Shine - The new software framework for the NA61 Experiment

MSc. Thesis

Supervisor: Dr. Ágnes Fülöp
External supervisor: Dr. András László
Author: Roland Sipos
External supervisor: Programtervező Informatikus MSc.

Budapest, 2012
Contents

1 Introduction 3
   1.1 Experimental physics 3
   1.2 The CERN NA61/SHINE Experiment 4
   1.3 Thesis objectives 7

2 Legacy software 9
   2.1 Software environment 9
   2.2 Software architecture 10

3 Software upgrade 20
   3.1 Lessons learned from legacy 20
   3.2 Proposal 23

4 The Shine Offline Framework 28
   4.1 Core of the framework 28
   4.2 Modules 41
   4.3 SHOE 46
   4.4 Utilities 50
   4.5 Support 56

5 Integration of legacy software 58
   5.1 Porting technique 58
   5.2 Dependencies 63
## CONTENTS

5.3 Problems ......................................................... 64
5.4 Validation ......................................................... 67

6 Results ......................................................... 68

7 Acknowledgments .................................................. 71
Chapter 1

Introduction

This diploma thesis is a study on the CERN NA61 software upgrade, especially to follow the advancement of the *Shine Offline* Software Framework. In this chapter a description is found about the topics which are related to the development, including detector setup and software systems as parts of the experiment. The most important fields concerning the software upgrade are summarized, starting with experimental physics, which is discussed in section 1.1, followed by an overview of *NA61/SHINE* in section 1.2.

1.1 Experimental physics

In the field of physics research we need to distinguish two main branch in the approach how theories become justified. In the early steps mathematical models are created by researchers of the theoretical physics field. In order to achieve demonstration of correct predictions, experimental physicists develop such instruments that can make measurements on the given phenomena and confront predictions with experimental results. Such apparata in high energy particle physics (*HEP*) are the different types of detectors. Measurements with such detectors are involved with specific software.

There are two essentially different groups of software used in high energy physics research, *online* and *offline software.*
• **Online software**: Applications from this category mostly perform supervising tasks during the time of the experimental data taking also referred to as *run*. These systems are low-level programmed software elements, and stay very close to the detector hardware itself. In this group the most important parts are the Data Acquisition (DAQ), Trigger and all kind of readout, monitoring or controlling systems, such as the Detector Control System (DCS).

• **Offline software**: This group of software meant to process the measured data and manipulate it on different purposes. The principal field of offline software are Simulation, Calibration, Reconstruction and the Data Analysis.

### 1.2 The CERN NA61/SHINE Experiment

SHINE stands for „SPS Heavy Ion and Neutrino Experiment”[1], Its physics goals are precise hadron production measurements for improving calculations of the neutrino beam flux in the T2K neutrino oscillation experiment as well as for more reliable simulations of cosmic-ray air showers. Furthermore it searches for the critical point and the onset of deconfinement of strongly interacting matter in ion-ion collisions. A two dimensional scan of the phase diagram of strongly interacting matter will be done by changing the ion beam energy at the SPS (13A - 158GeV) and the size of the colliding systems. The critical point would be indicated by a maximum in the fluctuation of the particle multiplicity and other physical observables. The onset of deconfinement is revealed by rapid changes in the hadron production properties. The experimental facility is found at the CERN SPS, and consist several types of detecting elements. Mainly inherited equipments from it’s predecessor, the NA49 experiment[2], however several new detectors were developed and installed. The setup has unique and outstanding attributes in order to achieve requirements:

• **Large acceptance**

• **High momentum resolution**

• **High tracking efficiency**
CHAPTER 1. INTRODUCTION

Figure 1.1: The schematic of the setup of NA61 Experiment detector.

- High event rate

1.2.1 NA61 Detector setup

The detector setup with the description of its parts are found in this section (see figure 1.1. and 1.2.).

The detecting elements with their functionality are the following:

- **Time Projection Chambers (TPCs):** These detectors are large gas volumes in which passing particles cause ionization traces. The applied homogeneous electric field causes an upward flow of electrons and leaves a signal through the electronics, and are stored with the help of a data acquisition system. The NA61 setup includes Vertex TPCs (VTPC-1 and VTPC-2), the Main TPCs (MTPC-L and MTPC-R), the Gap TPC (GTPC) and a new supplementary detector, the Low Momentum Particle Detector (LMPD) of this type.
CHAPTER 1. INTRODUCTION

Figure 1.2: The schematic top view of the NA61 Experiment detector setup.

- **Time of Flight (ToF-s):** This type of detectors are used for measure the flight time of particles along known trajectories between two points. This kind of equipment can provide information about particle velocity, which together with a known momentum, provides mass and thus the type of the particle. In the experimental facility these are the TOF-L, TOF-R, TOF-F detectors.

- **Projectile Spectator Detectors (PSD):** NA61 contains a calorimeter, which has been built to measure the number of the non-interacting nuclei in heavy-ion collisions. With the help of this information it becomes easy to determine the number of nucleons participating in a reaction, and thus the event centrality.

- **Further smaller detectors:** These detectors provide specific auxiliary event information which are necessary during reconstruction, An example is the measurement of beam trajectory via Beam Position Detectors (BPDs), to be precise BPD-1, BPD-2 and BPD-3 in the NA61 setup.

The two Vertex TPCs are mounted inside two 450 ton super-conducting magnets. This can produce 1.5 Tesla magnetic field in vertical direction. The field curves the trajectories according to their momentum to charge ratio, and therefore the momentum of the particle can be extracted from the shape. The main part of measurement
data come from the large-volume TPCs for tracking the particle trajectories. The type of the particles is identified on the analysis level also with the help of data from the further detectors, such as TOF-s.

1.3 Thesis objectives

The study, presented in this thesis, gives a detailed description about the software upgrade of the NA61 Experiment. The aims of this thesis can be summarized as follows.

- **Description of legacy software**: Currently the experiment uses the old NA49 software for event reconstruction, simulation and data analysis. The core of the machinery was developed in the early 90’s, on the platform of a client-server software architecture.

- **Software upgrade proposal**: I had the opportunity to participate in an international cooperative work, which aimed for the upgrade of NA61 software. In my Thais I will study the problems in detail which lead to the initiative of the software upgrade. This will motivate the design considerations for the Shine Offline Framework.

- **The Shine Offline Framework**: A new software framework was developed, called Shine, and was implemented with state-of-art solutions in the information technology scene.

- **Integration of legacy software**: Another important goal was a fast migration from the old software to Shine, therefore considerable parts of the old architecture was reused in the new Framework, using wrappers.

The rest of the thesis is organized as follows.

Chapter 2 discusses the old software and its environmental setup.

Chapter 3 provides a brief description of the proposed upgrade, with the software skeleton and the structural design.
Chapter 4 gives a full technical overview where the whole development process will be unfolded with all the main blocks of the machinery, supplemented with the utilities of the new Framework.

Chapter 5 deals with the difficulties of legacy porting. Integration is essential as it can be the basis of a testing, debugging and profiling of offline software. Also the validation technique is found under this chapter.

Chapter 6 summarizes the main results of the developed Shine Software Framework. Also the related publications are found in this chapter.
Chapter 2

Legacy software

In this chapter a detailed explanation is found about the old software along with its basic functionalities. A short overview is presented in section 2.1 about the environment in which the applications are embedded. Then one can find a discussions about the most important element, the DSPACK tool what is presented in section 2.2.1. Then the currently used offline resources and applications are found in section 2.2.2, such as the event structure, reconstruction chain, and the calibration factors.

2.1 Software environment

There are multiple services provided by CERN, on which the NA61 software depends on. These consist of several applications and resources. CERN’s PLUS (Public Login User Service) is a cluster that consists of public machines for interactive work for CERN users. Each node runs SLC (Scientific Linux CERN) distribution which is a recompiled Red-Hat Enterprise Linux with unique program packages to satisfy the researchers needs. With the help of the system, users can work on an independent and secured platform and share the results with their collaborations. The Public Batch Service consists of about 35,000 CPU cores and gives an interface for the users to submit jobs that requires high computing capacity. Mostly the automatised offline related tasks are sent and evaluated with the help of this service. As most of the experiments work with large-scale measurement data, in short time the order of petabytes are generated.
The storing of such amount of experimental data can cause extremely high disk usage, not to mention the backup needs of experiments. CERN provides the Advanced Storage Manager (CASTOR), a hierarchical storage management system, with file storing, listing and retrieving functionality. Usually the Data Acquisition systems collect the files on-the-fly, and upload these to CASTOR.

