cluster of machines can be running many simulation events in parallel. This is perfectly possible as the simulation events are statistically independent units. This does not require support of distributed computing software packages like PVM or MPI. What is important is a clear way to get access to the input data and parameters so a simulation job can be properly configured and run after being assigned to a given CPU or a node in a cluster of workstations. Unfortunately the form or format of the input data is not unified or following a standard.

Basically each general purpose simulation engine maintains its own set of input formats for various types of inputs.

This creates a number of problems inside the experiment's software chain because it either has to support multiple formats for each simulation engine in use or to develop an interface to hide the incompatibility and provide a unified view for its users. The latter is of course more preferable as it takes away the burden from users to know about multiple input formats required by different systems.

There are not really many of the general purpose simulation engines. One can count on one hand the ones being used in production. They are, however, all working internally using non-trivial data structures they use to represent geometry of a simulated device as well as the materials the modules are built from and finally the physics processes used to model various physics behaviours. The reader might have guessed already that the situation with the internal data models is not different from that of the input formats. Exactly! The same kind of troubles is caused by these differences. As in the case of the input formats the integration efforts on the experiment's framework side is not ignorable. Some way of unification or providing a common way to work with the different simulation engines is needed in order to make life of users easier.

1.6 Thesis goals and outline

The paragraphs above have briefly shown the main characteristics of computing in high energy physics experiments. The focus was kept on the environment where this work has been done. The previous section tried to introduce reader to the number of problems which complicate computing tasks of this community. The use-cases which are supposed to demonstrate this work's contribution try to deal with some of these problems and improve the situation. The goals of this work can be briefly summarized by the following points:

1. Provide a new design of object-oriented data model for detector description
2. Prototype implementation of the object storage based on the data model above
3. Investigate the data storage access patters for local and remotely stored detector data as well as the possibilities of detector data replication
4. Investigate possibilities of generic approaches to common data exchange for detector description data

In the following the outline of this document is presented with a brief description of each chapter:

- In chapter 2. we describe in detail the solution for Gaudi detector description.
- In chapter 3. we describe the implementation of GDML processing engine.
- In chapter 4. we compare the two implementations using various criteria.
- In chapter 5. we discuss the relevant theoretical models.
- In chapter 6. we mention the related work in the area of detector description
- In chapter 7. we summarize the outcomes of this work and discuss possible improvements and future research directions in this area referring to the latest research results in the related fields.

The text provides pointers for the technical details or theoretical background in the corresponding appendices or bibliography.

Chapter 2
Use-case: Detector Description in Gaudi Framework

The use-case of detector description solution of LHCb experiment will be described in this part. In order to understand the implementation context the Gaudi framework will be introduced and detector description sub-system will be described in more detail. After being familiar with the environment the implementation will be explained. For detailed description of the Gaudi framework refer to [22][23][26].

2.1 Gaudi framework

The goal of the project is to build a framework which can be applied to a wide range of physics data processing applications for the LHCb experiment. It should cover all stages of the physics data processing: physics and detector simulation, high level software triggers, reconstruction program, physics analysis programs, visualisation, etc. There is also the coverage of a wide range of different environments such as interactive non interactive applications, off-line and on-line programs, etc.

It spans the domains of on-line and off-line computing. It includes the algorithms to filter interesting events from background (so called high level triggers), as well as the full reconstruction and analysis tasks. The overview of these tasks are shown in Figure 2.1 using a data flow diagram. As it can be seen, the event data processing system will consist of a series of processes or tasks that will transform the data collected from the detector into physics results. The data processing is done in various stages following the traditional steps: data collection and filtering, physics and detector simulation, reconstruction and finally physics analysis. The development of all these data processing tasks will involve in one hand computing specialists to build the framework and the basic components and on the other hand the people specialised on each sub-detector that will build the specific code which will be needed for the reconstruction, simulation of each of the sub-detectors and its final combination. In addition, there will be the people developing the data analysis programs to produce the final physics results.

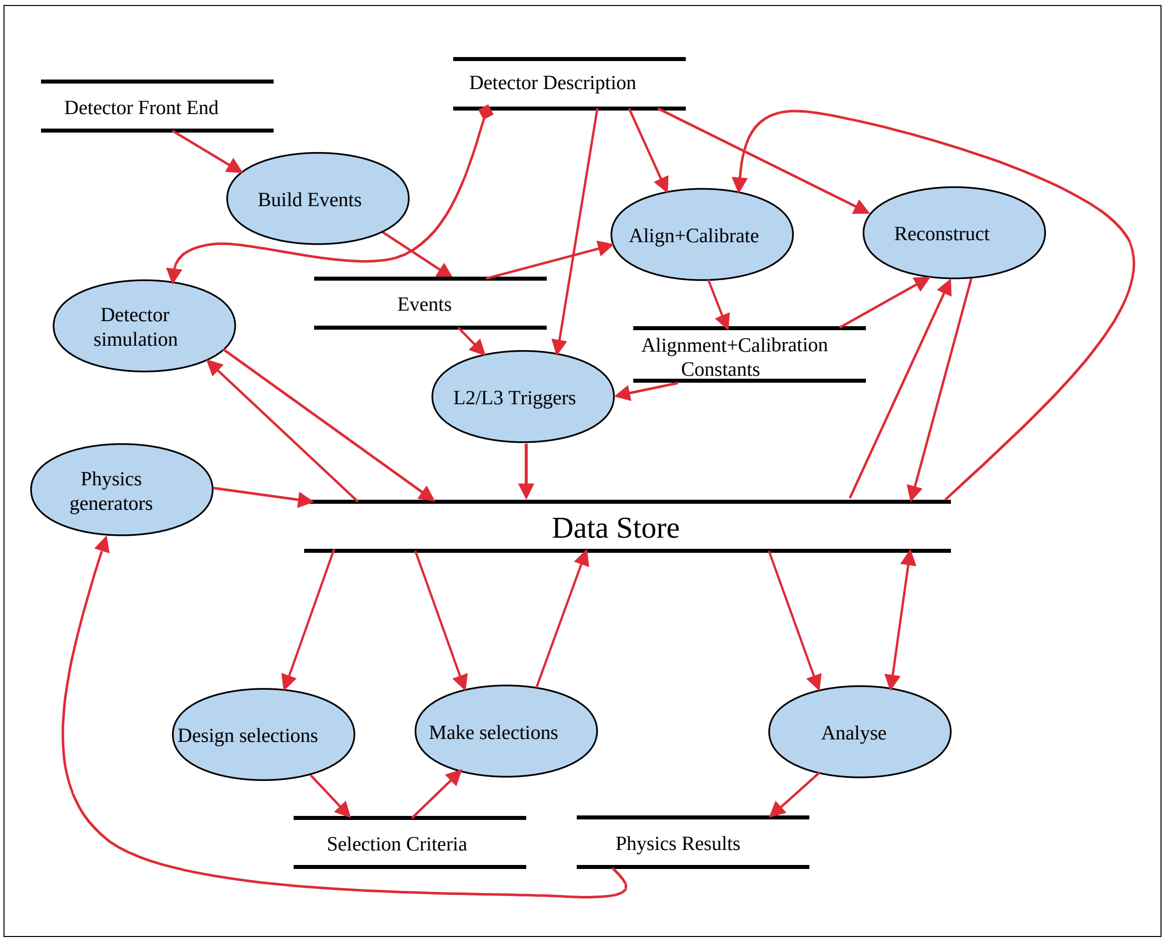


Figure 2.1 Processing data flow of the LHCb experiment

There are four categories of the people who work with the framework. This categorisation is not intended to be exclusive, it is a categorisation of interaction rather than of people and many people will belong to several groups:

1. Physicists. These people are principally interested in analysing reconstructed data and getting results. They produce histograms, statistical distributions which they fit to extract parameters, etc.
2. Physicist developers. These people will contribute to a big fraction of the system code in terms of number of lines. Their principal occupation is to implement components within the provided framework. These components are such things as: Detector simulation and response code; reconstruction code, etc. This activity requires more knowledge of the framework than that required by the average physicist user.
3. Configuration managers. These people are responsible for the management of data production (Monte Carlo, Reconstruction,...), versioning of detector geometry, calibration, alignment, etc. etc. They do not necessarily need to write much code, but they will probably need to be conversant with database management tools.
4. Framework developers. These people are responsible for the design, implementation and maintenance of the framework itself.

The architecture of the system is described in terms of the identified components and their interactions. A software component is a part of the system which performs a single function and has a well defined interface. Components interact with other components through their (abstract) interfaces.

The notation used to specify the architecture is the Unified Modelling Language (UML). This notation is not completely adequate for describing architectures in general, but in this case it seems to be sufficient and it has the advantage that is widely known, thus we no other notation is needed.

2.1.1 Major design criteria

Before the description of the architecture is done the set of design criteria and strategic decisions has to be documented.

2.1.1.1 Clear separation between “data” and “algorithms”

Despite the intention to produce an object oriented design, the decision has been made to separate data from algorithms. For example, thinking to have “hits” and “tracks” as basically data objects and to have the algorithms that manipulate these data objects encapsulated in different objects such as “track_fitter” or “cluster_finder”. The methods in the data objects will be limited to manipulations of internal data members. An algorithm will, in general, process data objects of some type and produce new data objects of a different type. For example, the cluster finder algorithm, produces cluster objects from raw data objects.

2.1.1.2 Three basic categories of data: event, detector and statistical data

There are three major categories of data objects. There is the event data which is the data obtained from particle collisions and its subsequent refinements (raw data, reconstructed data, analysis data, etc.). Then, there is detector data which is all the data needed to describe and qualify the detecting apparatus in order to interpret the event data (structure, geometry, calibration, alignment, environmental parameters, etc.). And finally the statistical data being the result of some processing applied to a set of events (histograms, n-tuples, etc.).

2.1.1.3 Clear separation between “persistent data” and “transient data”

A main feature of the design is that it separates the persistent data from the transient data for all types of data e.g. event, detector description, histograms, etc. Physics algorithms should not use directly the data objects in the persistency store but instead use pure transient objects. Moreover neither type of object should know about the other. There are several reasons for that choice:

- The majority of the physics related code will be independent of the technology we use for object persistency.
- The optimisation criteria for persistent and transient storage are very different. In the persistent world you want to optimise I/O performance, data size, avoid data duplication to avoid inconsistencies, etc. On the other hand, in the transient world you want to optimise execution performance, ease of use, etc. Additionally you can afford data duplication if that helps in the performance and ease of use.

- To plug existing external components into the architecture they have to be interfaced to the data. If they are interfaced to the transient data model, then the investment can be reused in many different types of applications requiring or not requiring persistency. In particular, the transient data can be used as a bridge between two independent components.

2.1.1.4 Data centric architectural style

The architecture allows development of physics algorithms in a fairly independent way. Since many developers will be collaborating in the experiment software effort, the coupling between independent algorithms should be minimised. The transient data storage is used as a means of communication between algorithms. Some algorithms will be “producing” new data objects in the data store whereas others will be “consuming” them. In order for this to work, the newly produced data objects need to be “registered” somehow into the data store such that the other algorithms may have the possibility of identifying them by some “logical” addressing schema.

2.1.1.5 Encapsulated “User code” localised in few specific places: “Algorithms” and “Converters”

One needs to take into account the need to customise the framework when it is used by different event data processing applications in various environments. Most of the time this customising of the framework will be in terms of new specific code and new data structures. A number of “place holders” needs to created where the physics and sub-detector specific code will be later added. There are two main places: Algorithms and Converters.

2.1.1.6 All components with well defined “interfaces” and as “generic” as possible

Each component of the architecture implements a number of interfaces (pure abstract classes in C++) used for interacting with the other components. Each interface consists of a set of functions which are specialised for some type of interaction. The intention is to define these interfaces in a way as generic as possible. That is, they should be independent of the actual implementation of the components and also of the concrete data types that will be added by the users when customising the framework.

2.1.1.7 Re-use standard components wherever possible

The intention is to have one single team with an overview of the complete LHCb software system covering the traditional domains of off-line and on-line computing. This way unnecessary duplication is avoided by identifying components in the different parts of the system which are the same or very similar. The standard and existing components are re-used wherever possible.

2.1.2 Overview of Gaudi components

Description of the architecture by an object diagram showing the main components of system is introduced in Figure 2.2. Using object diagrams are not the best way to show the structure of the software but they are very illustrative for explaining how the system is decomposed. They represent a hypothetical snapshot of the state of the system, showing the objects (in this case component instances) and their relationships in terms of navigability and usage.

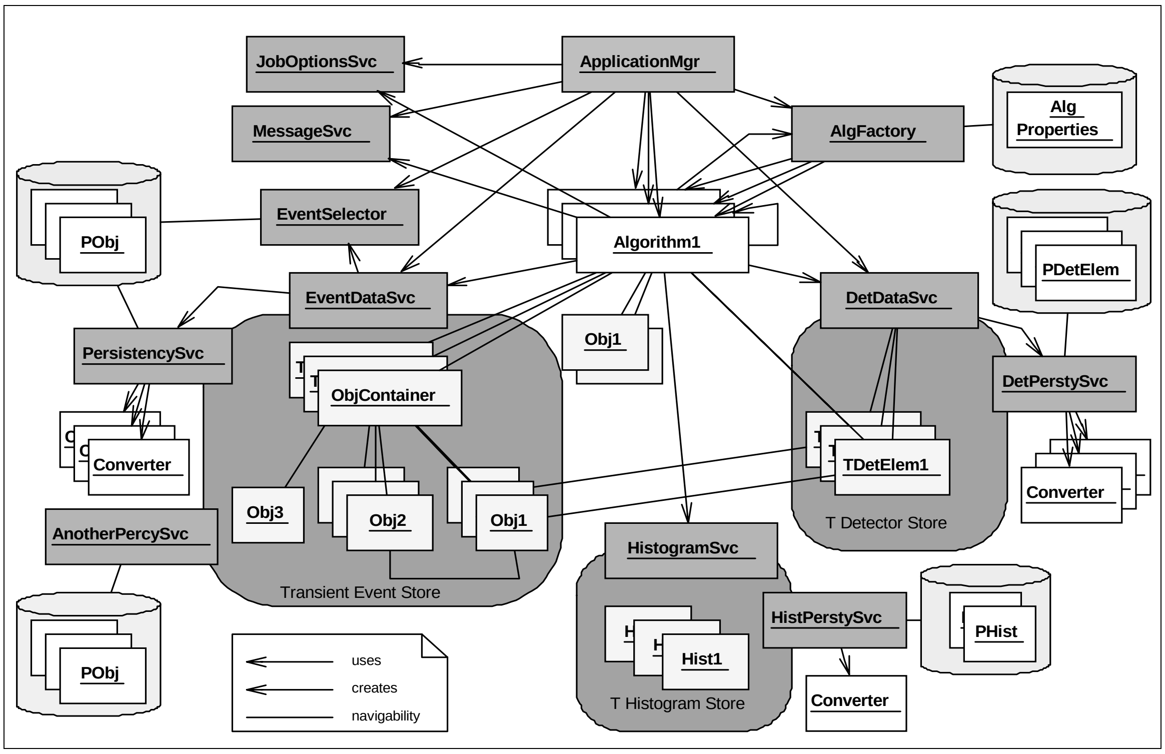


Figure 2.2 Object diagram of the LHCb software architecture, the kernel of any data processing application

2.1.3 Algorithms and Application Manager

The essence of the event data processing applications are the physics algorithms. These are encapsulated into a set of components that called ***algorithms***. These components implement a standard set of generic interfaces. Algorithms can be called without knowing what they really do. In fact, a complex algorithm can be implemented by using a set of simpler ones. At the top of the hierarchy of algorithms sits the ***application manager***. The application manager is the "chef d'orchestre", it decides what algorithms to create and when to call them.

2.1.4 Transient data stores

The data objects needed by the algorithms are organised in various transient data stores. The data are distributed over three stores as shown in the diagram. This distribution is based on the nature of the data itself and its lifetime. The event data which is only valid during the time it takes to process one event is organised in the transient event store. The detector data which includes the detector description, geometry, calibration, etc. and generally has a lifetime of many events is stored in the transient detector store. Finally, the statistical data consisting of histograms and n-tuples which generally have a lifetime of the complete job is stored in the transient histogram store. It is understood that the three stores behave slightly differently, at least with respect to the data lifetime (the event data store is cleared for each event), but their implementations have many things in common. They could be simply different instances of a common component.

2.1.5 Services

Number of components is defined which should offer all the services directly or indirectly needed by the algorithms. The idea here is to offer high level services to the physicist, so that they can concentrate on developing the physics content of the algorithms and not on the technicalities needed for making everything work. This category of components is called ***services***.

Some examples of services can be seen in the object diagram. For instance, there are services for managing the different transient stores (event data service, detector data service,...). These services should offer simplified data access to the algorithms. Another class of service are the different persistency services. They provide the functionality needed to populate the transient data stores from persistent data and vice versa. These services require the help of specific converters which know how to convert a specific data object from its persistent representation into its transient one or the other way around. Other services like the job options service, message service, algorithm factory, etc. which are also shown in the diagram offer the service which its name indicates. They will be described in more detail later.

2.1.6 Selectors

There is a number of components whose function will be to select subset of data according to a user defined criteria. For instance, the event selector provides functionality to the application manager for selecting the events that the application will process. Other types of selectors will permit choosing what objects in the transient store are to be used by an algorithm, by a service, etc.

2.1.7 Main components

The principle functionality of an algorithm is to take input data, manipulate it and produce new output data. Figure 2.3 shows how a concrete algorithm object interacts with the rest of the application framework to achieve this.