### 2.1.1 Releases and dependencies

Each experiment can request storage or computational capacity from the services above. For instance NA61 have its own AFS directory where the used offline and online applications are categorized. With time, collaborators make changes on the parts of the existing software which is version controlled. After significant changes the collaboration makes releases and tags from the current copy. A given release comes with an environmental setup to provide information about paths to important resources. To initialize an NA61 software environment for a given release on LXPlus, one can find the appropriate setup scripts near the release. For instance to setup file for the release v2r2-1g_slc5 is found in the following place:

/afs/cern.ch/na61/Releases/v2r2-1g_slc5/etc/na61env_slc5.sh

As NA61 uses significant parts from NA49 software, several environmental variables need to be set in order to start old software, such as setting up the directory of libraries and binaries. On top of the NA49 software, several other dependencies are required before running applications, such as CERN Program Library (CERNLIB)[3] and ROOT Data Analysis Framework[4]. These libraries are widely used in the HEP community and contains several data analysis and mathematical toolkits.

### 2.2 Software architecture

The core of the old software was developed from the early 90’s and mainly inherited from the predecessor experiment. The used techniques are somewhat obsolete today,
but one should keep in mind, that there were serious memory capacity issues in these times.

In computer science there is a software architecture called *Client-Server Model*, which is the base pattern of the main dependency for offline software. The ideology of this schema is, that the element which provides resource or service, and the requesters are logically separated. The ideology was that a kind of server application should handle all data requests and store the description of physics event. Clients asking for actual content from the provider, and manipulate parts of data inside the server. After the task is finished, another client can make different data handling or retrieve the data on demand. With this brief ideology, complex applications were made, where every sub-task is processed by a single client. A chain of these standalones represents an offline application, such as reconstruction.

### 2.2.1 DSPACK

DSPACK[5] is an implementation of a Client-Server Model’s server with extended capabilities, such as combinations of early object oriented principles. The main purpose of this tool is to provide user interface for storing and manipulating experimental data, during an offline related process.

The package was implemented in *C* and *Fortran* languages and uses several old scientific libraries. Furthermore this tool need to be compiled with a non-standard Fortran compiler, because it uses PGI dialect and it’s unique features. The interface that is provided by DSPACK makes possible for data handlers to dynamically define objects from data files. The use-cases became really wide as definitions are able to contain needed run-time options and parameters. Also the standalone clients can use unique structures for data description.

DSPACK can communicate with processing elements in two ways:

1. **Local mode**: Mainly for simple and standalone processing the tool is initialized with this option.

2. **Server-Client mode**: In the case of this option the DSPACK data is stored in
the operating system provided shared memory and the client communication is handled by FIFO-s.

An important issue is, that the mainly applied method is the server-client mode, as relations between sub-tasks are described easier and more agile. For instance parts of the application was changeable in the sense of substitution of standalone clients. Local mode solutions required recompilation on every source code modification.

**Object description**

In the terminology of the tool, a class stands for settling relationships between objects and define options and behavior for these. Manipulations are managed through messages which are sent by class methods. To achieve this, creators implemented an entry point, a so called Class Driver, which are structures to handle objects. Although DSPACK objects are defined by direct calls in interactive mode, to fix settled structures, the *Object Description Files (ODF)* are used. These contain new object definitions and initialization, with given rules and syntax.

This is done via user edited ASCII files. For instance if one would like to define and initialize the object *foo*, then the file should contain the following:

```
DEFINE .foo_t S 2 ! My first DSPACK object definition.
  .INT4    barint
  .foo_t   *foo_p ! Pointer to another foo.
END DEFINE

INIT .foo_t ! Initialization of my object.
  barint   = 42
  foo_p    = 0
END INIT
```

These descriptors are self documenting and agile options for creating complex structure hierarchy, even if clients handle these objects in different aspects. Also these
definitional definitions can be made and understood by DSPACK, written in C or Fortran like syntax.

So the example object description above, written in PGI-Fortran (2.1. source code chunk) and in C (2.2. source code chunk) are also correct.

structure/foot/
    integer barint
    integer foo_p
end structure

Code chunk 2.1: Object definition in PGI Fortran.

typedef struct foo_t {
    int barint;
    foo_t *foo_p;
};

Code chunk 2.2: Object definition in C.

After the definition one can talk about Data Set, based on the DSPACK terminology. This refers to the stored data behind the object description, with given attributes such as the size, name and entry of the given data set.

Environment and tools

To be able to start processing with DSPACK, some environment variables need to be set on the operating system level in the case of client-server mode. Also several supervising tools are added to control the software when it’s running in the background, and these are essential for the initialization of production process. The following list contains the environment variables that must be set in order to use these tools:

- **DSPACK_HOME**: The path to the directory that contains the DSPACK software.
- **DSPACK_FILES**: The path to the directory where DSPACK stores the FIFO files for communication purposes.
- **DSPACK_SERVER**: This variable contains the name of the DSPACK server, as the software is protected from running two server instance in the same time.
DSPACK SECTION1 and DSPACK SECTION2: The package uses a separation of stored data, based on context. Initially DSPACK stores data in two sections, one for short term (event-by-event) and the other for permanent (constant) data. These environment variables hold the size of the sections.

DSPACKRC: If the user wants to use unique configuration, it is done via custom rc files. Basically a set-up file for DSPACK, pointed by this environment variable.

If the environment variables are correctly set, the DSPACK server is started and users can control it. The following list focuses on the frequently used tools, which are available in the client-server mode.

- dsserver: Starts the DSPACK server.
- ds kill: Stops the DSPACK server and cleans up the FIFO files.
- ds open: Opens a DSPACK format file and reads in the header of it.
- ds mark: Select objects for writing them to the opened output.
- ds write: Write the marked objects to the opened output.
- ds close: Close the previously opened DSPACK file.
- ds clear: Clear the temporary section from the DSPACK memory.
- ds db: An interactive memory explorer, for object examination.

The main purpose of the client interface, is the possibility of creating distributed processes of complex task-flows on the same data set. The DSPACK package contains the interface for programs via functions and methods, stored in a shared library.

Clients

Basically every standalone program that attempts to connect to a DSPACK server instance and manipulates data, are called clients. Suppose that previously a client
already created and inserted information about a foo object that was mentioned previously and is stored in a server instance called myDSPACKserver and the definition of foo is accessible from the global namespace.

Our client, written in C (2.3. source code chunk) should follow a function call sequence as follows.

1. Initialize a DSPACK server instance and connect to it.

2. Load in locally defined objects from file or manually via function calls.

3. Load from or save into the DSPACK memory the marked objects with the two most important functions idsget and idsput.

4. If we used communication with file content, close the input or output units with the fsendi and fsendo functions.

5. After the data handling the client can terminate.

```c
#include "ds.h"
#include <stdio.h>

int main(int argc, char** argv) {

    /* The name of the DSPACK server. */
    const char* server = "myDSPACKserver";
    /* The error flag. */
    int ierr;
    /* Data set pointer. */
    foo_t *myfoo;
    /* Kind of secure descriptor to avoid multiple searches and parsings. */
    int iv_foo = 0;
    /* Available entries */
    int foo_num = 0;

    /* Try to initialize DSPACK connection. */
    if (!dsinit(0, server, 2, 0, 0, &ierr))
```

printf("Can't connect to DSPACK server.
");

/* Available entries under the foo_t. */
myfoo = (foo_t *)idsget_ds("foo_t", &iv_point, &foo_num);

/* Iterating through available foo entries. */
for (int i = 0; i < foo_num; ++i)
{
    /* Write out barint, contained by each foo object. */
    fprintf(stderr, "Foo under %d. is %d.", i, foo_t->barint);
    foo_t++;  
}

/* Data handling finished, exiting. */
return 0;
}

Code chunk 2.3: DSPACK dummy client in C.

To represent a complex task, the use of a single client is unconcerned, as the model supports cross-communication on runtime through environment and global variables. One can create standalone programs that use DSPACK to solve sub-tasks of an application. The application itself will be the chain of client calls and DSPACK instructions in a script. The following source code chunk shows an example application that uses DSPACK features.

#!/bin/sh -x

# Some helper environmental variables
SECTION1=100000000
SECTION2=65000000
DSPACK_SERVER=fooDS

# Initialize server instance.
dsserver -s "fooDS" -l "logfile.txt" -m1 ${SECTION1} -m2 ${SECTION2}

# Open output file.
ds open –o "outFile.ds"

# Mark section 2 for output.
ds mark –section 2

# Open raw file.
ds open "../path_to/run-009917x054_exampleevent.padr.raw"
ds rw –count 1

echo "--------- Begin processing on content ---------"

# Read record from section 1
ds read –section 1

# Some processing on DSPACK with various clients.
fooClient1 –arg1 42
fooClient2 –asd –file="../foo.txt"
fooClient3 –option1=23

ds write

echo "------------- Ended processing -------------"

# Close files.
ds close

# Kill DSPACK server instance.
ds kill

Code chunk 2.4: Dummy chain of DSPACK clients.

2.2.2 Offline software

In the offline software concept every task is related to data processing, such as track reconstruction and simulation projects. This section explains each category in a well detailed way, and its application in the NA61 experiment.
Simulation is useful for comparing the measured data with the simulated events, furthermore to measure reconstruction inefficiencies. There is a simulation framework called GEANT[6] that provides a basic interface to make detector description, and numerically simulate interactions. With the help of particle generators, complete simulated events are made. NA61’s simulation software contains a given setup of beam energy in GeV, the type of projectile and target with a selected event generator, such as EPOS, VENUS or GHEISHA to simulate a required amount of data. The detector description stored in GEANT3 format and each release have a reference to a given simulation chain. The format, so the processing techniques sticks to the DSPACK environment.

NA61’s calibration procedure consist several steps for correcting the measured data before reconstruction. As the drift velocity in the TPCs make huge impact on reconstructed geometry, it needs data based calibration. It is an important task to define the proper time delays between the instant of a particle passage through our start detector (S1) and the start of the read-out of given pad of a TPC. This correction is called \( T0 \). There are three kind of contribution to this, such as the global-\( T0 \) that is a delay of the DAQ read-out start. The second contribution is defined for each TPC, called detector-\( T0s \) that is a delay of TPCs with respect to the start signal. The last one is the pad-by-pad \( T0 \) that is similar to the detector T0s, but the synchronization happens between pads. There are several other important calibration factors, such as TPC residuals, TOF calibration. The used software tools are fairly old and lot of development knowledge is not available anymore.

The main task of offline software is the reconstruction. It stands for determining track parameters and obtain an accurate estimation of particle momenta, impact parameters for vertices and the errors of these. NA61 software contains a robust client chain called reconstruction chain to reconstruct the events. The so called run scripts set up the needed environment before the processing and reads raw data binary files. After this clients connect to the DSPACK instance and manipulate the data, and on the very end the result is a DSPACK output file with reconstructed events. The tracking itself follows the Global Tracking Scheme where the main tasks are divided into principle
parts. First the *cluster-finder clients* (e.g.: DIPT) localize cluster from points, then *local tracking* is done on each detector (e.g.: PATREC), finally *momentum fitting* (e.g.: R3D) ends the chain. Along with several important helper clients (MPAT, dedxn61) the reconstruction software results in a very long and rigid DSPACK client call sequence. The experiment suffers from the rigid tools and the difficult-to-use chain, however source codes are access-able in the repository. **Analysis** stands for understanding the data content and gather physics results from the reconstructed events. There are several applications in N61 and the used data are stored in *ROOT* based structures called *miniDST*. 
Chapter 3

Software upgrade

The experiments data taking period started in 2007 and will continue to collect data until 2015. An agile and portable software environment is rather a need than an option to aid the continuous productive work for the Collaboration. Since many years NA61 made huge efforts to maintain the offline software without great success. They made a decision on a software upgrade, based on the known problems and conclusions are found in section 3.1. Considerations are described and based on the sent proposal[7] for the CERN Software Development for Experiments (SFT) group who added further notes[8] on the project. Then one can find the expectations from the new software with its requested key features, and the structural design that are discussed in section 3.2.

3.1 Lessons learned from legacy

As mentioned before not just the main elements of the experimental facility was inherited from the predecessor (NA49), but largely the software was further modified to aid NA61 needs. The reuse of the existing code has lead to several unfinished and incorrect patches. This section contains the major problems of the current software, that forced the collaboration to decide a considerable revision of offline software.
3.1.1 Portability issues

The main pillar of the legacy software is the DSPACK tool what is embedded to the environment that CERN services provide. The software is compiled with the licensed PGI compiler and its environment was set up on the lxplus/lxbatch clusters with Scientific Linux (SLC) 4 Distributions. The Collaboration experienced difficulties with a simple migration to SLC 5 already. The foundation of these problems come from the complicated dependency between used tools and libraries and the robust and obsolete memory handler, the DSPACK. In summary, the rigid environmental setup and fuzzy dependency tree, makes the software package bounded to the CERN infrastructure, that can be disagreeable in many situations.

3.1.2 Concurrent data formats

With years the data model of the old software became highly rigid and provide several concurrent data formats, mainly in DSPACK structures. However the solution supports a generic object-oriented data handling, it is NA61 specific and not used in the HEP community, Apart from this, the greater problem is that the Collaboration uses the DSPACK-based DST that is the output of the reconstruction chain, and converts to ROOT-based miniDSTs for analysis purposes. In the case of problems, deeper data debugging forces the analysis team to use the DSPACK level. The divided data format results in obvious overhead, code duplication, and makes the work difficult for collaboration members.

3.1.3 Mixed programming languages

The main part of current NA61 applications were written in Fortran and C, especially the production clients for simulation and reconstruction. As mentioned before the Fortran issue comes with another circumstance, the PGI dialect what contains several non F77 and F90 features and requires specific knowledge. Both languages became old-fashioned with time and nowadays it is hard to motivate student and young researchers to develop in these languages, not even to understand and modify existing
code. However as detailed in the previous chapters when we talk about production software, generally the discussion is about steering routines, such as reconstruction or simulation chain, therefore we can also mention the extensive use of script languages.

3.1.4 Obsolete software architecture

The ideology of the client-server software architecture for representing complex task and information flow in a system, is associated with many positive aspects. For instance the extensions could be added very easily with a user friendly steering approach. New elements can define private objects and manipulate even the global structures on request. This would be true if the data model would not be rigid and the proper cross dependency map between clients would be settled. If this important thing is missing, to figure out the real mechanism and work-flow is really complicated, as it is the case for NA61 software.

On the other hand if the system includes several hidden bugs, the results become uncertain and make serious side effects inside the applications that causes corrupted data. The clients are standalone programs communicating through global variables and DSPACK structures. This solution is too error-prone and hard to maintain and cannot be allowed with the mission critical data handling.

3.1.5 Out-dated production tools

Another important issue is that the whole detector simulation chain is based on GEANT3. This simulation framework is out-dated and does not even meet present physics requirements. For instance the PSD, cannot be inserted easily to the simulation chain, as the quality of the particle generators implemented in GEANT3 cannot give correct model for the hadronic showers what this sub-detector requires. On the other hand the collaboration is aware about the fact the generally the used techniques are old fashioned and with time, modern and optimized algorithms were invented, but not updated in the software due to the not user friendly system.
3.1.6 Support and documentation

Within the current software there is no testing environment at all. In this case several unpleasant results may turn out after many years when development knowledge is already lost. To resolve such problems and fix bugs costs considerable work-time and experience. Even worse if these bugs do not cause fatal errors, but stays hidden in the software and keep corrupting data, in this case all the wrong information propagates through the applications.

A further problem is the sparse, but at least existing documentation. Partly missing or deficient information is found along the code or among collaboration repositories. Due to this, if one would like to reuse or modify the source code, first the development knowledge need to be re-acquired with reading and reprocessing the code. For example the DSPACK memory manager has a broad documentation with user guide, but it contains several misleading and even incorrect information.

3.2 Proposal

The NA61 Collaboration realized that the further work with the existing software will not meet todays experimental physics and computer science needs. Apart from the fact that is difficult to motivate students and younger researchers to work on obsolete systems, the existing software cannot be restored and maintained properly. On the other hand it is a goal to preserve it in the new software as a supervised legacy structure and further analysis can be made on the legacy about the contained problems.

The experiment concluded on the fact that a full overhaul need to be done in the current system and they recommend the implementation of a new offline software that provides support for simulation, calibration, reconstruction and data analysis. In this section one can find the expected key features.
3.2.1 Unified language and data format

The proposition contains the unification of programming languages, and it sticks to the use of C++ as the only programming language in the new software. Not only because it is widely used and popular among the HEP community, but standard enough to expect that everyone can achieve at least the user-level programming knowledge. With the multi-paradigm style of the language, a wide range of ideology and pattern can be implemented without the sacrifice of computation speed and loose flexibility in the case of modification of the framework.

As described in the previous chapter, NA61 offline software provides several concurrent data format. To avoid confusion a new and unified data model should be invented and structured only with STL\cite{9} data containers. In the case of I/O of the event data should be streamed with ROOT library provided tools, and to reach the setting of verbosity level for the event structure.

3.2.2 Framework and modularity

The new software should follow the software framework abstraction that will be based on the existing Offline Software Framework of the Pierre Auger Observatory\cite{10}. Generally the framework architecture provides universal and agile platform for develop applications and solutions.

As the design contains distinguished key features, such as extensibility. There are three principal pillars describing the planned framework as follows.

- **Modules**: Basically modules stand for processing elements and the user interface for the framework. They should be assembled and sequenced through XML configuration files. This will stand as the application programming interface (API).

- **Event**: The global data model which stores offline related information. Modules can acquire data from the model, and only if needed, they can communicate through this structure.

- **Detector description**: The experimental facility need to be modelled and imple-
mented in the framework and stands as a provider of the updated and correct data what corresponds to detector configuration. Generally to provide a default behavior on the software framework level.