The figure shows the four main services that algorithm objects use:

- The event data store
- The detector data store
- The histogram service
- The message service

In addition, a fifth service, the job options service (see) is used by the Algorithm base class, but is not usually explicitly seen by a concrete algorithm.

Each of these services is provided by a component and the use of these components is via an interface. The interface used by algorithm objects is shown in the figure, e.g. for both the event data and detector data stores it is the IDataProviderSvc interface. In general a component implements more than one interface. For example the event data store implements another interface: IDataManager which is used by the application manager to clear the store before a new event is read in.

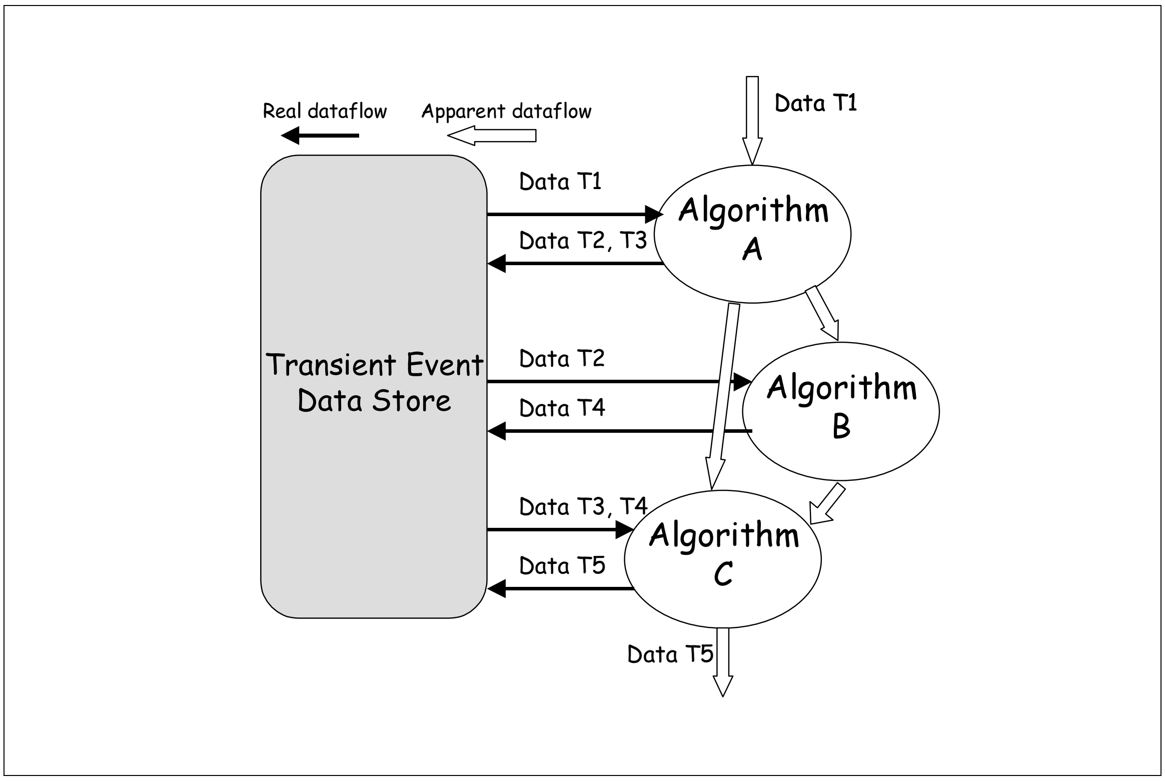


Figure 2.3 Interaction and data flow between algorithms in the Gaudi system

An algorithm's access to data, whether the data is coming from or going to a persistent store or whether it is coming from or going to another algorithm is always via one of the data store components. The IDataProviderSvc interface allows algorithms to access data in the store and to add new data to the store. It is discussed further in chapter where we consider the data store components in more detail.

The histogram service is another type of data store intended for the storage of histograms and other “statistical” objects, i.e. data objects with a lifetime of longer than a single event. Access is via the IHistogramSvc which is an extension to the IDataProviderSvc interface, and is discussed in chapter. The n-tuple service is similar, with access via the INtupleSvc extension to the IDataProviderSvc interface, as discussed in.

In general an algorithm will be configurable: It will require certain parameters, such as cut-off, upper limits on the number of iterations, convergence criteria, etc., to be initialized before the algorithm may be executed. These parameters may be specified at run time via the job options mechanism. This is done by the job options service. Though it is not explicitly shown in the figure this component makes use of the IProperty interface which is implemented by the Algorithm base class.

During its execution an algorithm may wish to make reports on its progress or on errors that occur. All communication with the outside world should go through the message service component via the IMessageSvc interface. Use of this interface is discussed in.

As mentioned above, by virtue of its derivation from the Algorithm base class, any concrete algorithm class implements the IAlgorithm and IProperty interfaces. IProperty is usually used only by the job options service.

Top level algorithms, i.e. algorithm objects created by the application manager are controlled via the IAlgorithm interface. This consists essentially of the methods: initialize(), execute(), and finalize().

The figure also shows that a concrete algorithm may make use of additional objects internally to aid it in its function. These private objects do not need to inherit from any particular base class so long as they are only used internally. These objects are under the complete control of the algorithm object itself and so care is required to avoid memory leaks etc.

The terms "interface" and "implements" have been used quite freely above. The term interface is used to describe a pure virtual C++ class, i.e. a class with no data members, and no implementation of the methods that it declares. For example:

Listing 2.1 Pure abstract class (interface) definition

```
class PureAbstractClass {
  virtual method1() = 0;
  virtual method2() = 0;
}
```

is a pure abstract class or interface. It is said that a class implements such an interface if it is derived from it, for example:

Listing 2.2 Concrete implementation of a pure abstract interface

```
class ConcreteComponent: public PureAbstractClass {
  method1() { }
  method2() { }
}
```

A component which implements more than one interface does so via multiple inheritance, however, since the interfaces are pure abstract classes the usual problems associated with multiple inheritance do not occur.

Within the framework every component, i.e. services and algorithms, has two qualities:

- A concrete component class, e.g. TrackFinderAlgorithm or MessageSvc.
- Its name, e.g. "KalmanFitAlgorithm" or "stdMessageService".

In addition, as discussed above, a component may implement several interfaces. These interfaces are identified by a unique number which is available via a global constant of the form: IID_InterfaceType, such as for example IID_IDataProviderSvc. Using these it is possible to inspect what interfaces a particular component implements.

2.2 Gaudi Detector Description

As was already mention above, in the section 2.1.7 Main components, Gaudi architecture identifies various transient data stores. One of them is the focus of this work, the ***detector data store***. The data sets managed by this store consist of so called ***detector data objects*** which are very important in all event data processing tasks. The data typically consist of the description of a detector in terms of its geometry, the calibration and alignment constants, the environmental conditions and parameters like a temperature or pressure. A framework has been developed to manage in a coherent way all the detector data and to ease the implementation of the specifics of the sub-detectors in the LHCb experiment. The whole development process has been use-case driven. One of the most important use-cases has been that users want have a single detector description for all applications (simulation, reconstruction, event display, etc.). Therefore the information in the database should be a union of all the information needed by all applications; the level of detail should be selected by the client application. The developer using framework should concentrate only with the specifics of his/her sub-detector and leave the commonialities to be provided by the framework itself.

2.2.1 Architecture

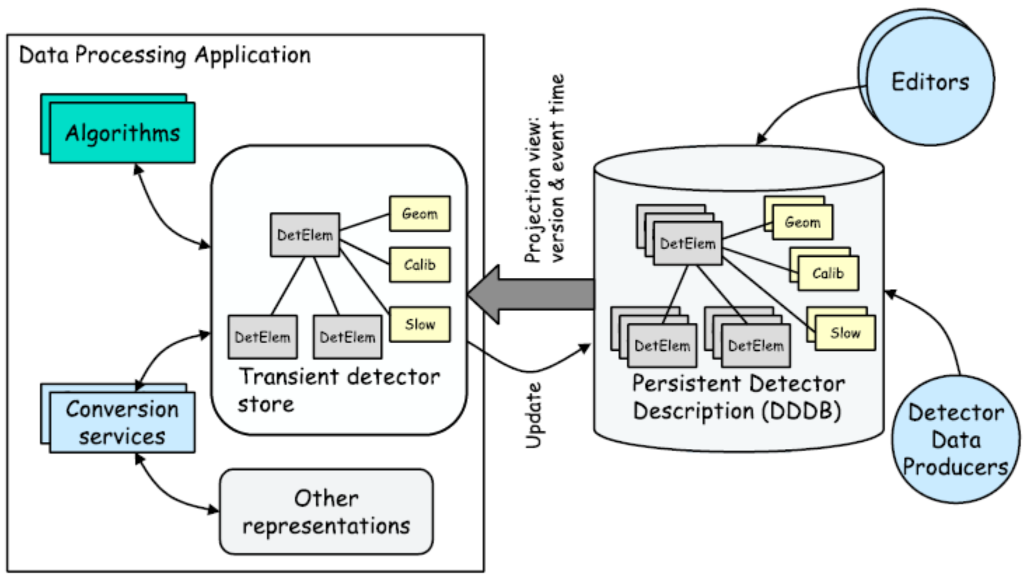


Figure 2.4 Gaudi DDDB store

Separation of the transient data from the persistent representation applies also to detector data. This fundamental design choice of Gaudi is clearly visible also in the Figure 2.6 which shows the Gaudi detector description object diagram. Gaudi architecture is followed here as well, as one can see that *algorithms* access detector data through the transient representation. This representation is obtained from the detector description database via *conversion services*. There may exist multiple kinds of transient representations(e.g. visualisation graphics primitives or Geant4 data model[10]) for the same detector data type thus the conversion services are not exclusively dedicated to conversions from persistent to transient world.

2.2.2 Transient store structure

The detector data store defines two basic views. The logical detector view and the physical detector view. The latter provides access to the physical description items like dimensions, materials as well as detector element specific attributes usually defined by the sub-detector experts.

The logical view defines the user oriented view focused on the subject begin described, e.g. the detector itself. It provides two basic functions:

- Simplified access to the physical parts via hierarchical navigation across whole detector structure.
- Defines detector elements identification so each detector element can be referred to by its logical name.

Figure 2.5 shows how the transient detector data store is organized. The physical organization of the store is defined in terms of data catalogues which play a similar role of directories in a UNIX file system. These can be either organized further in a hierarchical way or simply contain a flat list of data objects. One can see in the figure that ***Structure*** catalogue is hierarchically organized. It is actually the logical view of the LHCb detector described above. There are other data catalogues, for example the ***Geometry*** catalogue is part of the physical view and holds geometrical attributes of each of the detector elements described in the logical view.

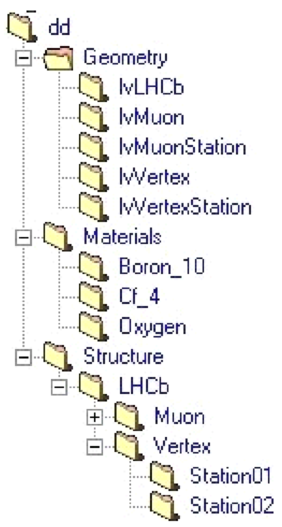


Figure 2.5 Detector description transient store snapshot

In Gaudi any data object registered in the transient data store must be directly or indirectly identifiable by a name. Indirectly here means that for practical reasons is sometimes more convenient to keep data objects inside one transient data container. This is often more efficient from the memory allocation point of view or simply because the data objects of that type are very tiny and to define unique identifier for each of them is inefficient as it would occupy more space than the information of the data object itself. In this case such objects are kept inside one container and the container is the identifiable entity in the data store. The unique name of a particular tiny data object inside the container can be easily obtained by merging the name of the container with the position index of the given data objects in the container.

For example the logical identifier of the first sub-detector inside the Vertex detector in the logical view is `/dd/Structure/LHCb/Vertex/Station01`. The same way one can access physical data objects, for example one access material oxygen using `/dd/Materials/Oxygen`. In the case one deals with the data object container, for example let's consider that data catalogue of materials is a data container one can address a first material as `/dd/Materials/[0]` which means that `Boron_10` material would be addressed.

2.2.3 Detector data access scenario

The typical way an *algorithms* access data n the detector data store is show in Figure 2.6. The data store is populated by detector data from the persistent storage on demand as algorithms ask for the data they need or Gaudi ApplicationManager component triggers synchronisation of event and detector data when a new event is loaded into the even data store.

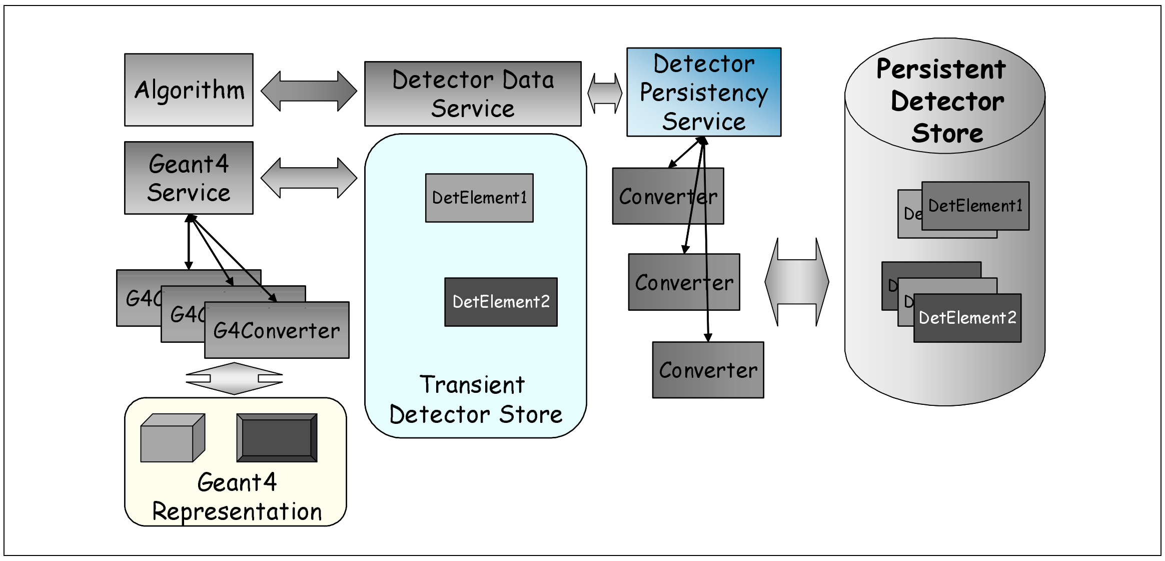


Figure 2.6 Object diagram of detector description components' interaction

The algorithm in the figure demands a detector data by asking the Gaudi *DetectorDataService* by giving it a detector data identifier. The service then scans the content of the transient data store to make sure that the data object is already present. If that is the case the reference to that data is returned back t the algorithm.

If the data has not been already loaded the *DetectorDataService* makes a request to the *DetectorPersistencyService* to load the required data from the persistent storage. The persistency service locates the set of *converters* for the given data type and passes them the logical identifier of the data object to be loaded. If converter exists for such data type the logical identifier is resolved into its corresponding persistent reference and the converter performs access to the persistent storage. After persistent representation of the data is located and accessed by the given converter it transforms it to the required transient data object and returns its reference back to the *DetectorPersistencyService* which in turn passes it back to *DetectorDataService*. When data service has received the fresh data object it registers it in the transient data store and passes its transient data store reference back to the algorithm which asked for it.

The figure shows also the second scenario which does not trigger the access into persistent storage but triggers on the fly conversion from one transient data representation into another one, in this case Geant4 geometrical data object. This way Gaudi transient store can play role of a bridge between two independent components or frameworks.

2.2.4 Detector data model

Model of the transient detector data has been designed strictly following the use-cases and requirements coming from the LHCb user community[27]. It covers all the domains required for the proper processing of the event data. The model has been tuned, however, to specific characteristics of the detector description data. It has been known from the past experience that detector data have a different life-time than the event data. For example when a new event has been loaded and detector description needs to be accordingly synchronised not all data in the detector description data store need to be re-loaded or updated.

On the other hand the detector data, in opposite to event data, can exhibit very rich set of internal cross-references and can share a lot of common pieces of information. For example one can have inside detector setup many different detector elements built from the same material. It would be very inefficient and wrong to load the same material as many times as the number of detector elements. All kinds of such optimisations and requirements have been taken into account during design of the transient detector description data model.

The transient detector data model, for a simplified UML diagram see Figure 2.7, covers the following major domains:

1. **Detector setup**
   - defines hierarchical organization of detector elements
   - more than one setup can be loaded into the same detector data store
   - the principal data object, a generic *DetElement* class can be customized by a sub-detector expert to fit his/her detector specific properties
2. **Geometry**
   - detector geometry is modelled as hierarchy of so called *logical volumes* and *physical volumes* following closely the principles of detector geometry defined in Geant4[10] general purpose simulation toolkit
   - logical volume represents a non-placed volume described in terms of a solid defining its shape and material it is made of plus the (possibly zero) number of children volumes (physical volumes) placed w.r.t its own system of coordinates
   - standard part of geometry data model is a set of useful function offered to algorithms like local-to-global coordinates conversion and vice versa or locating a volume which contain the given point having points' 3D coordinates
3. **Materials**
   - required by logical volumes
   - one can define isotopes, elements which can be optionally built from isotopes and material mixtures which can be built from elements or other mixtures either by specifying the number of atoms of fractional mass for each constituent in the material mix

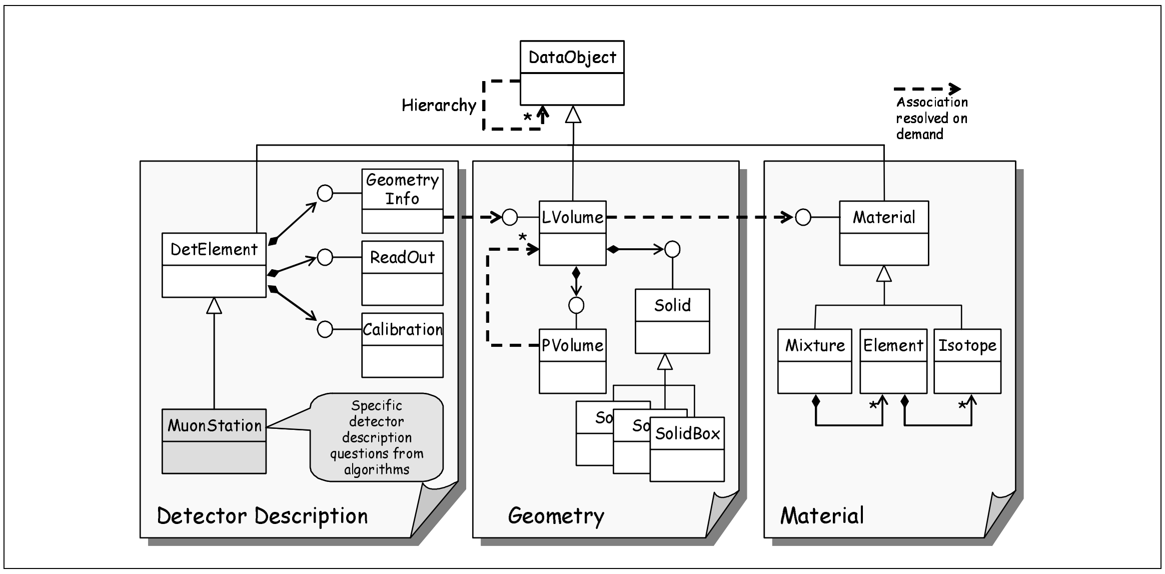


Figure 2.7 The detector data model UML diagram

2.3 Persistent representation of detector data

The benefit of using a system on Gaudi architecture is due to its clear separation of the transient and persistent data. That design choice has its direct impact on users and developers. It shields end users from a persistent technology and simplifies their daily development as they need to interface only to their transient data models. On the other hand a system built using such an architecture allows co-existence of different persistency technologies so a data set can be read from one technology and after some data processing the produced results may be written into another type of persistent storage.

The typical scenario is reading in the detector description data matching the event currently being processed. When data are in the transient store a user may need to visualize the selected detector element and possibly launch a simulation task. In order to configure the simulation, the present transient representation needs to be converted into the representation understood by the simulation engine. At the end of the application run the user might need to store the detector element view into a graphics format for documentation of presentation purposes and write down the data he/she has produced, for example a set of statistical data in form of the histograms. One can see that multiple representations are needed for the data involved which fit best the particular type of data.

Since many different persistent representations need to be supported, the machinery which allows such a functionality must be well defined to allow this in a flexible and scalable manner. This goal has been achieved in the Gaudi design by keeping a technology specific code only on well defined places, the *services* and *converters*.

Detector description framework defines a set of *conversion services*, one per technology type. Each of the conversion services is managing a set of converters, usually one per data type. There may be some generic converters in cases there is a family of data types sharing some common characteristics or make sense to be read and written always bound one to each other. This is a typical case for converters of containers holding polymorphic data from the same class hierarchy.

This conversion services machinery requires however a hint which lets it discover what technology needs to be used and what types of data are to be transformed from/into persistent storage. This hint is provided by the data service of the given transient store which passes it as a part of the request for data conversion down to the persistency service. Gaudi architecture defines for this purpose so called ***opaque addresses***. An opaque address object is created from a data object's logical identifier supplied by the user asking for a data. The opaque address object contains encoded information about the technology, database name and possibly a container or table name. In order to be able to uniquely locate a given persistent data object the opaque address defines its persistent reference which is technology specific, for example it can be positional index inside a container where the object is placed or it can be a primary key into a table in case of relational database system.

A careful reader might have observed that such a machinery for reading and writing of data objects works on very generic foundation and thus allows to build flexible and scalable solutions with many degrees of freedom. The freedom allows then to perform optimisations if needed at various levels and do it per technology or even per each data type!

Because of the features described above this system is not forcing a premature choice of the storage technology in very early stages of the framework development. For the same reason the choice of persistency technology for detector description has not been fixed. The simple argument for this was that nobody can predict what kind of technology will be available in a few years from now. In the case of detector description in Gaudi framework the time period was 7 years to the expected start of LHC.

2.4 The Choice: XML

When first prototypes of Gaudi haven been released into public the number of storage technologies it supported was not high. The focus was on the legacy data formats used by the previous generation of software in order to be able to read and write the existing legacy data. In addition to that the question of database technology was wide open at that time as well. There were three parallel streams of activity in this field each trying to provide a satisfactory solution for LHC data storage in terms of data volume scalability at the first place and how fast the required set of operations on the data can be performed. The first it was the using object-oriented databases where the principal implementation was build using commercial OO database system[12]. The second stream was still trying the well known RDBMS technologies and their new object-relational extensions coming at the time. The last group was trying to address this problem by building a home grown solution[13].

Since the whole high energy physics community has not been definitely convinced by any of the approaches the choice of persistency technology for Gaudi was open. Therefore the detector description has been developed a bit differently. It has been decided that first the transient data model is defined and implemented together with technology independent persistency part of the system. Because the performance issues were not a priority at the time the requirements for the persistency technology were focusing on the ease of use, flexibility and possibility of sharing the data among different applications with the possibility to be imported later into a database system if needed.

After some initial attempts to define a ASCII based human readable format with hand written parser it has been decided that the primary detector description format will be based on XML technology. The XML specification has reached at that time the W3C Recommendation status[14][60]. It was a reasonable choice as it provides features fitting well the functionality required by detector description data. In brief, XML exposes the following set of features:

XML 1.0 standard
W3C provided standardisation process for XML and related technologies which ensures a guarantee that these technologies will be adopted by industrial solutions and supported by the major software vendors world wide.

Powerful expressiveness with relatively simple syntax
XML has been derived from the Standard Generalized Markup Language (SGML)[59]. The design of XML tried to keep its expressive power but a considerable effort has been invested to make it less complex with stricter syntax rules to simplify writing of XML software processing applications. Basically this design followed the 80 - 20 rule which means XML keeps 80% of SGML flexibility and only 20% of its complexity.

Data exchange mission
XML was deigned for application independent data exchange on the World Wide Web (WWW). This original idea has been, however, soon adopted not only in the world of WWW but adopted rather quickly in database community.

Safe investment
The fact that the family of XML standards is maintained by W3C consortium with industrial partners being involved in the standardisation process presents low risk factor in adopting XML technology because it does not require an additional in house expertise and increasing support from industrial vendors promises that a number of XML processing tools will be available.

2.4.1 XML features overview

XML is a meta language. It is supposed to be used to define custom languages describing a specific data domain. For that reason XML provides a relatively simple syntax which allows to annotate data in a human readable format clearly indicating what a given piece of data means. This annotation is done via so call tags attached to each piece of data. In order to make this format to be easily processed by a computer program there is a set of constraints which prescribes the process of attaching the tags to data. The tags can be also called marks that is why it is called a mark-up language. Having for example a simple e-mail message in its raw text form like

```
From: user@domain.org
To: admin@company.com
Subject: The account request

I would like to ask for an account
.
```

one can produce marked-up form of the raw text above as

```
<?xml version="1.0" encoding="UTF-8"?>
<email>
 <from>user@domain.org</from>
 <to>admin@company.com</to>
 <subject>The account request</subject>
 <body>
  I would like to ask for an account
 </body>
</email>
```

One can see that each piece of raw text data has been enclosed by a pair of tags in the form `<tag>...</tag>`. Careful reader may have noticed that all of the XML tags follow proper opening and closing convention. The reasons for that will be explained bellow. The markings, tags, in the XML snippet tell a reader immediately what kind of information they present. The raw text version does not exposes this immediately as a computer program must scan the raw text data itself to find out what it means and requires some hard wired logic to be able to deduce what type of data has been found. In case of XML snippet, a XML parser can scan only the tags and deduce the semantic information immediately without a need to parse the data values.

This was one of the main reasons for inventing XML technology for the Web. The language of the WWW is the famous HTML, invented at CERN by Tim Berners Lee the father of the World Wide Web, which has gone through very dynamic evolution with the global adoption of WWW. The number of web sites and HTML pages has grown exponentialy during a few years and it became very difficult to find an information on the Web. This has led to invention of Web search engines which dynamically discover the new Web sites and pages and cache pointers to them inside their indexing databases. The problem with HTML is that it provides no indication at all what kind of data a web page holds. HTML provides only the formatting and hypertext facilities exploited by the Web browser applications. With the increasing number of information presented on the Web it became very difficult for the search engine to provide a useful classification of the indexed data in order to make search optimisation. It is important that a user search request does not produce a big number of irrelevant results. Apart from having the result page full of wrong hits it causes wasting of Internet bandwidth. Because HTML does not provide any information about the data in a web page (meta data) a new way of presenting the information on the web was required. XML has been designed to fill this gap by providing means of attaching the meta data information to each piece of data on the Web.

The second reason for which XML has been created is the fact that the due to the war of Web browsers on the Internet software market these went into extremes of ability to process any HTML documents very often breaking many HTML syntactic rules. HTML is derived as well as XML from SGML but with focus on simplicity, hypertext support and presentation facilities. It was this design decision which has made it so popular on Internet because writing a HTML page was extremely easy and with hypertext facilities it was possible to build a very complex web site in a short time. HTML has reached its limits due to introduction of many features forced by needs of commercial market and the major vendors of Web browsers competed in supporting as many of them as possible. People writing HTML markup very often did not realize they break many HTML syntactic rules as browsers tried their best in parsing broken HTML so there are many broken HTML pages on the web today. This number is very high and poses the real problem for other internet applications like search engines and web directories.

Therefore design of XML has taken this into account and there are two levels of syntactic safety built into the XML specification. At the simplest level any XML document must be ***well formed***. The XML well formedness means that each XML tag is properly opened and closed as is shown in the snippet of the marked up e-mail message above. All the tags must be properly nested inside their corresponding parent tag apart from the top-level one. The is at most one top-level tag in a XML document. The top-level tag than defines the type of the XML document. For example the top-level tag of the XML snippet above is `<email></email>`.

The second level of XML safety relies on the concept of ***validity***. XML validity means that a given XML document contains only allowed set of tags which are organized according to some prescription. This prescription is implemented in XML by defining so called XML schema document before hand. Originally there was only one XML schema language called Document Structure Definition language (DTD)[1][60]. As XML itself the XML DTD has been derived from SGML DTD format. DTD describes XML document structural constraints and set of valid tags and their corresponding attributes to be used in a XML document valid w.r.t this DTD. Having a XML document which refers to a DTD it is possible to validate the given XML document and say whether this document is valid or corrupted. It has practical implication for the XML document processing because it allows to check automatically by a computer program if the incoming XML data are valid or not. The rule of error handling in case of a broken XML document is quite strict saying that if an error has been observed on the given instance of a XML document the XML processing application should report the deviation from the given XML schema and stop. The validation mechanism for XML DTD is based on the context free grammars adopted from SGML DTD but limited to deterministic content models only. The details of the XML schema internal mechanisms will be discussed in the next chapters.

The last issue related to XML versus HTML case is the ability of visual presentation of the document content. Unlike the HTML tags which hold the visualisation hints the tags of XML document do not say anything about how a given piece of XML data should formatted for human visual consumption. It has been decided that in order to be able to provide such functionality the XML documents can be accompanied by so called XSL style sheets[62] which define how a given XML document should be post-processed. There is already a similar technology available for HTML called cascading style sheets (CSS)[69] which is however able to describe formatting only for the fixed set of HTML tags. The XML way is much advanced than the CSS capabilities because it can perform complex transformations on the input XML data and it can even produce on its output a brand new XML document valid w.r.t a different XML schema than the original one. XML thus managed to separate the data content from its behaviour[2]. This is very important postulate confusing XML newcomers which often thin that one can write in XML a procedural code which performs a function. This is invalid statement and it has to be emphasized that XML describes only data and no data behaviour.

2.4.2 XML based Detector Description in Gaudi

Before the XML has been applied to detector description data in Gaudi framework the complete transient data model has been designed and the transient detector data store prototype has been implemented. The task thus has been to define the way how to map the transient data types of Gaudi detector description model into the XML persistent form. This process had to take into account the design choices made for Gaudi architecture and the guidelines developed along the Gaudi implementation. There were additional factors affecting this process due to the user requirements that

1. There many other XML schema languages available today, they will be described in detail in the next chapters, in particular W3C XML Schema language[66].
2. The same postulate is defined for Gaudi architecture

the syntax should be as compact as possible so it can be typed into an editor of user's choice and it should be easy to learn keeping its adoption easy. LCHb users also required support for the WWW hypertext paradigm in the XML detector description of Gaudi to allow remote references across the whole detector description database.

The implementation phase of XML based detector description persistency consisted of the following steps

Domain decomposition in terms of components, see Figure 2.8

The set of components has been identified and their interactions defined. At this stage the overall structure has been defined and the outcome of this was the class and sequence model of XML converters and set of selected classes to be stored in the XML format. Not all transient classes need to be stored to minimize the volume of the data in XML and reduce the XML processing time involved in reading/writing of the data from/into XML form.

XML schema design for the selected classes using DTD model

The set of DTD schemas has been developed capturing detector data model classes in order to be able to validate the XML data defined by the detector description users. The XML tags defined in the DTD were identified according to the adopted mapping strategy.

Definition of the strategy for mapping of C++ detector data types into XML

Due to the different type systems of C++ and DTD some convention had to be established how to find correspondence between C++ classes and DTD structured types.

Evaluation and proposal of the XML API for C++ language

There were just a very few XML C/C++ APIs available at that time. Basically the only choice was to use either SAX[50] or DOM[51] APIs. The way they work is totally different so some evaluation had to take place first before the final decision could be made. The choice for the implementation was SAX API. The details behind this choice will be described later in this chapter.

Design of remote linking facility

One of the user requirements was the ability to refer to a XML data stored in a different file. The reason for this is to be able to emulate the C++ pointers or references which do exist in the transient world. There is a pure practical reason related to the way users work with the XML data files. It is more natural keep the data for different detector modules in the separate files rather than in one possibly huge XML file. There are some very complex detector modules which consist of many other modules which are still complex more than enough. Populating and keeping separate XML files up to date is much easier if they can be updated independently of each other.

Design of strategy for detector element customizing

The specific detector description can be made available to algorithms by customizing the generic detector element. Customizing is done by inheritance of the DetectorElement base class. The sub-detector specialist can provide specific answers to algorithms based on a combination of common parameters and functionality (general geometry, material, etc.) and some specific parameters. The specialist can "code" the answer by using the minimal number of parameters specific to the detector being described. For example, an Algorithm may need the local position of a calorimeter cell knowing its cell ID. In this case, probably, a simple parametric equation in a detector element method can give the answer.

2.4.2.1 XML conversion services

Conversion between the transient and persistent worlds in Gaudi happens with help of the conversion services and their converter components. The role of a conversion service is to manage a set of converters for a given technology and to dispatch the right converter for a given data type. The converter has to provide a set of data conversion functions depending on the context of its invocation. There are three basic contexts when a converter is activated:

- Read
- Write
- Update

There are other situations when a converter can be activated, for example to resolve or update references to other objects but these will not be described in detail. For full description of this mechanism see [22][23].

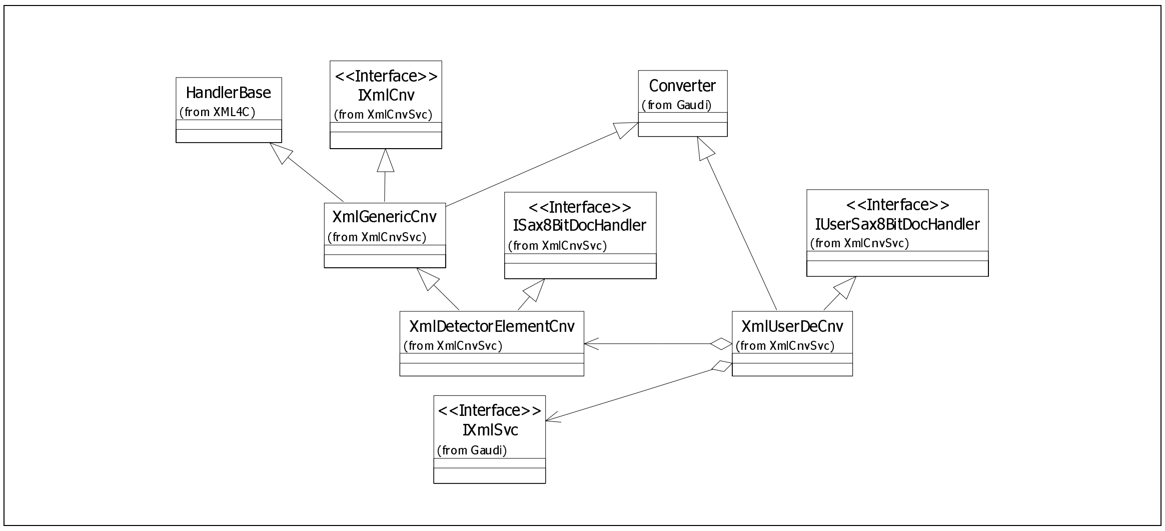


Figure 2.8 Data model of XML converters in Gaudi detector description framework.