With a framework supervised flow control of processing modules, different applications are created for given offline purposes, such as simulation, or the reconstruction chain. With the framework approach the collaborative development, code exchange, algorithm refinement and sequence modification are done in a fast and easy way.

3.2.3 Legacy porting

Another expectation is the preservation of the legacy software within the new framework. The basic idea is the so called \textit{wrapper modules} for clients and to place the required tools in a \textit{NA49Legacy} library that the framework will support. Other dependencies such as external data and libraries will be integrated to the framework and the whole legacy package will be able to be acquired on demand.

There are multiple motivation behind this solution:

- \textbf{Production readiness}: After the framework is done, it should be used immediately in production. Namely with the legacy reconstruction chain inside, later with the new modules written by users.

- \textbf{Validation}: To be able to prove that the results from the new framework are correct, a validation of ported clients is requested.

- \textbf{Debugging}: In the period of the validation, the reason of differences should be investigated and corrected and back-propagated to the legacy software.

3.2.4 New algorithms

Due to the recent advances in detector simulation, migration from GEANT3 to \textit{GEANT4}[11] became timely. Conversion tools are available so it should not require much effort to place the modified detector descriptions to GEANT4 formats.
On the other hand some basic algorithms will be presented as tutorials for collaborators to make the learning progress easier and to get a clear view about the framework mechanism, such as how to create production ready algorithms what handles event data.

3.2.5 Accessories

The framework will contain an automated testing environment with unit, coverage and stress tests to take control on the errors in the framework, and with an automated buildbot\cite{12} the history and status of the systems health will be monitored during development. This mainly will help the core developers work, but later it helps to log user activities, as each new module will be required to do self-tests.

For such complex software it is necessary to provide user friendly dependency control. However one has to avoid unnecessary dependency usage within the framework. The goal is to keep users install the whole system without major problems. Based on seniors experience from the development of the Auger Offline Framework, a package handler tool was developed, called Shape that can install all dependencies and build up the needed environment for the Shine Offline Framework.

To prevent what happened before with the existing sources, the group need to make efforts to automate the documentation with Doxygen. In addition, providing further detailed information about source code is still necessary.

3.2.6 Manpower

Based on the software upgrade proposal, a core development group was specified with 3 senior and 2 student NA61 Collaboration members to complete the project.

The following table contains the members of the Shine Group:
## CHAPTER 3. SOFTWARE UPGRADE

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>Institute</th>
</tr>
</thead>
<tbody>
<tr>
<td>András LASZLO</td>
<td>senior</td>
<td>CERN, PH-SME (research fellow)</td>
</tr>
<tr>
<td>Michael UNGER</td>
<td>senior</td>
<td>Karlsruhe Institute of Technology</td>
</tr>
<tr>
<td>Darko VEBERIC</td>
<td>senior</td>
<td>University of Nova Gorcia</td>
</tr>
<tr>
<td>Roland SIPOS</td>
<td>M.Sc. student</td>
<td>Eotvos University, Budapest</td>
</tr>
<tr>
<td>Oskar WYSZYNSKI</td>
<td>Ph.D. student</td>
<td>Jagellonian University, Krakow</td>
</tr>
</tbody>
</table>

Also joined to the cooperative work later:

<table>
<thead>
<tr>
<th>Name</th>
<th>Status</th>
<th>Institute</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tom Paul</td>
<td>senior</td>
<td>Department of Physics - Northeastern University</td>
</tr>
<tr>
<td>Marek Szuba</td>
<td>senior</td>
<td>Karlsruhe Institute of Technology</td>
</tr>
<tr>
<td>Antoni Marcinek</td>
<td>Ph.D. student</td>
<td>Jagellonian University, Krakow</td>
</tr>
</tbody>
</table>
Chapter 4

The Shine Offline Framework

In this chapter the review of the developed Shine Offline Framework is found, covering the full technical description, theoretical considerations and implemented structures. First the core of the framework in section 4.1 is presented that contains the fundamental parts of the machinery. After this section, the new and unified event structure is presented in section 4.3, followed by the modular design, that stands as the general user interface for the framework is discussed in section 4.2. The essential accessories are found in the utilities in section 4.4. The chapter closes with the description of the build system and how the external dependencies are handled that is presented in section 4.5.

4.1 Core of the framework

The core of the framework is controlled by a steering and configurating machinery that handles given modules and finds selected configurations. Finally to build up the detector interface to provide correct data about hardware.

The framework’s entry point is the ShineOfflineMain program, that stands for command line option handling and gives the opportunity for users to override the sequencing method. One can use the standard setup provided by the fwk::RunController with a given module sequence. Interruption is barely used and only applied in well-founded cases, when one have a special application which cannot be supervised by
the RunController.

Basically at this point the user has two choices:

1. Implement unique ShineOfflineUser function for custom steering.

2. Provide the sequence XML file with selected modules.

4.1.1 Central configuration

One of the most important expectations from the Shine Offline Framework is the full flexibility in order to create a wide range of applications. To achieve this, the framework needs to be well configurable and be able to provide global tracking and logging of the current state for each run of the program. The singleton class called fwk::CentralConfig stands as the interface to every configuration related task, such as setting up detector description, sequencing or configuring modules.

The goal is to provide access for users to these configuration files and its content through the class instance and give option for read, write and modify the settings in user code, as seen in the 4.1. source code chunk.

```cpp
CentralConfig& cc = CentralConfig::GetInstance();

// Get configuration in a string.
const string fullConfig = cc.GetConfig();

// Jump directly into the XML configuration file, with given config id.
Branch fooBranch = cc.GetTopBranch("FooConfig");

// Even write current configuration to file.
cc.WriteConfig("");
```

Code chunk 4.1: Accessing configuration.

The fwk::CentralConfig is configured via a so-called bootstrap XML file, that is usually passed to the executable as a parameter.

```
./usershineoffline -b bootstrap.xml -l logfile.xml
```

Code chunk 4.2: Bootstrap configuration passing and logging.
The file contains a bootstrap element that may contain many sub-elements, such as:

- **centralConfig** and **defaultConfig**: A branch collection of so-called configLinks that obligatory contains the id that will identify the entry in the configuration dictionary, and the link reference that points to the exact path or URL for the configuration. Optionally two switches for validation and fingerprint checking directives are set under each configLink. The 4.3. XML chunk shows an example configLink.

```
<configLink
  id            = "FooConfig"
  xlink:href    = "./path_to/FooConfig.xml"
  fingerprintCheck = On
  validate      = Yes />
```

Code chunk 4.3: The configLink element.

- **schemaPathReplacements**: These element allow to use different logical names for schema file paths with the replace sub-element, that makes the connection to the machinery easier. Developers do not need to fight with long path names in module configurations, neither with complex path trees along the machinery work flow. The example of usage is shown on the following bootstrap element (4.4. XML chunk):

```
<bootstrap
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNameSpaceSchemaLocation="[MY_SCHEMAPATH]/bootstrap.xsd"
  xmlns:xlink="http://cern.ch/na61/schema/types">

  <schemaPathReplacements>
    <replace old = "[MY_SCHEMAPATH]"
             new = "/very/long/path/to/my/application"/>

  <configLink
    id            = "FooConfig"
    xlink:href    = "[MY_SCHEMAPATH]/FooConfig.xml"
    fingerprintCheck = On
```
Code chunk 4.4: Example of schemaPathReplacement.

- **parameterOverrides**: One can override parameters in config files, with given identity of the configuration, and the sequence of branch types what one would like to re-parameterize. Seen in the example below in the 4.5. XML chunk:

```xml
<bootstrap

  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="./bootstrap.xsd"
  xmlns:xlink="http://cern.ch/na61/schema/types">

  <configLink id="FooConfig1">
    <parametersToOverride1>
      <data>42</data>
    </parametersToOverride1>
  </configLink>

  <configLink id="FooConfig2">
    <parametersToOverride2>
      <dataWithAtt att="newAtt">42.0</dataWithAtt>
      <dataWithUnit unit="meter/second">111.11</dataWithUnit>
    </parametersToOverride2>
  </configLink>

</bootstrap>
```
Example and standard configurations

The bootstrap files may become very long if several modules will be used or there are multiple data sources for a given run. The framework provides already made configuration files for the most typical and frequently used modules, divided by the two group. Both of them requires a declaration of an XML ENTITY that defines an alias, such as in the 4.6. XML chunk, what will be replaced during the XML parsing.

Code chunk 4.6: Alias for configurations.

- **Example configuration** shows for users, how they can manage configurations along their applications. Based on the previous example, we can use the example configuration as follows (4.7. XML chunk).