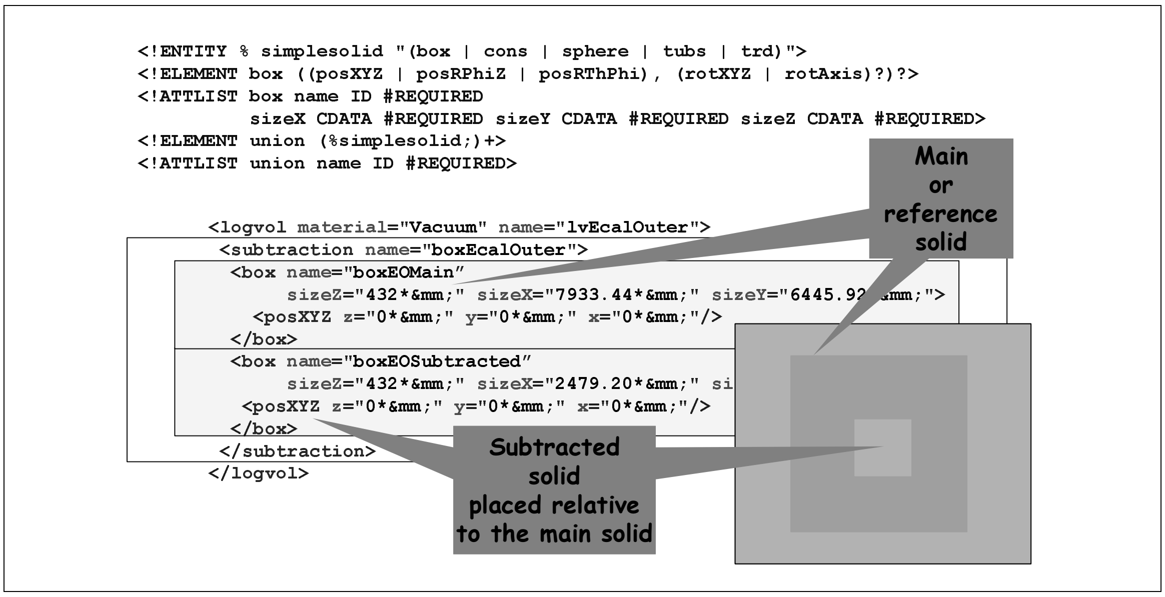
The only place in Gaudi architecture which deals directly with a concrete technology are the converters. Only the converters directly access the persistent storage in the technology specific way. In case of XML they access the XML data using the chosen XML API and after validating the file they try to locate the required XML data inside the file and finally perform the conversion. Since the data are spread across many files, as will be described later in the section explaining the remote object linking, the XML converters had to be optimised from the beginning to minimize the number of access to the XML data files to avoid costly parsing. This has been achieved by the strategy that whenever a new object has been loaded from XML and it contained one or more remote references, this information has been recorded by converter on the transient store. When the request came to load the object behind the reference the right converter has been immediately activated and direct access has been made to the file which was marked in the previously seen reference. This has minimized the need to re-open the file again just to get the embedded reference. More details about the internal workings will be explained in the next sections.

2.4.2.2 DTD definition for detector description data model

Having the DTD schema available helped to fulfil two goals. First it enabled the automatic data validation whenever the XML data files have been accessed. The second goal was it provided reference documentation for the users. By reading the documentation the users could learn what tags are allowed and what attributes they can have.

There were actually multiple DTD files defined one for each domain of detector data model. They were finally combined to a single DTD via built-in DTD inclusion mechanism so the users could refer to it from their XML data files. One of the DTD files defined overall logical structure similar to the structure of transient detector data store. It provided a view to XML data in a fashion similar to that of the transient detector data store. The important tag defined in this DTD was `<catalog>` tag. This tag defines a container like XML type which plays role similar to that of directory in the file system. Recalling the Figure 2.5 there is one top-level catalog and three major ones `Geometry`, `Materials` and `Structure`. `Structure` uses embedded catalogues to allow hierarchical data organization.

The rest of the DTD files defined XML types for most of the classes in the detector data model. For example Figure 2.9 shows definition and example XML data of `<box>` and boolean ***solid*** `<union>` XML types which represent geometrical shape respectively a boolean combination of a volume shapes:



```
<!ENTITY % simplesolid "(box | cons | sphere | tubs | trd)">
<!ELEMENT box ((posXYZ | posRPhiZ | posRThPhi), (rotXYZ | rotAxis)?)?>
<!ATTLIST box name ID #REQUIRED
          sizeX CDATA #REQUIRED sizeY CDATA #REQUIRED sizeZ CDATA #REQUIRED>
<!ELEMENT union (%simplesolid;)+>
<!ATTLIST union name ID #REQUIRED>


      <logvol material="Vacuum" name="lvEcalOuter">
       <subtraction name="boxEcalOuter">
        <box name="boxEOMain"
             sizeZ="432*&mm;" sizeX="7933.44*&mm;" sizeY="6445.92 &mm;">
          <posXYZ z="0*&mm;" y="0*&mm;" x="0*&mm;"/>
        </box>
        <box name="boxEOSubtracted"
             sizeZ="432*&mm;" sizeX="2479.20*&mm;" si
         <posXYZ z="0*&mm;" y="0*&mm;" x="0*&mm;"/>
        </box>
       </subtraction>
      </logvol>
```

Figure 2.9 Example of DTD definition in Gaudi detector description framework

The full description of the DTD including UML diagrams can be found in [24].

2.4.2.3 C++ to XML mapping strategy

The type systems of DTD and C++ are different. DTD can define only structural constraints of the valid set of tags in a given class of XML documents and has primitive reference model. C++ on the other hand allows to define abstract data types (classes, structures), sub-typing relation (inheritance), aggregation of types by composition of by reference (data members of complex types or data members as pointer or references to other objects of complex data types). There is no direct correspondence between these two worlds so a convention is needed to bridge this gap and find a way of coming from C++ classes to DTD XML elements. The Table 2.1 shows the mapping rules adopted for Gaudi detector description data model.

Table 2.1 XML to C++ and vice-versa mapping used in Gaudi detector description

C++	**XML**
Class	XML element
Class data members of base types	XML element attributes
Class data member of a complex type	XML element in content model of the parent element
Reference, pointer	XML reference of IDREF type holding the URL of the remote XML element

There is however one sensitive spot in this strategy. The question is whether to use attributes or embedded XML elements for class data members. There is no definitive answer to this question even among the XML experts themselves. The strategy adopted in Gaudi is just one of many possibilities which could be applied here. In the Appendix A is shown by example how this strategy is applied for DTD modelling of C++ classes.

2.4.2.4 Implementation of XML converters

After the mapping from C++ to XML has been defined a set of XML converters has been implemented. Before actual implementation started the evaluating of the available XML processing tools has been done. The most mature tool for XML processing in C++, found at that time, was XML4C XML parser developed at IBM XML laboratories. This tool has been later donated to Apache project for its suite of open source XML processing tools[15] and today is known as Xerces-C XML parser. This parser supported from the beginning both standard APIs SAX and DOM and later has been extended with SAX2 support. The Table 2.2 shows brief comparison of SAX/SAX2 and DOM APIs.

The final choice for implementation was to use SAX based implementation. The main reason for that choice was uncertainty about the memory requirements of DOM based implementation. DOM requires the document to be present in memory and in the way Gaudi is working there may be many XML documents open at the same time. Even if documents are kept on disk as 8 bit ASCII encoded files the DOM will expand the size of document in memory by factor of 3 to 10! The actual factor depends very much on the document itself the number of child elements, number of sattributes and structure of text content inside the XML elements. The reason for the size explosion of XML data in the DOM case comes first from the fact that by XML specification the XML processors should treat any XML data as Unicode[16] characters which are 16 bits long in opposite to 8 bit ASCII characters. This is already a factor of 2 in size. The explanation for factor 3 as the lower bound is that DOM data structures themselves require some memory in order to build the proper DOM document tree. DOM is a great API for the document oriented XML applications which read, update and write back XML documents like XML editors of browsers.

Table 2.2 Comparison of two common XML APIs

Feature	**SAX/SAX2**	**DOM**
Parsing method	event based[a]	load whole file
Requires whole document in memory	NO	YES
User API type	push style	tree navigation style[b]
Preserves document structure	NO	YES
Memory footprint	minimal	can be a problem if the file is big

a. incremental parsers do exist, Xerces-C can be run in incremental mode

b. can be push style too using DOM tree visitor which calls application call back a la SAX

The SAX API relaxed the worry about the memory footprint and there was possibility to parse documents incrementally which means that once the required data have been localized in a XML file the SAX parser has been postponed. When there was a need to load more data the parsing has been resumed and more XML data have been loaded and so on. This way there were multiple instances of converters running each of them driving its own instance of SAX parser. At that time there was no urgent need to write XML data out of detector data store so SAX was sufficient[1]. See Appendix B for detailed description of the SAX and DOM APIs.

The simplified view of XML processing chain in Gaudi detector description implementation is shown in Figure 2.10.

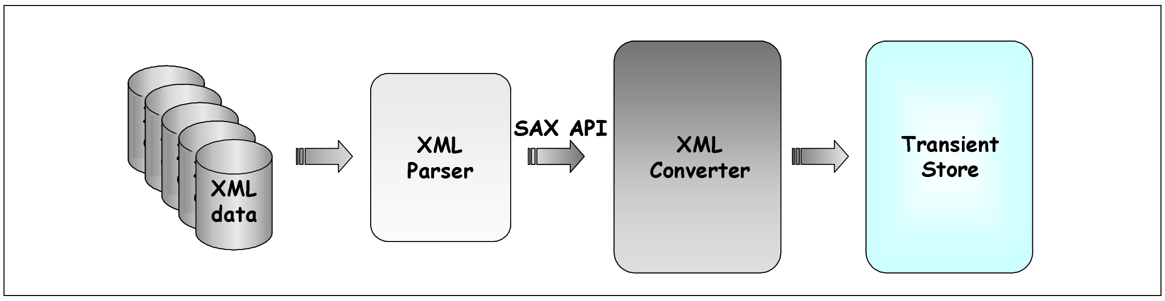


Figure 2.10 Conversion process for XML based data in Gaudi detector description framework

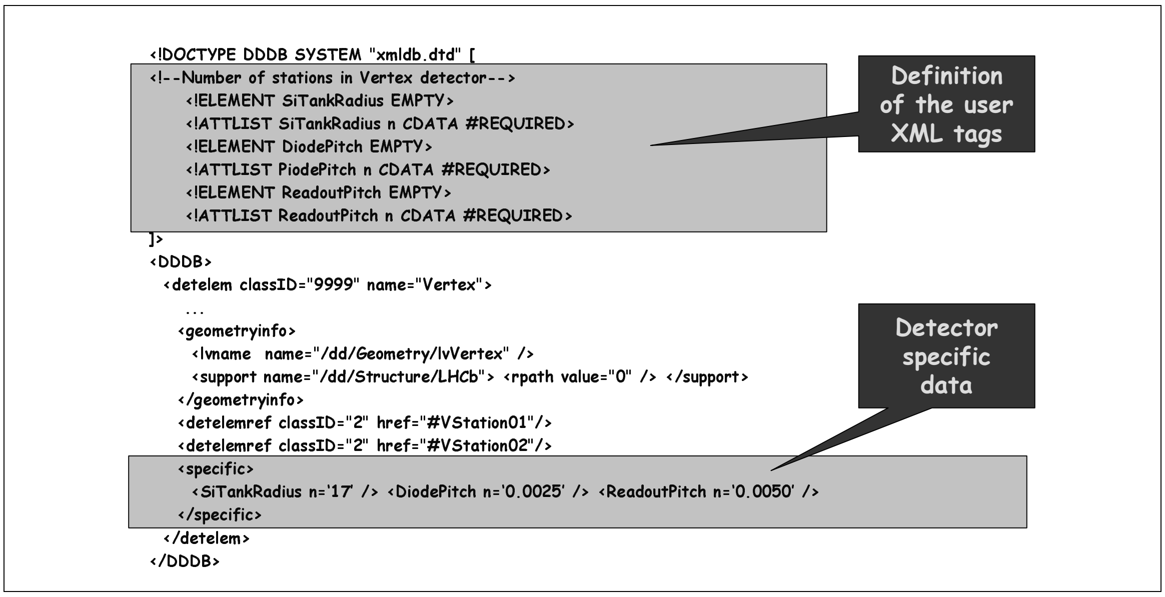
There is typically one XML converter per detector data type aligned with the C++ to XML mapping rules. What has been done in addition to the standard practice in XML programming is the introduction of ASCII only layer between Xerces-C APIs and Gaudi code. The reason was that none of the HEP C++

1. The recent releases of Gaudi detector description use DOM based converters but with tunable caching mechanism behind. The transition to DOM based converters happened more than one year after I left LHCb collaboration so it is not discussed in this work. The detailed description of the DOM based converters can be found at URL: http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/DetDesc

tools is truly Unicode based (not even all operating systems are) and this layer provided the shield for non-Unicode code. This layer can be, however, very easily removed if needed in the future.

2.4.2.5 User defined customizing of detector elements

The detector data model tries to define all the data types in order to cover as much as possible of the detector description domain. Nothing is perfect in our world, and there had to be left one degree of freedom in the detector description data model. The subject in question is the possibility of having the user defined detector elements without a need for heavy programming which is the job of core Gaudi developers. The solution to this problem in XML based detector description is shown in Figure 2.11.



```
<!DOCTYPE DDDB SYSTEM "xmldb.dtd" [
<!--Number of stations in Vertex detector-->
     <!ELEMENT SiTankRadius EMPTY>
     <!ATTLIST SiTankRadius n CDATA #REQUIRED>
     <!ELEMENT DiodePitch EMPTY>
     <!ATTLIST PiodePitch n CDATA #REQUIRED>
     <!ELEMENT ReadoutPitch EMPTY>
     <!ATTLIST ReadoutPitch n CDATA #REQUIRED>
]>
<DDDB>
  <detelem classID="9999" name="Vertex">
     ...
    <geometryinfo>
      <lvname  name="/dd/Geometry/lvVertex" />
      <support name="/dd/Structure/LHCb"> <rpath value="0" /> </support>
    </geometryinfo>
    <detelemref classID="2" href="#VStation01"/>
    <detelemref classID="2" href="#VStation02"/>
    <specific>
      <SiTankRadius n='17' /> <DiodePitch n='0.0025' /> <ReadoutPitch n='0.0050' />
    </specific>
  </detelem>
</DDDB>
```

Figure 2.11 Emulation of inheritance by using DTD internal subset

The solution on the C++ implementation is relatively simple because one can simply inherit from a base class defined for this purpose and which deals with all low level details. Such a user defined detector element extended according to the user requirements can be directly used as any other detector description classes. What has to be done in addition on the user side is to implement a set of methods which allows to initialize properly the additional information defined by the user. There are many ways to achieve this in C++. The problem is more visible on the side of XML based persistent storage. The DTD is defined once and its modifications are controlled and carefully coordinated not to break the rest of the system. By using internal DTD subset inside the XML file the new custom tags can be defined and placed inside the `<specific>` tag and there is a well defined procedure how to implement in easy way the user defined XML converter for a customized detector element[23].

Chapter 3

Use-case: Geometry Description Markup Language (GDML)

The Gaudi detector description use-case has shown the importance of the detector data for the high energy physics data processing applications. It shows the overall view touching all the corners of a typical software environment one can find in any HEP experiment. The context of the Geometry Description Markup Language is placed a bit deeper in the software chain. It is close to the environment of detector simulations where it has been created. It is also more focused, to cover only the geometry data used by the simulations engines. Nonetheless, its use-case deals with problems at the same level of complexity as the Gaudi detector description use-case.

Why there is a need for such a language? What is the benefit of having a common XML format for geometry data? These questions have been partially answered already by reasoning behind the design of detector description use-case. There are, however, many other projects which keep their detector description data in XML, see Chapter 6 Related work.

During the past years these projects have built their own dialects of XML. There is nothing bad about it if users and developers work only in context of a single project. The situation in HEP community is a bit different. Many individuals are involved in multiple projects at the same time. The consequence is that the people need to be familiar with multiple XML dialects in order to be able to do their work. So far, there is no common XML format in HEP which allows to share data across application or project boundaries. The reason for this may be the fact that HEP collaborations and their developer groups started to apply XML based technologies only recently and without much experience. It can be said that these days most of the projects matured enough and more effort will be spent on integration tasks then development of new features.

Very similar situation is in the simulation community. The classical form of geometry data which exists today is in the source code form (C, C++, FORTRAN, Python). There are of course other sources of geometry data like various text based and proprietary semi-structured data formats and geometry data stored in relational databases. There is certain analogy here. There is no common way to share geometry data between any of the simulation tools in use.

Being actively involved on both sides I have started the GDML project[28] in believe that this project fills the gap of problems described above.

3.1 Requirements

Before any work on GDML project has begun an open discussion was launched among the users and developers actively working on XML based solutions across high energy physics community. The aim of this discussion was to collect the user requirements about the common XML geometry description format. The focus has been intentionally narrowed down to the scope of geometry and materials. The requirements collection process focused on the most generic features. The result of this activity in form of informal requirements is briefly summarized by the following points:

Geometry model

High energy physics community is traditionally used to work with many geometry models. The most common way of geometry representation in program's memory is a tree based model. Unlike the geometry models of various CAD systems which are often organized in a flat way the HEP geometry tree like models exploit the tree topology for navigational purposes. The navigation is for example used for either interactive browsing or by simulation engines to propagate a particle through whole detector geometry and pass the information about the environment at the current position down to the physics processes.