Code chunk 4.7: Example configuration.
• **Standard configuration** made for different purposes, as they store a kind of initial configuration for applications and modules, such as reconstruction or analysis. The content of these are production ready information that determined by the responsible teams of the offline related tasks. In the framework there is a fixed location for these settled configuration files. The example of standard configuration is seen as follows (4.8. XML chunk).

```xml
<!DOCTYPE bootstrap [
  <!−− alias for THE standard Reconstruction config file −−>
  <!ENTITY standardRecoConfig SYSTEM '@CONFIGDIR@/standardRecoConfig.xml'>
]

<bootstrap>
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="./bootstrap.xsd"
  xmlns:xlink="http://cern.ch/na61/schema/types">

  <!−− get THE standard Reconstruction config −−>
  &standardRecoConfig;

</bootstrap>
```

Code chunk 4.8: Example of standard configuration.

It is essential to catch each error during production runs, if these are related to altered configurations. Every configuration file comes with an Md5 fingerprint that is made during the installation of the framework and checked at run time. If there is a difference, the framework log warning messages.
4.1.2 RunController

In a software framework a key element is how can one steer the application in different setups. The `fwk::RunController` singleton class is responsible for sequencing the appended modules in the configuration, with settled rules contained by an XML file, and for monitoring and tracing the running environment of modules. Generally this makes the bridge between the framework and modules as an interface.

A basic sequence of modules can be the following (4.9. XML cunk):

```xml
<sequenceFile>
  <moduleControl>
    <loop numTimes="2">
      <module FooModule1 </module>
    </loop>
  <moduleControl>
</sequenceFile>
```

Code chunk 4.9: A basic sequence file.

In the mentioned example the RunController first invokes the initialization method of the `FooModule1`, then the processing method twice and finally the finishing function will be called. To achieve more complex sequencing one can define mixed loops in the sequence, such as in the following 4.10. XML chunk.

```xml
<sequenceFile>
  <moduleControl>
    <loop numTimes="unbounded">
      <module FooModule1 </module>
    </loop>
    <loop numTimes="2">
      <module FooModuleA </module>
      <module FooModuleB </module>
    </loop>
  </moduleControl>
</sequenceFile>
```

As modules need to have kind of communication toward to the RunController, the loop mechanism is depends on module states. The processing call must have a return value what describes the actual outcome of the module. For example success or even loop breaking instruction to the controller.

### 4.1.3 Detector description

One of the principal parts in experimental physics software is that how the detector will be modelled and how will be the interface working to access instrument related data. This need to be separated by two important sections as one can see in the figure 4.1.

1. **Detector User Interface**: The user deals with a simple and single interface when want to access the detector element, and all the sub-parts what it contains.

2. **Detector Backend**: The sought data can come from different sources and without restrictions and it is provided with lazy fetching on the request of given data.

#### User Interface

The experimental facility contains several components and physical components that needs to be accessed by the user. The idea is to achieve the mechanism as seen on the following 4.11. source code chunk.

```csharp
// Goal: Get the measured drift velocity value from VTPC-1 detector, from 2010-10-18 with run number 13137.
```
CHAPTER 4. THE SHINE OFFLINE FRAMEWORK

Figure 4.1: Detector description logic.

```cpp
// Get the reference of the detector object.
Detector& detector = Detector::GetInstance();

// Update to the given time and run number.
detector.Update(UTCDateTime(2011, 10, 18, 1).GetTimeStamp(), 13137);

// Get the reference of the VTPC-1 chamber from the Detector.
const VTPC& vtpc1 = detector.GetTPC().GetChamber(TPCConst::eVTPC1);

// Get the drift velocity from VTPC-1.
const double drift = vtpc1.GetDriftVelocity();
```


To achieve this technique the strict model of the detector description is required and its current state, based on time and a given measurement period, marked by a runID. The fixed coordinate system with the time and runID members record the
actual state of the *det::Detector* singleton class.

Secondly it should compose all the elements that contains all getter functions to the objects inside. Each sub-element will stand as a class, inherited from a generic *det::Component* class (see figure 4.2), providing access to properties for components. Each component have to store its own *det::ValidityRange* object what keeps tracking if contained data is valid or have expired during processing. This is essential to warn the users when the required data is probably cannot be used further. The actual state is described with various component pendant data. On top of the runID and timestamp, such as additional sub-element informations, calibration factors and operational properties are found.

The list of the components are the following:

- **PhysicalComponent**: A subclass of Component with some extensions such as spatial position of its geometric center and rotation. The main reason is to separate components that are physically present in the detector, as these needs unique treatment in some cases.
• **MagneticField**: As the detector has a magnetic field, this class describes the current state of the field. It handles multiple magnetic field maps, boundary conditions and all information related to the field. One of the most important part is the `CalculateField` function which computes the magnetic field vector for given co-ordinates.

• **Beam**: This simple class stores the properties of the beam, such as actual *momentum*.

• **Target**: This class holds the actual status for the target, as for a measurement period the target is used or not, in other word *in* or *out* and filled with target material or not, marked as *full* or *empty*.

• **Trigger**: The class holds trigger channel mappings for *BOS* channels.

• **TPC**: The most complex component is an aggregator class for all Time Projection Chambers what are found in the facility. It contains every *TPCClamber* objects, storing all *TPCSectors* for the chamber. Finally each TPCSector contains its *TPCPadrows* and *TPCPads*. In each level the related data are settled, such as calibration factors for chambers, or global informations in the TPC class.

• **DAQ**: An aggregator class to handle Data Acquisition System related information, such as storing geometrical location index for each *TPC pad*.

• **TOF**: The class is an aggregator for all Time Of Flight detectors, each of these described by a *TOFWall* component what stores all property of a single ToF, such as the container of *TOFScintillator* object.

• **PSD**: This component describes the PSD structures with data format, geometry and current status.

• **BPD**: The class stores the description about the Beam Position Detectors, containing plane and strip details and conventions.

The aggregator classes containing sub-elements inside, store their child components in a `boost::ptr_vector` container for a convenient approach. In case of storing them
in `std::vector`, one would need to implement several options, such as access dereference, the begin iterator or pointer deletion.

With this hierarchical structure of detector components it is really simple to add new parts if the facility will be extended. All the getter functions return with constant references to data, as these information is acquired only but not modified by modules.

**Managers**

The detector back-end should provide the possibility to fetch the required data from different sources and formats. This needs a flexible mechanism to allow multiple sources what we call as the Manager ideology, based on the Bridge design pattern to decouple the abstraction from its implementations. The source of data may come via interfaces what controls SQL Databases, XML, TGeo or even plain text.

Each detector component that needs to update property data for a given time and runID, have to send a request for an implemented Manager, inherited from the `det::VManager` protocol class. This is the common interface for every detector managers and manager registers. Every request from components must be identifiable by two strings. The first stands for the component name and the second is its property name, with a so called IndexMap of index types and index numbers. The managers answers with runID and time matching instance of requested data. It is important that this interface defines the type for the passed information between the manager and the interface.

The `det::ManagerRegister` class, inherited from `det::VManager` provides a container for a set of managers. This is the main entry point for the detector user interface for fetching data, as the request is received by the ManagerRegister. It is iterating through the registered Managers and check, which one is capable to provide the required data. Based on the result, the `VManager::EStatus` flag returned, stores the information about the success or the failure of the lookup.

Managers has to implement the `GenericGetData` (4.12. source code chunk) virtual function, where the return data is passed with a void pointer. In the implementation it is necessary to easily find out what is the type of the requested object, so the
typeIdName string is also passed around. To aid this, the Handle generic class stores the void pointer type data and the typeIdName.

```cpp
virtual
EStatus
GenericGetData(Handle& returnData,
    const std::string& component,
    const std::string& property,
    const IndexMap& index)
```


A dummy manager to provide an integer property for the TPCs, with the implementation of its GenericGetData can be the following (4.13. source code chunk):

```cpp
VManager::EStatus
GenericGetData(Handle& returnData,
    const std::string& component,
    const std::string& property,
    const IndexMap& index)
{
    // Currently we handle only one property...
    if (property == "DummyIntProperty") {

        // Check type.
        if (returnData.Is<int>()) {
            returnData = 42;
            return eFound;
        } else if (returnData.Is<std::string>()) {
            returnData = "Fourtytwo";
            return eFound;
        }
    } else {
        return eNotFound;
    }
}
```
CHAPTER 4. THE SHINE OFFLINE FRAMEWORK

The Manager ideology provides agility from the detector description point of view, as the source can vary and it is hidden from users. It comes with a lazy reading of data with a validity range what shows, what time range the data will stay correct and can be used.

4.2 Modules

The structure where users able to insert unique processing elements with framework provided tools, detector description and manipulate the event structure, is the Module. In the framework ideology this stands for the Application Programming Interface (API), with slight modifications. The users are not experts in software technology, so the interface need to be understandable and easily insert implementation.

4.2.1 Module interface

As we described in the \texttt{fwk::RunController}, an interface required, in order to the controller sequentize the module methods. This interface is called \texttt{fwk::VModule} and defines the standard functions to be implemented by the user.