Hierarchical geometry model must be assumed by a XML geometry description format.

Numerical expressions

The geometry data involve regular use of values of various quantities with units. In order to describe some complex geometrical setup users apply various numerical expressions which contain variables holding the quantities defined before hand. Very much like in an imperative programming language. For the reasons of easier maintenance is preferred to keep pre-defined set of globally known mathematical and physical constants in single place and build a set of numerical expressions from these later when needed. XML contains only Unicode[16] data and no data behaviour defined. Clearly the regular expressions need to be evaluated but pure XML parser simply can't perform such an operation. For this reason the use of numerical expressions has been agreed and responsibility to evaluate them was defined to be done by a GDML processor in addition to GDML parsing.

The use of numerical expressions must be supported and evaluation of numerical expressions should be done by a GDML processor before the data are passed into the client application.

Uniform treatment of 3D transformations

There are multiple ways of interpreting transformations in 3D space. Many applications working with geometry do it in a compatible way but this behaviour is not guaranteed in general. The problem of geometry exchange format is that geometry data can be exported from a different application then the application consuming it. The possibility of ambiguous handling of 3D transformations led to the following requirement.

The exchange format must define the uniform rules for treatment of transformations in space and it must be ensured that GDML data being exported/imported follow the rules.

Identification and linking mechanism

Having the possibility to uniquely identify geometry elements and refer to them from remote locations using hypertext links is essential. The local linking mechanism is built into XML via ID/IDREF mechanism. W3C XML Schema[66] has more advanced mechanism based on key/keyref. More sophisticated XML technologies like XLink[63], XPath[61], Xpointer[64] look promising but not yet considered for geometry data exchange.

Geometry data format must provide means to express unique identity for geometry elements. It should be possible to define references pointing to such elements.

Application independent model for geometry

It was considered a bad idea to bind design of geometry exchange format to philosophy of a particular geometry model. The exchange format should support first the common ways for geometry modelling with focus on its modularity and extensibility in order to support less common practices if required. Neither of the extremes, nor the minimal common denominator nor “The Whole World” approach, were encouraged for geometry exchange format.

The geometry exchange format should be modular, open and extensible in order to support evolution driven by use-cases and requirements.

Simple application binding

At some moment the data stored in geometry exchange format must to be delivered into the client application. In order to make the GDML adoption easier it is essential if GDML defines an easy application binding strategy. It would gain lower maintenance costs and save human resources spent on developing redundant code.

GDML should provide an easy way to integrate its processing software into a target application or framework.

3.2 GDML Schema

GDML schema has been defined using W3C XML Schema and its design has been driven by the set of colleted requirements. Figure 3.1 shows the GDML schema component breakdown and its connection to the major application frameworks and libraries in HEP community. As one can see the schema has been defined in a modular way. XML Schema features allowed object-oriented approach for GDML design in a similar way an object-oriented application design is done. From the very beginning the design of GDML focused on providing very generic foundation on top of which other GDML schema components can be built. The concept of reusability has been applied as much as possible.

In order to come up quickly with a working schema prototype some features have been omitted from the first public release of GDML schema, namely the remote element linking. Schema has been released in a simpler version and supports geometry element unique identification and references only in scope of a single GDML data file. Most of the other GDML schema components have been defined and there are already plans for some extensions demanded by GDML clients. The whole process of GDML extensions is under control and is discussed with all involved parties.

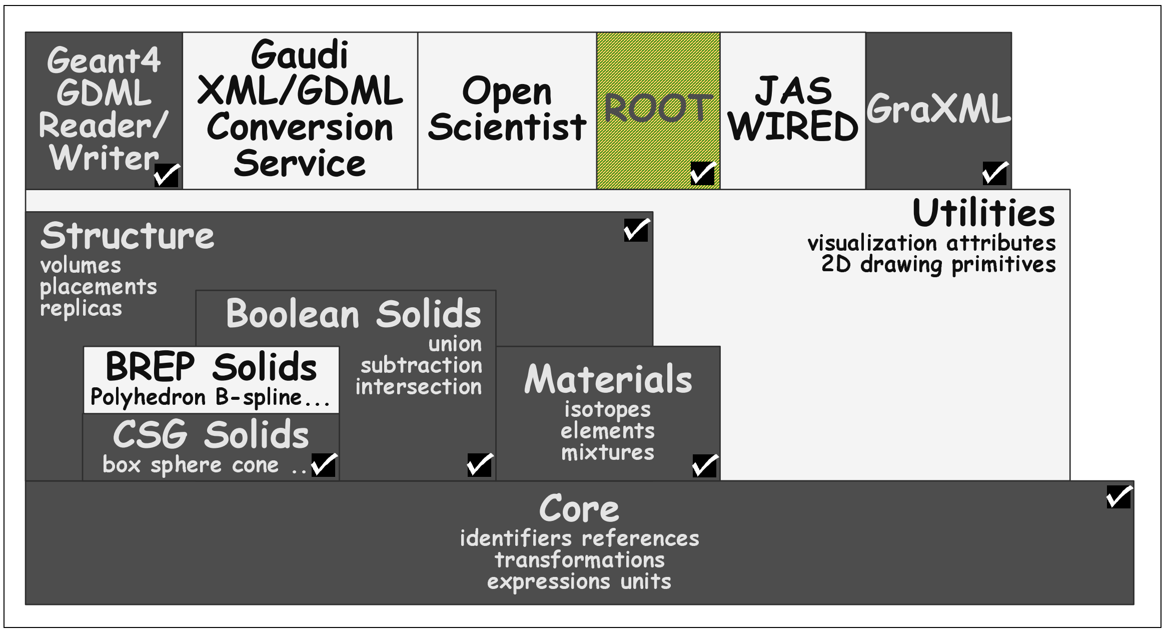


Figure 3.1 GDML Schema components[a], implementation status and its adoption status[b].

a. The components in dark colour are implemented

b. The OK sign marks the components for which underlying support in GDML processor exists

3.3 Processing Architecture

The requirement for simple application binding was one of the strongest on the list. All developers who used to be involved in XML related programming knew that application binding for XML may require considerable effort, especially for XML schemas with many elements. In the case GDML the number of elements is not yet high but can be already considered an issue from application binding point of view.

Due to the requirement the development of GDML schema has been done together with the design of application binding. The driving forces behind the application binding architecture are:

Separation of XML dependent code

GDML binding can't force a choice of specific XML parser tool to its clients, so the processing engine must not depend on any particular XML API.

Minimize the amount of the application specific code

GDML is generic format and does not rely on any concrete application data model. However, the application native data model must be populated in some way from a GDML input. GDML should not require an extensive amount code on the application side.

GDML processing code should minimally suffer from schema updates

Changes to the GDML schema will inevitably happen which implies update of GDML processing code. GDML architecture must take this into account and make sure that effect of changes to the schema have minimal impact on GDML processing code as well as on the GDML clients.

3.3.1 Analysis of data types

Simulation environment in high energy physics is a jungle of data models. There are models for geometry, for materials, for physics processes and many others. They are different in each simulation tool and their implementations exist in many programming languages. This is not the ideal situation for for a common exchange format like GDML aims to be. It is not critical from semantics point of view as the structure and behaviour of the data models is rather well understood. The more serious problem seems to be the side of integration, the data binding between two different worlds. On one hand there is the common exchange format in XML and on the other hand there are many different implementations of data models. It is impossible to find a single common solution which fits all of them.

In order to find a starting point for a solution of this puzzle the classification of all the various type systems has been done. Since the goal was to provide quickly a working GDML prototype the analysis focused only on the C++ applications which are being used by LHC experiments in their production environments.

Let’s look at the use-case of reading XML data into an application. In the use-case the application is a simulation tool implemented in C++, GDML processing software is also written in C++ and XML data format is based on GDML schema implemented in W3C XML Schema language.

Looking at the scenario just described, the following type systems have been identified:

XML element types

Defined using XML schema, as shown in the left column of Table 3.1. These types define the set of XML elements to be read from the XML data file. GDML schema defines such types.

Application data types

Defined using the application’s implementation language. In this use-case it is the application native C++ data model to be populated from the XML data in GDML format. There is no guarantee of direct correspondence between XML element types and the application native data types.

C++ element types

The reason to have such type system is due to the motivation point above, saying that the application and GDML processing code should be insulated from any XML API. If such insulation layer is provided the question is: *How are the XML data manipulated if they cannot be accessed directly in order to avoid dependency on XML API?*

Why not to have a mirror image of the XML element types defined in C++? It is not very difficult to define such data types. Their only purpose is to hold the information encoded by XML element types, for an example see the right column of Table 3.1.

Table 3.1 Example of mirroring the GDML types into C++

XML element type	C++ element type

`<xs:element`
`    name="box"`
`    substitutionGroup="Solid">`
` <xs:complexType>`
`  <xs:complexContent>`
`   <xs:extension base="SolidType">`
`    <xs:attribute`
`        name="x" use="required"`
`          type="ExpressionOrIDREFType">`
`    </xs:attribute>`
`    <xs:attribute`
`        name="y" use="required"`
`          type="ExpressionOrIDREFType">`
`    </xs:attribute>`
`    <xs:attribute`
`        name="z" use="required"`
`          type="ExpressionOrIDREFType">`
`    </xs:attribute>`
`   </xs:extension>`
`  </xs:complexContent>`
` </xs:complexType>`
`</xs:element>`
	`#include "Saxana/SAXObject.h"`
	`#include "Schema/SolidType.h"`
	`#include <string>`
	
	`class box : public SAXObject,`
	`             public SolidType`
	`{`
	`public:`
	`  box();`
	`  virtual ~box();`
	`  virtual SAXObject::Type type()`
	`  {`
	`    return SAXObject::element;`
	`  }`
	`  const std::string& get_x() const;`
	`  const std::string& get_y() const;`
	`  const std::string& get_z() const;`
	`  void set_x(const std::string& x);`
	`  void set_y(const std::string& y);`
	`  void set_z(const std::string& z);`
	
	`private:`
	`  std::string m_x;`
	`  std::string m_y;`
	`  std::string m_z;`
	`};`

3.3.2 Functional decomposition

GDML processing architecture defines a set of software components and their interactions during processing of GDML files. There are four types of components identified in the whole GDML processing chain:

XML Engine

Separation of XML API dependent code from the rest of the GDML and application processing code is done via this component. Only this component sees the real XML data coming from XML API such as SAX/SAX2 or DOM. It can be implemented in terms of SAX or DOM style but exposes its own single interface to the rest of the GDML processing system.

XML schema handlers

Set of components which are activated whenever a new XML data have been loaded by XML engine component. Their role is to construct an instance of a C++ element type corresponding to the freshly loaded XML data. There is usually one schema handler per XML element of a given XML schema but having a single handler for multiple XML element types is possible.

Application subscribers

The actual construction of application specific data objects is performed by these components. They receive as input the instances of the C++ element types and produce application native objects. The name subscriber indicates that these components can register themselves to get notified about any number of C++ element objects they want. The logic behind their implementation is driven by application needs. It's perfectly possible to write only a single subscriber handling on its input all C++ element types.

Numerical Expressions Evaluator

The component handling evaluation of expressions. In addition to that it acts as registry of constants, quantities and expressions including proper interpreting of their physical units following the standard of SI units.

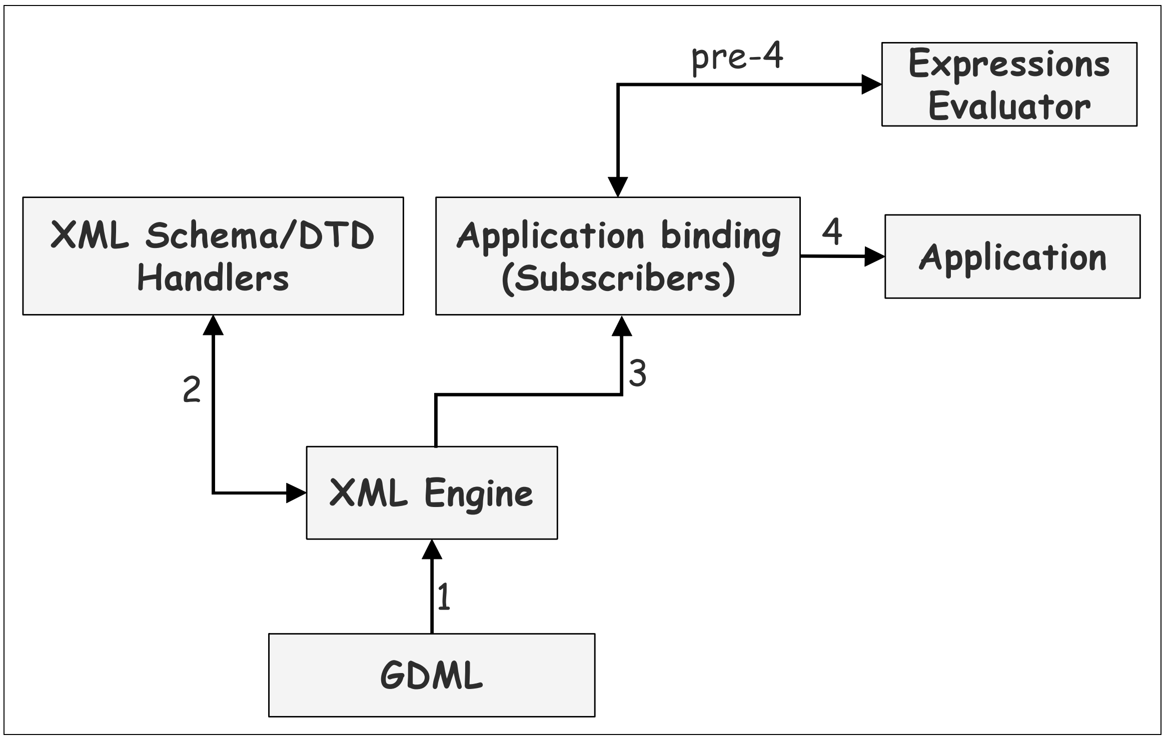


Figure 3.2 Schematic view of GDML processing architecture and its components interactions

The Figure 3.2 shows a simplified collaboration diagram of interactions between GDML processing components. The figure captures the use-case of reading GDML data by a simulation application. As the first step the XML engine starts parsing GDML input by using some XML parser tool. When XML engine observes a complete XML element type it performs a dispatch step which activates a schema handler component corresponding to the type of XML element just loaded. The schema handler produces a new instance of a C++ element type and sends it back to the XML engine. The engine performs a second dispatch during which it checks if there is a subscription pending for a given type of C++ element. If yes, it activates the corresponding subscriber and passes the fresh C++ element object to it. The subscriber extracts the data it needs and if possible it creates application native data object. Before it sends this new native object to application it may ask expressions evaluator component to evaluate any numerical expressions found. The last step is that application receives its new data object as if it was created the standard way.

3.4 Implementation of processing components

The implementation to be described in this section is not the final one. It started as a proof of concept prototype but its implementation has been kept to be production level as much as possible. A new implementation is planned and it will be built using the experience with this prototype. The only way how to test a geometry exchange format is to use it with a real client tool to exercise various scenarios and input data sets. The next sections describe two implementations one which allows to import GDML data in Geant4[10] simulation program and the second which allows to export existing geometry of a running Geant4 application into a user defined output stream or file on disk. Both implementations are going to be included as part of the Geant4 distribution.

3.4.1 GDML Reader

The first thing that needed to be exercised was reading the GDML input data into Geant4 based application. The second important issue was to experiment with insulation layer of XML APIs for GDML processing software. All the components identified in the previous analysis needed to be implemented. The following paragraphs will mention briefly how this was achieved.

3.4.1.1 XML insulation layer

To achieve complete separation of XML API from the rest of the system the XML dependent part has been written in two phases. During the first phase the XML parsing part of engine has been defined and its prototype written based on SAX paradigm, for SAX details see Appendix B.

In the second phase the dispatching part has been developed and the interfaces defined for activation of XML element handlers and Subscribers. The dispatching part of the XML engine implements a state automaton based on theoretical model of hedge regular grammars[43], this will be discussed in detail later in this document.

This part of GDML processing software is the most generic one and is 100% re-usable.

3.4.1.2 C++ element types library

In order to let the XML element handlers perform their tasks the C++ element type library was required as discussed above. The data types defined in this library are kept as simple as possible by intention. The original idea is to generate them directly from a supplied XML schema. For this prototype the hand written approach has been adopted in order to study the feasibility of possible generative approach for the future implementation.

This library is dependent on GDML schema but the approach taken for the hand written data types has gained valuable experience.

The issues and the possible future improvements will be discussed in Future research steps.

3.4.1.3 Processing components

The last phase of GDML Reader prototype consisted of implementation of the XML element handlers and actual application binding components, the Subscribers. In order to make the whole system flexible the plug-in system has been developed first. This system allows to add or remove processing components very easily and keeps internal run-time dependencies at the minimal level. Whole system is initialized in plug and play manner and components are loaded and registered fully automatically at the application start-up.

The first group of components was the XML element handlers. Their implementation is not very complex and as in case of C++ element types it could be generated together with C++ element types library.

This part is GDML schema dependent because requires knowledge about C++ element types internals.

The actual application binding, the subscriber components, was the last and the smallest part of the whole GDML reader implementation. It was possible because all the hard work is done by the rest of the system and the only job of a subscriber component is to extract data values from the passed in C++ element type object and use the data as proper arguments into the application native data types' constructors. The number of subscribers was kept low because some of the subscribers process multiple C++ element types.