These virtual function are the following (4.14. source code chunk):

```cpp
// Initialization of user code for every run (NOT event).
virtual EResultFlag Init() = 0;

// User written processing, invoked once per event.
virtual EResultFlag Process(Event& event,
                             const utl::AttributeMap& moduleAttributes) = 0;

// Invoked at end of the run (NOT event).
virtual EResultFlag Finish() = 0;
```

Code chunk 4.14: The VModule’s virtual functions.
• **Init**: This function invoked at the beginning of each run, controlled by the Run-Controller. It is really important that the beginning of a run not equals with the beginning of the event. Generally this function implements needed preliminary settings before the run for the module, and to be done only once for initialization.

• **Process**: This method contains the processing code part and what should be done once per event. Two parameters are passed, first the reference to the *Event*, and secondly a constant *AttributeMap*.

• **Finish**: When a run finished the modules *Finish* process called, usually for closing files or writing out data such as histograms.

### 4.2.2 Framework communication

Each module method returns with a so called *ResultFlag* what shows if the current function resulted with success, error or even with further instructions for the controller. The available reports are stored in the enumerated *EResultFlag* what are the following:

• **eSuccess**: No problem occurred in the function, and it returns with success to the RunController.

• **eFailure**: An error occurred in some place of the method. If this was returned to the RunController, and it terminates execution.

• **eBreakLoop**: Reports to the RunController to break the current loop. It is essential if further modules in the loop required modification on event by the actual function.

• **eContinueLoop**: Reports to the RunController to skip the further modules in the loop and continue with next iteration.

The *fwk::VModule* contains the *registration macro* used to register the user modules in the framework. A one-liner should be put at the bottom of the users class definition.
CHAPTER 4. *THE SHINE OFFLINE FRAMEWORK*  44

as follows in the 4.15. code line.

```
REGISTER_MODULE("FooModule", FooModule);
```

Code chunk 4.15: Use of registration macro in modules.

In order to prevent strictly ruled module syntax, but keep a safe and well-balanced way to connect the user code to the framework, this macro (seen in 4.16) was implemented what provides the static getters for registration and subversion identity handling. It inserts the creation function what will be called by the RunController through the module factory what basically provides a mono-state such as structure, the module instance.

```
#define REGISTER_MODULE(_moduleName_, _ModuleType_, _svnId_) 
  
public: 
  
  static std::string GetRegistrationId() { return _moduleName_; } 

  static 
  
  fwk::VModule* 
  Create() 
  
  { 
    static _ModuleType_ module; 
    return &module; 
  } 

  std::string GetSVNId() const 
  
  { return _svnId_; } 

private: 

  utl::ObjectRegistrar<_ModuleType_, fwk::VModuleFactory> fAutoModuleReg
```

Code chunk 4.16: The VModule’s registration macro.

This syntax is very easy and clear to understand, as parameters stands for the module name by which it will be registered within the RunController, and the type what is the class name of the module. Instead of write all the content of the macro mentioned before, users simply connect their code with three obligatory implemented
functions, and with the one-liner.

4.2.3 Example

Here one can find a very basic module described line by line with explanation what a module have to contain (4.17. class header file):

```cpp
#ifdef _MyFirstModule_MyFirstModule_h_
define _MyFirstModule_MyFirstModule_h_
#endif

// For the module interface.
#include <fwk/VModule.h>
// For nice logging.
#include <util/ErrorLogger.h>

#include <sstream>

namespace MyFirstModule {

/**
 \file MyFirstModule.h
 \brief Definition and implementation of MyFirstModule,
 what basically does nothing.

 \author Me.
 */
 class MyFirstModule : public fwk::VModule {

 public:
 MyFirstModule();
 ~MyFirstModule();

 // Implementation of the Init function. Invoked on every run start.
 fwk::VModule::EResultFlag Init() {
   const string initmsg = "Hello for the run from Init!";
   INFO(initmsg);
   }

```
CHAPTER 4. THE SHINE OFFLINE FRAMEWORK

// Implementation of Process function. Invoked on every new event.
fwk::VModule::EResultFlag Process(evt::Event& event,
    const utl::AttributeMap& attr)
{
    const evt::EventHeader& eventHeader = event.GetEventHeader();
    ostringstream msg;
    msg << "Hello for the current event in Process! "
        << "RunNo.: " << eventHeader.GetRunNumber();
    INFO(msg);
}

fwk::VModule::EResultFlag Finish()
{
    const string byemsg = "Goodbye for current run from Finish!";
    INFO(byemsg);
}

private:
    // Any private members can come here...
    const int foo = 42;

    // Finally the registration of my module to the Framework.
    REGISTER_MODULE("MyFirstModule", MyFirstModule);

};

} // End of my useless Module.

#endif

Code chunk 4.17: Module example.
4.3 SHOE

SHOE stands for *Shine Offline Event* and aims to replace *DS, T61* and *NDST* structures by one unified data format. It is based on ROOT and made of streamable collection of classes for offline purposes, and allow simple drawing of structures with the `TTree::Draw` function. The content of information is scalable so supports different verbosity levels. With this option the stored information is flexible and supports extendibility with schema evolution.

The main parts of the new event structures are the following:

- **BosRecord** is a one-to-one copy of the raw data format (BOS). It is a general preliminary format before converting data into `RawEvent`. This sub-structure can be useful for selecting raw data or prepare it for embedded legacy clients.

- **RawEvent** contains decoded and uncompressed `BosRecord` that makes raw data handling easy. All reconstruction modules should access this structures instead of `BosRecord`. The structure is independent from the change of raw data structure.

- **ProcEvent** contains processed raw data and low level manipulated data set, and stands as a helper structure. (E.g.: pedestal-subtracted BPD traces.)

- **RecEvent** (see figure 4.3) holds all information what needed by high level physics analysis. It contains a bi-directional tree from vertices to tracks to hits.

- **SimEvent** contains simulated data, such as generated particles or hits in the detector.

The most important part of data are, what measured in the TPCs. These trajectory elements are categorized in the following way:

- **Cluster**: This class contains a point in space with the covariance matrix and the recorded charge.
• **Track**: This class is a collection of *Clusters* and track parameters what represents a track piece in a given chamber.

• **VertexTrack**: A special track, what is assigned to a vertex and refitted under this assumption.

• **Vertex**: The class represents a collection of *VertexTracks* and defines interaction or decay.

SHOE stores these information within lists and navigation is supported from interaction vertices to tracks to decay vertices to decay products are achieved by following links within the objects to their child objects or parent objects.

### 4.3.1 Event

The event structure stores all information related to a single event, and stands as the parent of all data. It contains the run header with the event and run identification and the trigger setup. Stores raw data via *BosRecord* and *RawEvent* and processed data within *ProcEvent*, *RecEvent* and if needed in *SimEvent*.

The part of the *Event* class looks as the 4.18. source code chunk:
class Event {
public:
...
private:
    EventHeader fEventHeader;     // Event and run Id.
    Trigger fTrigger;             // Trigger information.
    BOSRecord fBOSRecord;         // BOS data. (unformatted)
    RawEvent fRawEvent;           // Raw data. (human readable format)
    ProcEvent fProcEvent;         // Pre–processed event.
    RecEvent fRecEvent;           // Reconstructed event.
    SimEvent fSimEvent;           // Simulated event.
};

Code chunk 4.18: Chunk of the Event class.

The RecEvent holds reconstructed objects, and the number of content will grow fast, including ambiguous track reconstruction results and poorly matched Tracks. Due to the frequent random access, all objects are stored in std::lists and VectorLists. The skeleton of the class is similar to the 4.19. class chunk:

class RecEvent {
public:
...
private:
    std::list<rec::Cluster> fClusterList;
    std::list<rec::Track> fTrackList;
    std::list<rec::VertexTrack> fVertexTrackList;
    std::list<rec::Vertex> fVertexList;

    VectorList<rec::Cluster> fClusters;
    VectorList<rec::Track> fTracks;
    VectorList<rec::VertexTrack> fVertexTracks;
    VectorList<rec::Vertex> fVertices;
    ...
};

4.3.2 SHOE-laces

Each object has a unique index with which it can be directly, random accessed and also the removal of objects is supported. This is made by the implemented \textit{Index<T>} \texttt{objectIndex} where \( T \) is the class name of the object (e.g.: Cluster, Vertex). The 4.20 source code chunk shows an example about navigating through objects in the event structure. (4.20. source code chunk).

```cpp
// start with vertex track (VertexTrack -> Track -> Cluster)
const VertexTrack& vertexTrack = ...;

// could be a V0, so better check if Track exists
if (vertexTrack.HasTrack()) {
    // get Track from RecEvent via Index<Track>
    const Index<Track> trackIndex = vertexTrack.GetTrackIndex();
    const Track& track = recEvent.Get(trackIndex);

    // loop over cluster indices associated to this track
    for (ClusterIndexIterator iter = track.ClustersBegin(); iter != track.ClustersEnd(); ++iter) {

        // get cluster from RecEvent given Index<Cluster>
        const Cluster& cluster = recEvent.Get(*iter);
        // ... do something
    }
}

// Even removal of object is supported.
if (vertexTrack.GetStatus() != 0) {
    recEvent.Erase(vertexTrack.GetIndex());
}
```

Code chunk 4.20: Navigation in SHOE.
4.4 Utilities

This section contains the tools what used by the core framework and in modules for make development process easier and clear. Some of them related to mathematical manipulation, logging, custom template definitions or even file parsing. Generally we call these as utilities and are found within the utl namespace of the Framework.

4.4.1 Time

It is essential for different time formats to be synchronized and stored in a unified format. The framework provides several classes to hold time, with specified cross-converting functions between them. The convention is to use utl::TimeStamp instead of unix_t. The list below contains the frequently used date and time formats, but time related tools are found too, such as utl::Stopwatch or utl::RealTimeStopwatch, and kind of composition classes to store range of time, such as utl::TimeInterval and utl::TimeRange.

- **LeapSeconds**: Leap second is a GPS second where the seconds are in unix time and exceptionally allowed to take the value of 60.
- **Date** and **DateTime**: To specify year, month and day and to store date with hour, minute and second, both without time zone or time system.
- **UnixTime**: The basic class what holds the number of seconds what describes instants in time from Thursday, January 1, 1970.
- **UTCDateTime**: The class holds the date and the time in UTC standard format.
- **TimeStamp**: This class holds a GPS second and nanosecond and provide various methods to convert to Unix Epoch or UTC and the arithmetic operators between TimeStamps and composite classes such as utl::TimeRange and utl::TimeInterval.

With proper implementation one can construct complex time manipulation easily, such as follows in the 4.21. source code chunk.
CHAPTER 4. THE SHINE OFFLINE FRAMEWORK

```cpp
int seconds = 1287979200;

// Convert unix time to (0 stands for nanoseconds):
utl::UnixTime myUnixTime(seconds, 0);

// Convert to UTCDatetime:
utl::UTCDatetime myUTCDatetime(myUnixTime);

// Convert to TimeStamp:
utl::TimeStamp myTimeStamp = myUTCDatetime.GetTimeStamp();

```

Code chunk 4.21: Time conversions example.

4.4.2 Mathematics and Physics

A wide range of implemented mathematical functions and methods are found under utilities. There are several integrated algorithms to handle different formulas, such as Kolmogorov probability, normal distribution, binomial coefficients, product logarithm (Lambert W function) and Moyal product.

Runge Kutta ODE Integrator and Interpolation

ODE (Ordinary Differential Equation) Integrator classes were placed under the math utilities. For proper usage of these classes one can find a very detailed example in the test case of the Runge Kutta integrator under mathematic utilities.

One can find the TabulatedFunction class to hold collection of \((x, y)\) points and provide interpolation on them, based on reimplemented CERNLIB methods. Also spline and polynomial interpolator classes are found inside mathematic utilities.

Unit definitions

To keep data clean, a unified unit system is required. User have to use the unit definitions from utl::ShineUnits in all circumstances. With the help of conversion factors the data will be stored to Shine based SI units, so all dimensional quantities along source
code kept in a single system of units. Both MathConstants and PhysicalConstants were added to utilities to aid consistency and prevent mixing nomenclature.

So the user should write the following code if dealing with units (4.22. source code chunk):

```cpp
// The worst:
double x = 1.5;

// Just slightly better:
double length = 1.5; // Length in meters!

// With Shine:
double length = 1.5*m;

// Allows more complex objects to write.
double a = 4.2/(pow(cm,2)*s*sr*MeV)
// And it is obvious what it means...
```


**Geometry**

The framework provides basic linear algebra structures and coordinate systems stored in an instance of fwk::CoordinateSystemRegistry. It is really important to have the option to choose between reference coordinate systems within which we can find the described detector. With this ideology it becomes easy to switch between coordinate systems on demand. The geometry package contains Point, Vector, Plane classes.

### 4.4.3 XML Reader

As the Framework uses XML files frequently, there is an extension and wrapper utility for the Xerces[13] XML parser called utl::Reader. Also provides different validation options, such as DTD or Schema and iteration on the content of the parsed XML files.

As the information in an XML file is a hierarchical tree structure, the utl::Branch class represents a document branch, what is a DOM element node. The documents
root called the top branch, what consist several sub-branches. Branches have the GetData methods to accessing the data and cast it to the given type argument and makes able to navigating in the document with the GetChild or GetNextSibling functions.

Assume that we want to parse the foo.xml file (4.23. source code chunk), to iterate through every childBranch and fetch the data from these with the GetData function.

The module chunk what reads the file without validation, and writes out the content of every branch and sub-branch of the foo.xml file is the following (4.24. source code chunk):

```cpp
VModule::EResultFlag
FooModule::Process(Event& /*event*/, const AttributeMap& attr)
{
    // Create a Reader for our file, without validation.
    utl::Reader fooReader("./foo.xml", utl::Reader::eNONE);
    // Iterating through all DOM element in the XML file.
    for (Branch b = fooReader.GetTopBranch().GetFirstChild(); b; ++b) {
        const int number = b.GetChild("number").Get<int>();
        const string numberInString = b.GetChild("numberStr").Get<string>();
        ostringstream infoStr;
        infoStr << "New childBranch content! "
                << "Number is: " << number << " 
```
There is a strong link between \texttt{fwk::CentralConfig} and \texttt{utl::Reader}, as the described configuration files are acquired and processed through the CentralConfig. This is the most suggested convention along Framework modules, as follows in the \ref{code:chunk4-25} source code chunk.

\begin{verbatim}
VModule::EResultFlag
FooModule::Init()
{
    // We have a registered configuration for our module in CentralConfig.
    // So we can get access to it easily.
    CentralConfig& cc = CentralConfig::GetInstance();
    Branch topBranch = cc.GetTopBranch("FooModule");

    // Read the Branch as we want.
    string stringConfig = topBranch.GetChild("configBranch").Get<string>();

    return eSuccess;
}
\end{verbatim}


Another feature of the XML parser, that it settles the unique \textit{Shine Schema Types}, what will be frequently used in configurations and contains all historical types, such as \textit{unsigned31BitInt} or different lists (e.g.: \textit{listOfInts}) for aid more agile file parsing such as the \texttt{Get<std::vector<string>>} call on a \textit{listOfStrings} typed branch.

It contains a unit parser to be able to set units as attributes in schema types. For instance a \textit{nonNegativeDoubleWithUnit} typed branch called \textit{lengthBranch} (\ref{code:chunk4-26} XML chunk) is fetched as seen in the \ref{code:chunk4-27} source code chunk.
CHAPTER 4. THE SHINE OFFLINE FRAMEWORK

<br>

<someEntry>
  <length unit="m"> 42 </length>
  <length unit="cm"> 100 </length>
</someEntry>

Code chunk 4.26: Units in XML.

Branch b = fooReader.GetTopBranch().GetChild("someEntry");
int actualLength = 0;

// Get the length what is a non negative double with unit in the XML.
Branch lengthBranch = b.GetChild("length");
actualLength = lengthBranch.Get<double>();
ASSERT_CLOSE(actualLength, 42*m);  // Successful assertion!

// Get next length branch.
lengthBranch = lengthBranch.GetNextSibling();
actualLength = lengthBranch.Get<double>();
ASSERT_CLOSE(actualLength, 100*cm);  // Successful assertion!

// ... and so on.

Code chunk 4.27: Unit conventions in Shine.

4.4.4 Other utilities

The Shine Offline Framework uses custom exceptions stored in the utl::ShineException class what is inherited from std::exceptions. The constructor of the base class creates the exception with a message, what is retrieved by the GetMessage function. The GetExceptionName generic method returns with the name of the exception. A wide range of exceptions are available within Shine, such as NonExistentComponentException if a getter function was called on non-existing components, or the XMLParseException if problem occurred along processing XML files.

The framework contains an advanced log routing and writing what handled by the utl::ErrorLogger class. It can manage different severity levels what shows different
message types such as INFO, WARNING, ERROR or FATAL. The error stream and the verbosity level can be set and messages are logged via macros.

Under the STL (Shine Template Library) there is a collection of special template classes used for supporting consistent and safe handling of some parts of the framework. Some of them are essential for highly important parts such as `evt::Event` or the `det::Detector` to aid proper handling of the created objects such as `utl::ShadowPointer` class what represents a pointer with built-in initialization, deletion and deep copying. All the above described singleton classes are inherited from the `utl::Singleton` class what was implemented on the pattern of Meyers singleton.