This part of the system is very hard to implement using a generative approach because it depends on C++ element types as well as on the application native data model. Some parts of it like subscriber stubs and method skeletons could be generated in principle together with XML element handlers and C++ element types but the actual code invoking constructors of application native data types must written by hand there may be some semantics issues concerning the order of constructors and application settings.

I principle there could be a way of defining a rule based system to achieve the goal but populating such a system would be very impractical and would require a very skilled developer who understands the rules. The goal of this exercises was to minimize exactly this part of code, to reduce the number of lines of code which needs to be updated if some changes happen to the GDML schema.

3.4.2 GDML Writer

The GDML Writer prototype is a very simple tool which enables writing out valid GDML data into a standard C++ stream or file on disk. The goal was to provide a generic writer library as part of the GDML processing software. The little part has been left to the application developer who must write a bit of glue code on the application or framework side.

The separation from XML API was not needed in this case and the problem was simply solved by not using any XML API at all. Actually nothing like that is needed if one just wants to write out XML data. Since no XML API is used the GDML writer library it is pure C++ code with no dependencies on any external tool. It depends of course on the GDML schema as it must guarantee that an attempt to write out invalid GDML data must not be allowed.

The core part of the library is built around a simple data structure inspired by simple application of XML Schema formal model[65]. Once this internal model has been implemented the user API has been defined. The API defines a few database cursors like objects which represent the main GDML domains. The role of each cursor is to guard the validity of GDML data inserted at the current cursor position.

Chapter 4
Gaudi Detector Description versus GDML

The both presented use-cases try to address the storage for detector description data in XML based formats. The solutions will be compared using various criteria and the open issues in both solutions which will be discussed.

4.1 Impact of environment

Gaudi detector description is providing access to XML based detector data for user algorithms in context of the same framework. This makes it simpler because there is only one application data model to be considered which is the transient detector data store of Gaudi framework.

On the other hand the GDML, as common exchange format for geometry data, is addressing heterogeneous environment of simulation tools in high energy physics community where the number of target data models and implementations is virtually infinite.

4.2 DTD versus W3C XML Schema

At the more fundamental level, there is the difference at the schema language used by both solutions. Gaudi detector description has used XML DTD schema language while GDML has used W3C XML Schema to define its XML data structures. Brief comparison of both schema language is shown in Table 1. The following paragraphs will discuss each of the features in more detail.

4.2.1 Syntax

DTD syntax is inherited from old fashioned SGML DTD. Soon after XML has reached developers these started to complain about the fact that DTD syntax is different the that of XML itself. It caused complications for more sophisticated applications trying to use DTD schema information at more advanced level like code generators or database applications.

Table 1 Brief comparison of DTD and W3C XML Schema

DTD	XML Schema
non-XML syntax	XML syntax
no data types	data types supported, attributes inclusive
only structural constraints	structural and data type constraints + regular expression facets of values
single type without sub-typing	subtyping supported by extension or by restriction
poor for data oriented applications	data oriented applications may benefit from (semi) automated solution using XML Schema

XML Schema comes with the syntax which is the same as XML syntax. This allows to use the same tools which are used for XML data and thus process XML Schema in more natural way.

Parsing schema files makes sense for data oriented XML applications because schema holds very useful information about internal structure of the data. This information can be used either at run-time to perform various consistency checking or off-line to generate for example some modules of application data binding or documentation about the data model implemented by a schema.

4.2.2 Data types

DTD was developed by document oriented community. It is sufficient for document oriented XML applications like DocBook[17] or MathML[18]. For data oriented application like Gaudi detector description or GDML it poses certain problems.

DTD is pushing its schema definitions strongly into the syntactic level and the only “data” it allows to define is #PCDATA which is basically a variable text data. For a developer this means that nothing can be done by XML parser validating XML data against a DTD in order to check if the expected data value is a valid integer or float, for example. For the XML parser it will be a perfectly valid value if a #PCDATA was defined at the given place inside DTD. The direct consequence is that developer must introduce a lot of additional code to check if the string values read in by XML parser are really what a client code expects.

In Gaudi implementation a lot of code had to be written to ensure the proper value semantics at the application level.

With XML Schema this problem is solved as XML Schema can express base types[68] and complex data types[67]. So putting a floating point number in place of an integer will generate an error during run of XML Schema validating parser and thus programmer is freed from burden to implement additional test in his/her code.

In GDML case using XML Schema allowed to express more semantics constraint at GDML schema level including constraints like proper units in case of some types for physical quantities and properly typed default values without a need to of additional C++ code.

4.2.3 Sub-typing

Sub-typing relation or in other words ability to specialize one type by extending it or narrowing its value domain is called in object-oriented community inheritance.

DTD does not support something like that which forced complex expressions inside Gaudi detector description DTD which are hard to maintain.

The detector element customising issue in Gaudi needs to be treated explicitly in combination with additional application code. Because of missing inheritance in DTD it was required to emulate this behaviour.

XML Schema allows to define types by extension which roughly corresponds to inheritance in imperative object oriented languages. Together with this mechanism one can use type substitution very much like one does in object-oriented language. Sub-typing by restriction is a but special technique which allows to constrain the value domain of the derived types. This applies to some extent to types and cardinality of attributes and content model definitions.

In GDML it allowed to simplify many definitions and define type hierarchies in a compatible way with the C++ language. This was found as advantage during development of the C++ element types library because the associations defined in GDML schema could be translated easily into C++ code.

Further it also encouraged the idea of automatic generation of C++ elements data model from GDML schema because of the high degree of similarities between XML Schema and C++. In principle any object-oriented language more or less matches the XML Schema type system.

GDML schema can be easily extended without intrusive impact on the rest of the system as the old client will work as before.

4.3 Maintenance

Implementation of Gaudi XML converters suffered from the maintenance point of view due to the following reasons:

Direct use of SAX API

There is nothing bad on using SAX API for fast XML processing with low memory footprint. The problems arise when changes to DTD had to be done which implied updates on the implementation of XML converters. Changes like renaming of element tags, adding or removing element attributes were the reasonably simple ones.The real complications were caused by re-arranging element content models which affected the way XML converters based on SAX where assembled together. In other words the problem was that for some content models the XML SAX based converters were not stateless. Their state was dependent on the type of child element. Whenever for some reason the content model has been changed it triggered often set of massive changes to the code of XML converters.

Redundant code

Insufficient separation of the Gaudi code from the XML API caused a lot of redundant code across the whole implementation. In case some changes were needed a lot of code needed to be updated.

Complex control flow

SAX is so called push based API which means that SAX XML parser is in charge of program control during whole parsing period. The only moments when application code is activated are the implementations of SAX callbacks. In order to keep track of what the actual processing context is the XML converters needed to maintain their own state which consisted of the special stack and a set of control variables which required a special handling in cases where XML converter has been called recursively. This was required in a few places where it was too complicated to pass control back to SAX parser. One of these cases was remote link resolution task which created a new SAX parser instance reading the content of the remote link.

Efficiency

SAX is fast because it parses file in one go and no extensive memory allocation is involved during that process. The problem is if random access is needed to XML elements in a XML file. At the moment SAX based solution is not fast any more because it needs to parse the file again and again from the beginning to the point the data are found. This could happen easily in Gaudi as the Gaudi detector data store behaves like a random access storage device and triggers activation of converters any time a data object is required from the persistent storage.

Looking at maintenance issues in GDML using the same set of points one can see the following:

XML API well separated from the rest of the system

The proof of this is that switching parsing engine in the XML engine from SAX to SAX2 required 20 lines change in the XML dependent part and the rest of the system is working without noticing that.

MInimal code redundancy

The way GDML processing software is designed eliminated a lot source of code redundancy. There is some code redundancy inside the C++ element type library and XML element handlers but this code is simple and it was not worth to re-factorise.

Control flow well defined

GDML processing architecture defines control flow in the single place. Unlike in Gaudi XML converters this mistake was avoided by precise design of component interactions during each phase of GDML processing. The clean and simple control flow has been achieved by the implementation based on stack push down state machine. This state machine has very few well defined steps and simple book-keeping. The main advantage of this implementation is that is not affected at all by changes to GDML schema. This XML engine works the same way for any XML Schema.

Comparing GDML to Gaudi implementation from the efficiency point of view is not completely possible as GDML now supports only single input file and no remote object linking. It is at least efficient as Gaudi XML SAX converters are when processing a single XML data file.

Chapter 5
Connections to Theory

At this point we have gone through the complete descriptions of the presented use-cases including their comparison using various criteria. During the course of explaining the design choices and implementation decisions some forward references have been made indicating issues related to some theoretical models.

GDML solutions seem to provide better answers to the set of identified problems than the solution described of Gaudi detector description framework. Despite the fact that GDML has been built with more experience in hand there is still some space for improvements.

Recalling the discussion about implementation of the GDML processing architecture the careful reader can guess that two parts of the whole GDML processing implementation which were explicitly mentioned as hand written. The guilty parts are C++ element library and XML element handlers. In both cases they occupy a considerable amount of C++ code, which means a maintenance commitment in case of any update to GDML schema.

The second thing the careful reader may recall is the original intention of keeping the code of C++ element library type definitions and XML element handlers as simple as possible due to a possibility of generating this part of GDML processing code out of GDML schema.

There are reasons for the generative approach which have not been discussed so far. The logical connection between the generative approach and keeping the code simple is why the XML dispatching engine is implemented in a particular way. Very brief remark has been made, saying that the core part of the XML engine in GDML processing implementation, which is the bridging component between the world of XML and the world of transient C++ objects, is based on the principle of a state automaton.

The state automaton belongs to the family of push down stack automata known from the area of context free grammars[35][36]. Such automata are used to answer the question of the membership problem for a language defined by a context free grammar. However, such a test does not really happen in the implementation of the GDML XML engine component but the state machine itself follows the rules of this theoretical model. The reason for this will be discussed in the following sections which reveal the relevant pieces of theory as the discussion will go through the topic.

5.1 XML, trees and hedges

Let's consider the following XML snippet shown in Listing 5.1.

Listing 5.1 Simple XML snippet before being transformed

```
<a>
  <b>text</b>
  <c></c>
  another text
</a>
```

Now let's forget about the end tags in form </x> and let's draw the connector lines from enclosing (parent) elements to embedded (children) elements. The result may look like the one shown in Figure 5.1.

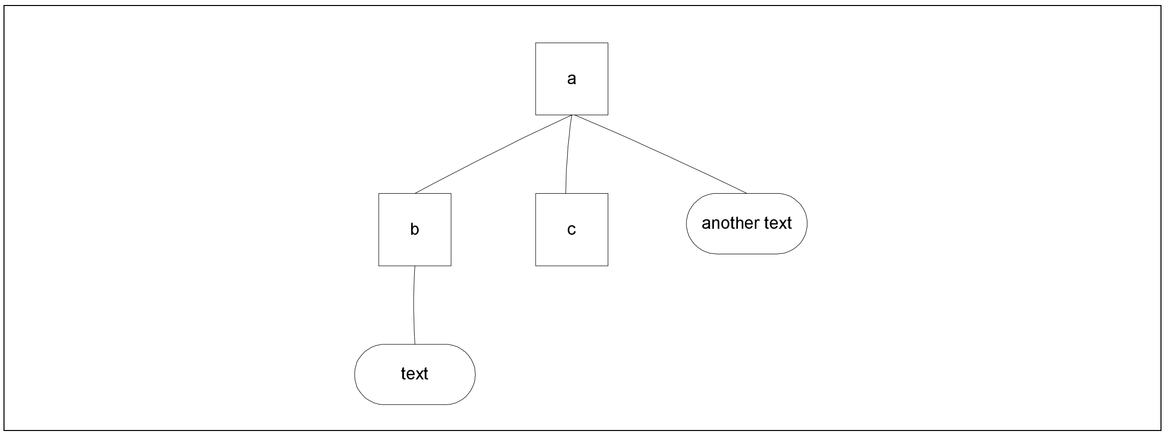


Figure 5.1 XML snippet after transformation

Yes, it's a tree. Note that the nodes containing just the text are leaves only. Nobody in XML community will ever consider something else than trees as the native representation for XML. Now, if we perform a virtual zoom into the top level element <a> we get the following picture shown in Figure 5.2. The tree is gone but we got a sequence of elements. If we attach a special "empty" symbol ***e*** to the element <c> we get clearly a sequence of trees apart from the text node "another text" which became a free flying leaf as shown in the Figure 5.3.

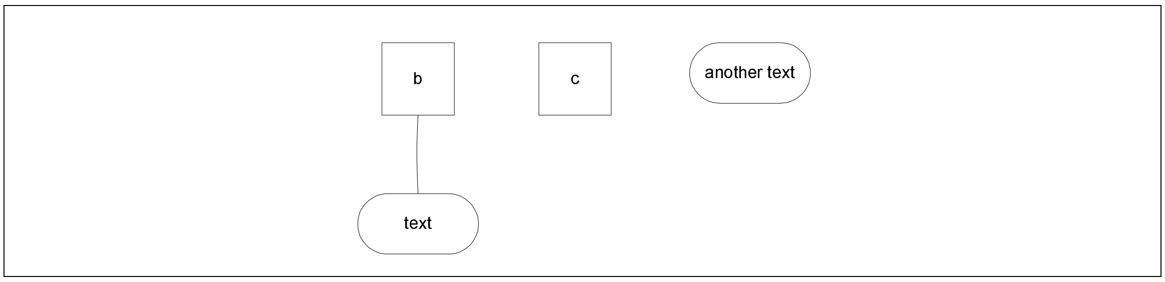


Figure 5.2 Inside the content of the top level element <a>

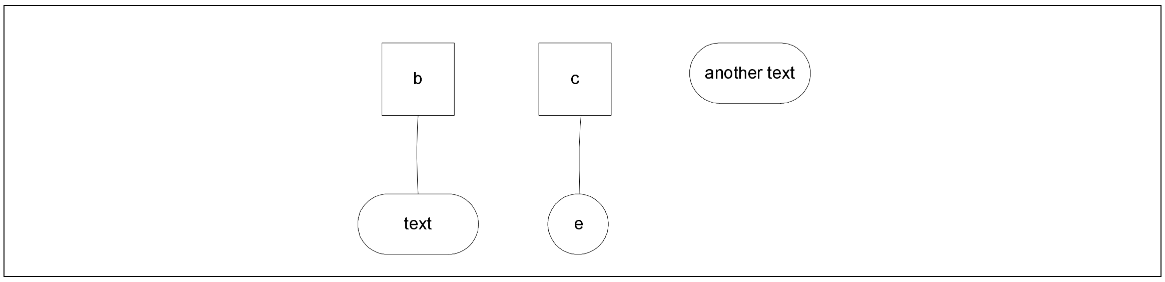


Figure 5.3 The content of <a> a bit decorated

Now the following formal definition from [43]:

Definition: Hedge

A hedge over a finite set Σ (of symbols) and a finite set X (of variables) is:

1. ε (the null hedge),
2. x, where x is a variable in X,
3. a<u>, where a is a symbol in Σ and u is a hedge
4. uv, where u and v are hedges (the concatenation of two hedges).

The elements of Σ (i.e., a and b) are used as labels of non-leaf nodes, while elements of X (i.e., x) are used as labels of leaf nodes. We abbreviate a<ε> as a. Thus, the third example is denoted by a b<b x>.

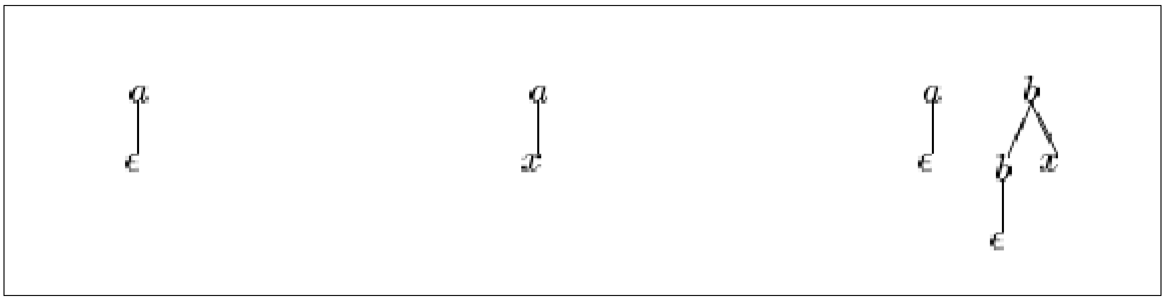


Figure 5.4 Examples of hedges: a<ε>, a<x>, and a<ε> b<b<ε> x>

makes perfect sense. The Figure 5.5 shows informally the definition above. Don't get confused by the word forest in the picture. The term forest applies equally well here, since hedge is a special case of a forest introducing ordering relation on its children, so unlike the term forest which means ***set of trees*** the term hedge means ***sequence of trees*** which describes the XML data more precisely especially if we consider the notion of validity of XML documents w.r.t. a DTD or W3C XML Schema where order of child elements matters[1].

1. We apologize for less formal treatment of the theoretical topics in this section, but interested reader is encouraged to follow references for the complete formal treatment of hedge automata in [43].

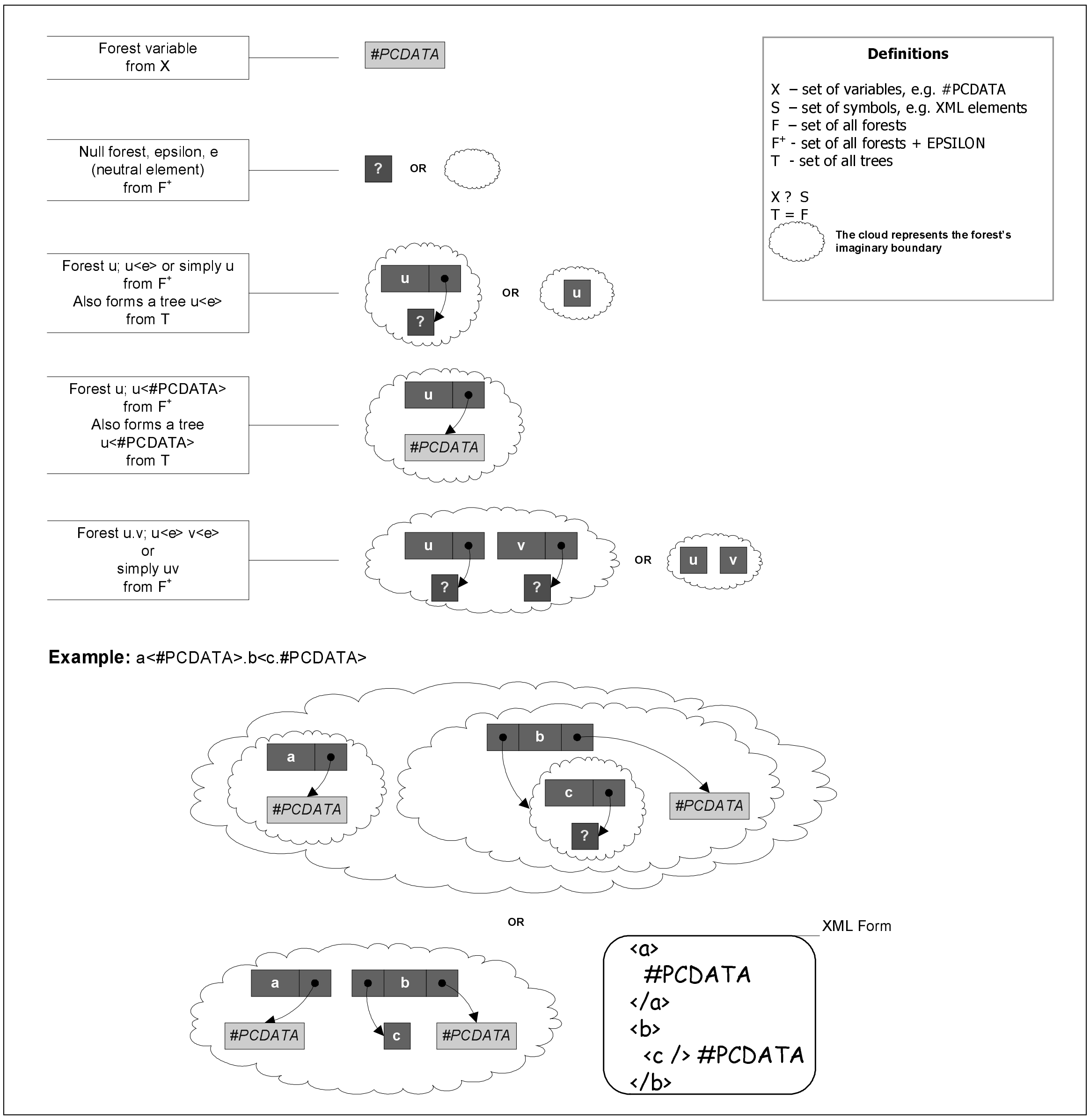

Forest variable from X
#PCDATA
Null forest, epsilon, e (neutral element) from F+
?
OR
Forest u; u<e> or simply u from F+ Also forms a tree u<e> from T
u
?
OR
u
Forest u; u<#PCDATA> from F+ Also forms a tree u<#PCDATA> from T
u
#PCDATA
Forest u.v; u<e> v<e> or simply uv from F+
u
v
?
?
OR
u
v
Definitions
X – set of variables, e.g. #PCDATA
S – set of symbols, e.g. XML elements
F – set of all forests
F+ - set of all forests + EPSILON
T - set of all trees
X ? S
T = F
The cloud represents the forest's imaginary boundary
Example: a<#PCDATA>.b<c.#PCDATA>
a
#PCDATA
b
c
?
#PCDATA
OR
a
b
#PCDATA
c
#PCDATA
XML Form
<a>
#PCDATA
</a>
<b>
<c /> #PCDATA
</b>

Figure 5.5 Forests or hedges seen as natural formalism for XML

5.2 Hedge automata and XML SAX API

Now let's recall again the way two SAX XML API works. In SAX style of XML parsing the parser travels through the XML document tree in depth first search manner. This behaviour corresponds to a run of bottom-up hedge automaton. In other words, during this traverse, first all children elements are visited then their parent element. This behaviour is exploited by XML engine of GDML processor in order to perform proper dispatching and activation of XML element handlers and subscribers.

During initial phase of implementation of XML engine the testing GDML schema was defined using DTD schema language. At that time no problems were observed and the dispatching worked well.

When GDML schema based on W3C XML Schema has been finished the C++ element library was updated and first run have shown some problems during dispatching where either wrong subscriber has been activated or none at all. After having a deeper look at the way the dispatcher was working the problem has been identified.

The problem happend due to the fact that DTD and XML Schema describe different classes of regular tree grammars [44]. While DTD does allow only deterministic content models for child elements the W3C XML schema defines a bit less restricted class of regular tree grammars, so called ***single type tree*** grammars according to [44].

The consequence is the following. In case of XML Schema there is a little non-determinism introduced. Thanks to that non-determinism one is allowed to define the schema to allow the following XML fragment which is not possible with DTD:

Listing 6

```
<structure>
 <assembly>
  <child><volumeref ref=”blabla”/></child>
 </assembly>
 <volume>
  <child><volumeref ref=”blablablabla”/><positionref ref=”one”/></child>
 </volume>
</structure>
```

The XML snippet shows two so called ***competing elements*** because of the <child> tag in their content model. The confusion happened when the <child>’s C++ element object has been successfully created and XML engine tried to locate a subscriber ready to consume the object. What happened was that there were two subscribers waiting to handle the <child> type object, e.g. the `assemblySubscriber` and `volumeSubscriber` components. XML dispatch step went wrong way, because it was ambiguous to choose the proper one of the two just knowing that the type of object in hand is <child>. In order to make the right choice one needs to know the parent’s element type.

That implementation of XML engine had simple bookkeeping based on single stack to keep track of the current context and active XML element handler. The subscriber look-up happened when end of the current element has been found.

The current implementation is using two stacks which should be sufficient for XML Schema class of regular tree grammars. This second stack is so called ***parent stack***. This stack keeps track of parent elements and uses this information to resolve ambiguous situations. In addition to the previous simpler dispatching the new implementation of XML engine also performs so called parent notifications about all the children observed inside the content model. This way it is easier to implement some cases and avoid ambiguous dispatching steps.

Chapter 6
Related work

The related work can be split into two groups. The first one deals with the detector description efforts to use XML technology for detector description. The second area is the world of transport, exchange and sharing formats where GDML project aims to belong, at least in scope of high energy physics applications.

6.1 Detector description domain

This section will discuss efforts and solutions competing with Gaudi detector description framework.

6.1.1 ATLAS AGDD

Very soon after XML has been deployed in LHCb experiment for purposes of detector description, the group of developers in ATLAS[4] collaboration launched AGDD XML project [5]. This project had very similar goals as Gaudi detector description framework.

The main difference was that ATLAS was just finishing at that time their common framework and not so many developers in ATLAS understood the new framework internals in order to be able to integrate their new developments. This affected AGDD initial design choices and subsequent implementations they provided.

The AGDD DTD focused more on advanced geometry patterns than a robust detector description foundation. Generic Object Model framework has been developed to enable import/export of AGGD XML descriptions into ATLAS applications. They had made a design mistake by implementing a tree like memory representation for AGDD data and they used DOM API to read XML data it into application's memory. The straight effect of that was high data redundancy due to the fact that at the same time the DOM and AGDD data structures occupied a lot of memory before any application data object has been created.

When AGDD developers realized the mistake they developed sophisticated system to allow writing so called compact AGDD XML data. This solution helped them to suppress memory footprint problems but it had impact on XML syntax and required additional user defined C++ code which was needed to properly expand the compact AGDD representation read from XML data file.

The lack of common framework support in their solution caused that many independent applications have been developed with low level of inter-operability. They could not handle distributed set of XML data files with remote object linking. Their solution was more towards using high level XML tools like XSL style sheets to prepare AGDD input data before an application could be run.

The future of this project is uncertain as today the production solution in ATLAS experiment is based on RDBMS system.

6.1.2 CMS DDD

The use of XML for detector description in CMS[6] experiment had started much later after the first deployment of the solution in Gaudi framework. The goals of this project, called DDD[7], were very much like the ones of Gaudi detector description framework. Their implementation enjoyed the support of the CMS common framework and they achieved the same goal as in Gaudi. The DDD had become the single source of detector description data for all CMS applications which required it.

Their processing model was based on tree like transient data model with focus on built in configuration management system. Their implementation did not support directly the remote object linking and loading on demand of detector description data. The effective implementation on their side has been achieved by preparing the input stream of detector description data by filtering their whole database according to the current configuration tag.

6.1.3 GLAST

The GLAST detector description solution is somewhere between Gaudi and ATLAS solutions. Their XML data format is inspired by ATLAS AGDD DTD and XML data processing part follows very much the ATLAS philosophy. Unlike the ATLAS AGDD, this project is still being actively developed.

6.2 Data exchange domain

GDML project is a bit outstanding because there is no other XML format in HEP community with the ambition to become the common exchange format for detector geometry. The competitors to GDML are the XML formats described above including the XML format of Gaudi detector description. But these formats rely on the common framework infrastructure and target usually only a single transient data model.

When looking for a related XML language with a similar mission like GDML, there two mark-up languages which are somehow related. The X3D[19], virtual reality language for the WWW and the GML[20], the Geography Markup Language used for transfer, exchange and storage of geographical data. In both cases, the evaluation studies have been done before GDML development has started, but conclusion of developers in HEP community was that these languages are not of relevant use.

Chapter 7
Summary

Looking back at the list of principal goals, I believe that all of them have been achieved at a good level of satisfaction. In my case it means that the solutions I have provided are still being used in the production environments.

The transient data model of Gaudi detector description has been designed and proven to work in the complex computing environment of LHCb experiment. This object oriented data model was not only fulfilling the requirements collected at the early stages of its analysis and design, but along the way even set of optimisations was implemented and well aligned with philosophy of Gaudi framework. The data model protects the optimal use of computing resources because it does not allow that a redundant data are loaded or data which are not needed at all.

The outstanding solution on the persistency side of detector description data has been provided as the first solution of this kind in high energy physics environment. The XML based persistent detector data in Gaudi framework was and is still the only one supporting transparent inter object navigation for detector data stored across distributed set of files.

There are some aspects of XML data processing in Gaudi detector description framework which were not resolved in the most optimal way. I believe that these mistakes have been avoided in the design of GDML solution which aims to become the common geometry exchange data format in the high energy physics computing environemt. The ambitions of this project are not only at the modular XML format design but as well on the software support side with the goal of providing very optimal solution for easy integration into experimental frameworks or simulation tools.

The GDML processing architecture has been carefully designed building on the experience gained from other projects in that domain. The numbers might tell a bit more about the GDML processing implementation. For the Geant4 GDML reader the total number of lines in C++ for the whole implementation is almost 16000. The number of lines of code required to integrate the GDML reader into Geant4 was roughly 2500 lines of C++ which is around 15% of total code base used for the whole prototype. On the side of GDML Writer the total number of lines in C++ is 3300 and to integrate it into Geant4 system required less then 300 lines of C++ which is around 9%. This clearly shows that the requirement of low cost application data binding in terms of glue code has been fulfilled.

The XML engine component of the GDML processing is written with care and its generic implementation, going closely along the theoretical models for XML, can be used for more general processing tasks.

References

1 European Laboratory for Particle Physics (CERN)
http://www.cern.ch

2 The Large Hadron Collider project ate CERN,
http://www.cern.ch/LHC

3 LHCb collaboration,
http://www.cern.ch/lhcb

4 ATLAS collaboration,
http://www.cern.ch/atlas

5 ATLAS AGDD in XML
http://www.nikhef.nl/~stanb/AGDD/AGDD.html

6 CMS collaboration
http://cmsinfo.cern.ch/Welcome.html

7 CMS Detector Description Database
http://cmsdoc.cern.ch/cms/software/ddd/www

8 GLAST Detector Description
http://www-glast.slac.stanford.edu/software/detector_description

9 GRID at CERN
http://www.cern.ch/grid

10 GEANT4, Object Oriented Simulation Toolkit
http://www.cern.ch/geant4

11 Gaudi Framework Project
http://www.cern.ch/Gaudi

12 Objectivity DB
http://objectivity.com

13 ROOT
http://root.cern.ch

14 World Wide Web Consortium
http://www.w3c.org

15 Apache XML project
http://xml.apache.org

16 Unicode
http://www.unicode.org

17 DocBook
http://www.docbook.org

18 MathML
http://www.w3.org/Math

19 X3D
http://www.web3d.org

20 Geography Markup Language (GML)
http://opengis.net/gml

21 LHCb Collaboration, *LHCb Technical Proposal*, European Laboratory for Particle Physics (CERN), CH-1211, Geneve 23 - Suisse, ISBN 92-9083-123-5, 1998

22 P. Mato and LHCb software architecture group, *Gaudi - Architecture Design Document*, LHCb experiment, European Laboratory for Particle Physics (CERN), CH-1211, Geneve 23 - Suisse, LHCb/98-064 COMP

23 P. Maley and LHCb software architecture group, *Gaudi User Guide*, LHCb Experiment, European Laboratory for Particle Physics (CERN), CH-1211, Geneve 23 - Suisse, 1999

24 LHCb Detector Description DTD
http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/DetDesc/Documents/lhcbdtd.pdf

25 P. Binko, *Object Oriented Databases in High Energy Physics*, in Proceedings of CERN SCHOOL OF COMPUTING 1997, ISBN 92-9083-120-5

26 P. Binko et al., *LHCb Computing Tasks*, LHCb experiment, European Laboratory for Particle Physics (CERN), CH-1211, Geneve 23 - Suisse, LHCb/98-042 COMP, Phys. Rev. D Volume 50, Issue 3, August 1, 1994

27 R. Chytracek, GAUDI Detector Data Model,
http://lhcb-comp.web.cern.ch/lhcb-comp/Frameworks/DetDesc/Documents/GaudiDDDB.pdf

28 Geometry Description Markup Language
http://www.cern.ch/gdml

29 Radovan Chytracek, "The Geometry Description Markup Language", In proceedings of CHEP 2001, Pages 473-476, Beijing, China

30 G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, G. Corti, M. Frank, G. Gracia, J. Harvey, E. van Herwijnen, B. Jost, I. Last, P. Maley, P. Mato, S. Probst and F. Ranjard, A. Tsaregorodtsev, "GAUDI - The Software Architecture and Framework for building LHCb Data Processing Applications", CHEP2000 proceedings, Padova, Feb. 2000.

31 G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, G. Corti, M. Frank, G. Gracia, J. Harvey, E. van Herwijnen, B. Jost, I. Last, P. Maley, P. Mato, S. Probst and F. Ranjard, A. Tsaregorodtsev, "Data Persistency Solution for LHCb", CHEP2000 proceedings, Padova, Feb. 2000.

32 G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, G. Corti, M. Frank, G. Gracia, J. Harvey, E. van Herwijnen, B. Jost, I. Last, P. Maley, P. Mato, S. Probst and F. Ranjard, A. Tsaregorodtsev, "The LHCb Detector Description Framework", CHEP2000 proceedings, Padova, Feb. 2000.

33 G. Barrand, I. Belyaev, P. Binko, M. Cattaneo, R. Chytracek, G. Corti, M. Frank, G. Gracia, J. Harvey, E. van Herwijnen, P. Maley, P. Mato, S. Probst and F. Ranjard, “GAUDI -- A software architecture and framework for building HEP data processing applications”, Computer Physics Communications, Volume 140, Issues 1-2, 15 October 2001, Pages 45-55. (http://www.sciencedirect.com/science/article/B6TJ5-4435B2B-7/1/250e27db65134bcd4bbb2065534b4ecd)

34 A. Ballaminut, C. Colonello, M. Dönszelmann, E. van Herwijnen, D. Köper, J. Korhonen, M. Litmaath, J. Perl, A. Theodorou, D. Whiteson and E. Wolff, “WIRED -- World Wide Web interactive remote event display”, Computer Physics Communications, Volume 140, Issues 1-2, 15 October 2001, Pages 266-273. (http://www.sciencedirect.com/science/article/B6TJ5-4435B2B-14/1/343fe541c0dda3c8e065cbecb4b19790)

35 A.V. Aho, R. Sethi, J.D. Ullman, “Compilers - Principles, Techniques and Tools“, Addison-Wesley, 1986

36 J.E. Hopcroft, R. Motwani, J.D. Ullman, “Introduction to Automata Theory, Languages, and Computation”, 2-nd ed., Addison-Wesley, 2001

37 H. Comon and M. Dauchet and R. Gilleron and F. Jacquemard and D. Lugiez and S. Tison and M. Tommasi, “Tree Automata Techniques and Applications“, 1997, http://www.grappa.univ-lille3.fr/tata

38 R. Behrens, “A Grammar Based Model for XML Schema Integration”,
in: Lings,B. et al (Eds.): Advances in Databases, 17th British National Conference on Databases, BNCOD 17 (London,UK,3.-5. July), Springer-Verlag 2000, LNCS, Vol. 1832, S. 172-190.

39 R. Behrens, “On the Complexity of Standard and Specialized DTD Parsing”,
12. Workshop “Grundlagen von Datenbanken”, Plön, 13.-15. June 2000

40 D. Beech, A. Malthora, M. Rys, “A Formal Data Model and Algebra for XML“, W3C XML Query working group note, September 1999

41 M. Fernandez, J. Simeon, P. Wadler, “A Data Model and Algebra for XML Query”, AT&T Research Labs Technical Report, Unpublished manuscript, 2000 http://www.research.att.com/~mff/files/algebra1.ps

42 A. Brown, M. Fuchs, J. Robie, and P. Wadler., “MSL. A model for W3C XML Schema”, In 10th Int'l World Wide Web Conf., Hong Kong, May 2001

43 M. Murata, “Hedge Automata: a Formal Model for {XML} Schemata”, 2000, http://citeseer.ist.psu.edu/murata99hedge.html

44 M. Murata, D. Lee, and M. Mani, “Taxonomy of XML Schema Languages using Formal Language Theory”, In Extreme Markup Languages, Montreal, Canada, Aug. 2001, http://www.cs.ucla.edu/dongwon/paper/

45 B. Stroustrup, “The C++ Programming Language - Special Edition”, Addison-Wesley, 2000

46 M.H. Austern, “Generic Programming and the STL”, Aiddison-Wesley, 1999

47 C. Szyperski, “Component Software”, Addison-Wesley, 1998

48 K. Czarnecki, U. W. Eisenecker, “Generative Programming“, Addison-Wesley, 2000

49 Cleaveland, “Program generators in Java and XML”, Prentice Hall, 2001

50 Simple API for XML (SAX)
http://sax.sourceforge.net

51 Document Object Model,
http://www.w3c.org/DOM

52 Apache XML Project, Xercec-C DOM Proramming Guide
http://xml.apache.org/xerces-c/program-dom.html

53 L. Dodds, “Parsing the Atom”, Xml.com, April 25 2001
http://www.xml.com/pub/a/2001/04/25/deviant.html

54 L. Dodds, “Painting by Numbers with SVG”, March 15 2000
http://www.xml.com/pub/a/2000/03/15/deviant/index.html

55 Lee, D., Mani, M., Chiu, F., Chu, W. W., “Nesting-based Relational-to-XML Schema Translation”. In: Int'l Workshop on the Web and Databases (WebDB). Santa Barbara, CA., May 2001.

56 Oracle XML-SQL Utility,
http://otn.oracle.com/tech/xml/oracle_xsu

57 IBM DB2 XML Extender,
http://www.ibm.com/software/data/db2/extenders/xmlext

58 THE BREEZE XML BINDER,
http://www.breezefactor.com

59 International Organization for Standardization, Geneva, Switzerland, “ISO 8879: Information Processing - Text and Office Systems - Standard Generalized Markup Language (SGML), 1986

60 T. Bray, J. Paoli, and C. M. Spreberg-McQueen (Eds.), “Extensible Markup Language (XML) Version 1.0”, 2nd Edition, Oct. 2000,
http://www.w3.org/TR/2000/REC-xml

61 J. Clark and S. DeRose (Eds.), “XML Path Language (XPath) Version 1.0”, Nov. 1999,
http://www.w3.org/TR/xpath

62 J. Clark (Eds.), “XML Transformations” XSLT Version 1.0”, Nov. 1999,
http://www.w3.org/TR/xslt

63 S. DeRose, E. Maler, and D. Orchard (Eds.), “XML Linking Language (XLink) Version 1.0”, June 2001,
http://www.w3.org/TR/xlink

64 S. DeRose, E. Maler, and R. Daniel (Eds.), “XML Pointer Language (XPointer) Version 1.0 - W3C Candidate Recommendation”, September 2001,
http://www.w3.org/TR/xptr

65 A. Brown, M. Fuchs, J. Robie, P. Wadler, “XML Schema: Formal Description”
http://www.w3.org/TR/xmlschema-formal/

66 D.C. Fallside, “XML Schema Part 0: Primer”, May 2001,
http://www.w3.org/TR/xmlschema-0

67 H.S. Thompson, D. Beech, M. Maloney, N. Mendelsohn, “XML Schema Part 1: Structures”, May 2001,
http://www.w3.org/TR/xmlschema-1

68 P.V. Biron, A. Malhotra, “XML Schema Part-2: Datatypes”, May 2001,
http://www.w3.org/TR/xmlschema-2

69 W3C Cascading Style Sheets
http://www.w3.org/Style/CSS

Appendix A
C++ to XML mapping

The couple examples shows how the C++ classes can be mapped into XML elements. The example demonstrations are done according to the mapping rules shown in Table 2.1.

A.1 Class

Table A.1 Class to XML element mapping

C++	DTD	XML example
`class A {...};`	`<!ELEMENT A (...)>`	`<A>...</A>`

A.2 Class data members of base types

Table A.2 Class data members of base type mapped into XML

C++	DTD	XML example

`class B {`
`...`
`private:`
`  int    m_i;`
`  double m_d;`
`};`
	`<!ELEMENT B(...)>`
	`<!ATTLIST B`
	`  i CDATA #REQUIRED`
	`  d CDATA #REQUIRED`
	`>`
		`<B i=”0” d=”3.14”>`
		`...`
		`</B>`

A.3 Class data members of a complex type

Table A.3 Class data members of a complex type mapped into XML

C++	DTD	XML example
class A {	<!ELEMENT A (...)>	<B i="0">
};	<!ELEMENT B (A)>	<A>
class B {	<!ATTLIST B	...
...	i CDATA #REQUIRED	</A>
private:	>	</B>
A m_a;		
int m_i;		
};		

A.4 Class data members of a container type

Table A.4 Class data members of a container type mapped into XML

C++	DTD	XML example
class A {	<!ELEMENT A (...)>	<B i="0">
};	<!ELEMENT B (A+)>	<A>...</A>
class B {	<!ATTLIST B	<A>...</A>
...	i CDATA #REQUIRED	<A>...</A>
private:	>	<A>...</A>
vector<A> m_a;		</B>
int m_i;		
};		

A.5 Class references

Table A.5 Class references mapped into XML

C++	DTD	XML example
`class A {`	`<!ELEMENT A (...)>`	`<B aref=”idA” i=”0”>`
`};`	`<!ATTLIST A`	`  ...`
`class B {`	`  name ID #REQUIRED`	`</B>`
`...`	`>`	`<A name=”idA”>`
`private:`	`<!ELEMENT B (...)>`	`</A>`
`  A*  m_a;`	`<!ATTLIST B`	
`  int m_i;`	`  aref IDREF #REQUIRED`	
`};`	`  i    CDATA #REQUIRED`	
	`>`	

Appendix B
Application Programming Interfaces for XML

B.1 SAX - Simple API for XML[50]

SAX is event based XML API. For each XML fragment of a given type it generates a correspondning type of event and pushes this to the application via its `DocumentHandler` interface. Listing B.1 shows example of SAX2 interface in GDML XML parsing engine. This interface has to be implemented by the client application and registered with the instance of the SAX parser used to parse XML data. The process of generating event is shown in Table B.1:

Table B.1 The process of generating SAX events

XML data	SAX events
	`startDocument()`
`<a>`	`startElement(a)`
`<b battr="val"/>`	`startElement(b,[battr,"val"])`
	`endElement(b)`
`<?instr, data?>`	`processingInstruction(instr,"data")`
`<c>text</c>`	`startElement(c)`
	`characters("text")`
	`endElement(c)`
`</a>`	`endElement(a)`
	`endDocument()`

The table shows only the basic events. There are more types of events depending on the version of SAX API. There are SAX and SAX2 APIs defined. The SAX2 API provides more types of events including XML declaration, comments and others. SAX2 has been created when XML developers started to use SAX based applications for more sofisticated XML processing which required more complete informaton about the XML document being processed. With the arrival of XML Schema and XML namespaces SAX API was not sufficient. SAX2 supports them well.

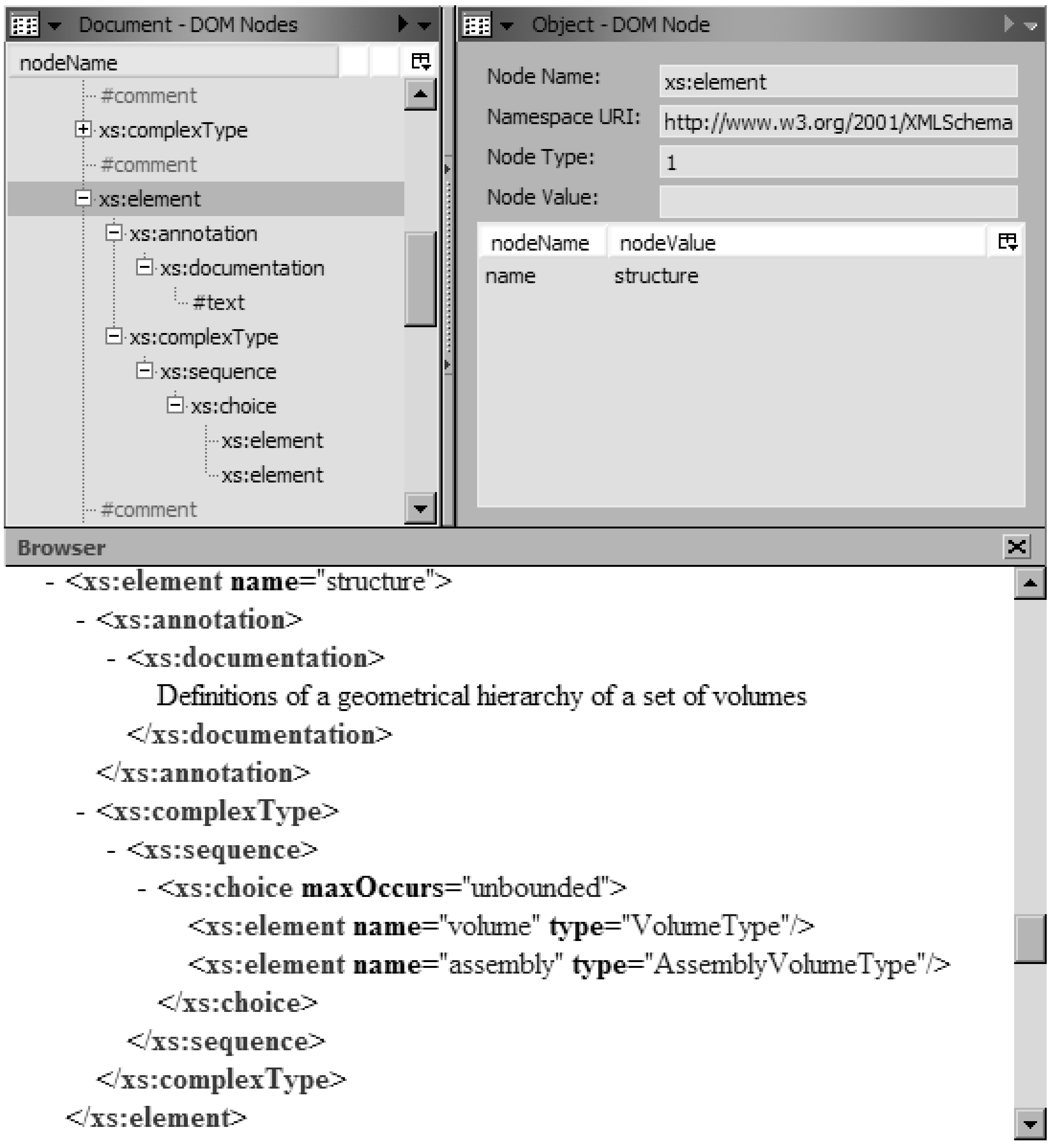
Listing B.1 DocumentHandler, the base SAX interface

```
#include <xercesc/sax2/DefaultHandler.hpp>
class SAX2EventGun : public xercesc::DefaultHandler {
  public:
    void characters( const XMLCh* const chars,
                     const unsigned int    length );
    void endDocument();
    void endElement( const XMLCh* const uri,
                     const XMLCh* const localname,
                     const XMLCh* const qname );
    void ignorableWhitespace( const XMLCh* const  chars,
                              const unsigned int  length );
    void processingInstruction( const XMLCh* const  target,
                                const XMLCh* const  data );
    void resetDocument();
    void setDocumentLocator( const Locator* const locator );
    void startDocument();
    void startElement( const XMLCh* const uri,
                       const XMLCh* const localname,
                       const XMLCh* const qname,
                       const Attributes&  attributes  );
    InputSource* resolveEntity( const XMLCh* const publicId,
                                const XMLCh* const systemId );
    void error(const SAXParseException& exception);
    void fatalError(const SAXParseException& exception);
    void warning(const SAXParseException& exception);
    void resetErrors();
    void notationDecl( const XMLCh* const name,
    const XMLCh* const publicId,
    const XMLCh* const systemId );
    void resetDocType();
    void unparsedEntityDecl( const XMLCh* const name,
                             const XMLCh* const publicId,
                             const XMLCh* const systemId,
                             const XMLCh* const notationName );
};
```

In addition, SAX ca be used to build so called XML filters which allow more sofisticated XML processing from software engineering point of view. The idea is to build a chain of objects which implement the SAX `DocumentHandler` interfaces and each of them is dedicated to process a given tag or subset of XMl elements. The filters are composed in producent/consument manner. Using this paradigm similar to Unix shell pipes one can combine them in the best way to fit the applications needs.

B.2 DOM - Document Object Model

DOM defines the W3C standard[51] programming interface for XML processing. Unlike SAX, this programmng interface loads whole XM document into memory and allows its manipulation via exposed set of DOM functions. Figure B.1shows example of DOM structur ein memory of GDML Schema structure element. DOM data consist of set of nodes for each XML element in the document. There are several types of nodes for each type of XML elements. DOM provides a convenient way of manipulating XML data including fetch by tag name and document tree traversal. The latest DOM specification DOM Level 3 is addressing the capability to load and save a XML document to support XML data serialization.



Node Name:	xs:element
Namespace URI:	http://www.w3.org/2001/XMLSchema
Node Type:	1
Node Value:	

nodeName	nodeValue
name	structure

```
- <xs:element name="structure">
  - <xs:annotation>
    - <xs:documentation>
        Definitions of a geometrical hierarchy of a set of volumes
      </xs:documentation>
    </xs:annotation>
  - <xs:complexType>
    - <xs:sequence>
      - <xs:choice maxOccurs="unbounded">
          <xs:element name="volume" type="VolumeType"/>
          <xs:element name="assembly" type="AssemblyVolumeType"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Figure B.1 Example of DOM model in memory

Listing B.2 Example of creating a DOM document[52]

```
//
//  Create a small document tree
//
{
  XMLCh* tempStr[100];
  XMLString::transcode("Range", tempStr, 99);
  DOMImplementation*
  impl = DOMImplementationRegistry::getDOMImplementation(tempStr, 0);

  XMLString::transcode("root", tempStr, 99);
  DOMDocument*   doc = impl->createDocument(0, tempStr, 0);
  DOMElement*   root = doc->getDocumentElement();

  XMLString::transcode("FirstElement", tempStr, 99);
  DOMElement*   e1 = doc->createElement(tempStr);
  root->appendChild(e1);

  XMLString::transcode("SecondElement", tempStr, 99);
  DOMElement*   e2 = doc->createElement(tempStr);
  root->appendChild(e2);

  XMLString::transcode("aTextNode", tempStr, 99);
  DOMText*       textNode = doc->createTextNode(tempStr);
  e1->appendChild(textNode);

  // optionally, call release() to release the resource
  // associated with the range after done
  DOMRange* range = doc->createRange();
  range->release();

  // removedElement is an orphaned node, optionally call release()
  // to release associated resource
  DOMElement* removedElement = root->removeChild(e2);
  removedElement->release();

  // no need to release this returned object
  // which is owned by implementation
  XMLString::transcode("*", tempStr, 99);
  DOMNodeList*    nodeList = doc->getElementsByTagName(tempStr);

  // done with the document, must call release()
  // to release the entire document resources
  doc->release();
};
```

The Listing B.2 shows an example of creating and manipualting a DOM document in memory. The lines in bold are the DOM API calls.

Definitions

Architecture	The software architecture of a program or computing system is the structure or structures of the system, which comprises software components, the externally visible properties of those components, and the relationships among them.
Framework	A framework represents a collection of classes that provide a set of services for a particular domain; a framework exports a number of individual classes and mechanisms that clients can use or adapt. A framework realises an architecture.
Component	A software component is a re-useable piece of software that has a well specified public interface and it implements a limited functionality. Software components achieve reuse by following standard conventions.

Abbreviations and acronyms

CERN	European Organisation for Nuclear Research[1]
LHC	Large Hadron Collider
LHCb	LHC beauty experiment
OO	Object-Oriented
OOA	Object Oriented Analysis
OOD	Object-Oriented Design
USDP	Unified Software Development Process
USDP	Unified Software Development Process
URD	User Requirements Document
ADD	Architecture Design Document
UML	Unified Modelling Language
ODBMS	Object Data Base Management System
OMG	Object Management Group
OMG	Object Management Group
OMG	Object Management Group
PB	petabyte, 2 to the 50th power (1,125,899,906,842,624) bytes, a petabyte is equal to 1,024 terabytes.
SGML	Standard Generalized Markup Language (ISO 8879:1985)
DTD	Document Type Definition
XML	eXtensible Markup Language
XSD	XML Schema Definition

1. Laboratoire Européen pour la Physique des Particules

Index