4.5 Support

Another important expectation from the Shine Offline Framework is flexibility and portability. Shine and its dependencies should come with user friendly building and installation techniques what can be modified and set up with ease. Also documentation and testing infrastructure was integrated to Shine. The installation instructions of the framework can be found under the experiment’s Twiki page[14].

The Shine Offline Framework comes with a Doxygen documentation[15] what is built with the framework in case one define `SHINE_DOCUMENTATION_ENABLED` for CMake.

4.5.1 Dependency handling

Shine comes with obligatory and one optional dependencies:

- **CMake** to compile and install the Shine Offline Framework.
- **CPPUnit** for the integrated testing environment.
- **ROOT** for the used libraries in the Event structures and in several parts of the Framework.
- **Boost** libraries for utilities and safe solutions such as filesystem handling.
• **Xerces** XML parser for handling configuration.

• **XZ-mini** for xz-format compression utilities.

• **DB-Downloader** to fetch the required calibration files.

• **Doxygen** is an optional, but recommended package for users if building documentation is requested.

These packages can be fetched via the tool called *Shape (Shine Advanced Package Environment)* to make the installation of external dependencies easier. One can obtain it from the top NA61 subversion repository and need to set up a `shaperc` file in the home directory to configure and mark the required packages and set downloading and installation destinations. Then Shape is commanded to install the external packages with the following command:

```
./Shape/shape install externals
```

After the installation the environment need to be set before building the Framework, as it needs to recognize that the externals are available on the system. To evaluate the environment one need to type the following command:

```
eval $(./Shape/shape sh)
```

In csh:
```
eval `./Shape/shape sh`
```

Users who want to use Shine on LXPlus can skip the dependency installation, as CERN provides these packages by default. They need to set up the environment with the script contained by the Framework, on the following way:

```
source Shine/env/lxplus.sh
```

In csh:
```
source Shine/env/lxplus.csh
```
Chapter 5

Integration of legacy software

One of the most important goal of the new software framework that it has to contain the old legacy software and be able to run it in a supervised environment. Resources are provided by the Shine Offline Framework therefore the integrated software packages and tools can use these. The main challenge of legacy porting is the unfolding of the obsolete dependency tree and renovation of the working process with elimination of hidden bugs and prove the correct behavior with validation.

The planned solution is discussed in section 5.1, it is followed by the integrated dependencies that are presented in section 5.2. Technical problems encountered during the porting are found in section 5.3. The chapter closes with the validation of the integrated legacy software what is presented in section 5.4.

5.1 Porting technique

The use of legacy software within the Framework must be possible to be disabled, in case one do not want to use it. Shine contains an option what directs the build system to compile and set up the legacy environment such as defining compiler flags and library or executable destinations. Finally the build system treats the NA49 legacy as a separated CMake project.

The porting work-flow is composed by the following steps:
1. Re-compile DSPACK with gfortran to avoid the licensed PGI compiler and its non standard features. Implement the DSInterface to avoid C style DSPACK calls within the Framework.

2. Insert client sources to the NA49 directory and create wrapper modules for client executables with the use of the macro trick to ifdef away the main functions.

3. Every appended external and Shine dependencies compiled to the NA49LegacyCore, and every legacy wrapper to the NA49Legacy shared library.

4. Implement modules that are required for Framework and Legacy software communication, such as the ClientInitializerSG that handles the initialization of DSPACK and environmental setup.

The memory management tool was renamed to DSHACK to avoid misunderstandings and to mark that it is not equals to DSPACK. The DSInterface was implemented to initialize DSHACK, allocate memory for objects or get required type of data (5.1. source code chunk).

```c
// Get instance of the interface.
DSInterface& dsInterface = DSInterface::GetInstance();

// How to access memory objects:
int n_geom_vt1_t = 0;
geom_vt1_t* const geom_vt1_t_obj = dsInterface.Get<geom_vt1_t>("geom_vt1_t", n_geom_vt1_t);

if (n_geom_vt_t != 1)
    ERROR("Too many GEOM_VT1_T calibration structures found in DSHACK!");

// How to allocate objects:
point_vt1* const pointin_array = dsInterface.Allocate<point_vt1>("point_vt1");
pointin_array[0].iflag = 137
// pointin_array[1].iflag = ...
// ...
```

Code chunk 5.1: Example of DSInterface usage.
Original clients are standalone programs connecting to DSPACK, manipulating data inside and finally writing back changed objects. As we want to make steering with the RunController, module wrappers need to be implemented. Client executables should be compiled rather as libraries, with renamed main functions supplemented with a client name prefix. Every client needs a header file, supplemented with the following lines (5.2. header file):

```c
#ifndef _clientname_main_h_
define _clientname_main_h_
#endif

ifdef __cplusplus
#define EXTERN extern "C"
#else
#define EXTERN extern
#endif

ifdef SHINE_CLIENT_IS_FUNCTION
EXTERN int main_clientname(int argc, char* argv[]);
#endif
#endif
```

Code chunk 5.2: Extern function in a client header.

To replace the main function of the client, one should extend the source file with the following lines (5.3. source code chunk):

```c
#include "clientname.h"

#ifdef SHINE_CLIENT_IS_FUNCTION
int main_clientname(int argc, char* argv[]);
#else
int main(int argc, char* argv[])
#endif
{
  // Connect to DSPACK.
  // Manipulate data in DSPACK.
  // Terminate client.
```

CHAPTER 5. INTEGRATION OF LEGACY SOFTWARE
CHAPTER 5. INTEGRATION OF LEGACY SOFTWARE

```c
return 0;
}
```

Code chunk 5.3: Ported client source.

After compiling the modified clients, Framework wrappers should call these functions for processing. As most of the clients have unique command line parameters and external data files, these are stored in the modules configuration file (5.4. XML chunk).

```xml
<ClientNameModuleSG
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation='[SCHEMAPATH]/ClientNameModuleSG.xsd'>

    <externalFiles>
        ..,/path_to_external_files_of_client/
    </externalFiles>

    <commandLine config="full"> -init -d all </commandLine>
    <commandLine config="minimal"> -init </commandLine>
    <commandLine config="onlyMTPCL"> -d mtl </commandLine>

</ClientNameModuleSG>
```

Code chunk 5.4: Client module configuration.

Client modules define SHINE_CLIENT_IS_FUNCTION and includes the wrapped clients header file what contains the entry point for the client, and the module header registers the module with the macro. The implemented Process function only calls the function to start the client code (5.5. module chunk).

```c
#include <utl/ErrorLogger.h>
#include <evt/Event.h>
#include <na49/NA49.h>
#include <na49/CommandLine.h>

#define SHINE_CLIENT_IS_FUNCTION
#include "ClientNameModuleSG.h"
```
using namespace std;
using namespace fwk;
using namespace utl;
using namespace na49;
using namespace evt;

namespace ClientNameModuleSG {

VModule::EResultFlag
ClientNameModuleSG::Init()
{
    CentralConfig& cc = CentralConfig::GetInstance();
    Branch topBranch = cc.GetTopBranch("ClientNameModuleSG");

    string externalFiles;
    topBranch.GetChild("externalFiles").GetData(externalFiles);

    InitCommandLineCache("clientname", topBranch);
    return eSuccess;
}

VModule::EResultFlag
ClientNameModuleSG::Process(Event& /*event*/, const AttributeMap& attr)
{
    // Tokenize command line options.
    CommandLine& cl = GetCommandLine(attr);
    INFO("Executing: " + cl.GetString() + "\n");

    // Run the client and report on failure.
    const int errorCode = main_clientname(cl.GetArgC(), cl.GetArgV());
    if (errorCode > 0) {
        ostringstream err;
        err << "Client returned with error code: " << errorCode;
        ERROR(err);
        return eFailure;
    } else if (errorCode < 0) {
        ostringstream warn;
CHAPTE... INTEGRATION OF LEGACY SOFTWARE

[Code chunk 5.5: Example of client wrapper modules.]

With the ported clients a new Shine application was made, which replaces the old reconstruction chain. This is the reconstruction module sequence calling the client wrapper modules to manipulate the event structure. The ideology of the porting sounds easy and fast, but unfortunately several hidden and unexpected problems occurred along the porting. Every step contained difficulties to be solved and needed decent work time to maintain.

5.2 Dependencies

Along the porting several unresolved dependencies were detected and integrated to the Framework and linked to the NA49LegacyCore library. All dependencies were available within CERN public services and were fetched from these locations.

The dependencies are the following:

- **Minuit**: Library for function minimization and error analysis, required by several clients.
• *CERNLIB*: Significant part of CERNLIB were added to client sources.

• *PADTRANS*: The library made by NA49 was added what contains general transformation routines from TPC pads to internal or external coordinates.

• *TRKSTEP*: This library stores steering routines for tracking within then NA49 magnetic field.

5.2.1 Calibration constants

The experiment stores calibration files in DSPACK format, and placed at the experiments AFS directory. The collaboration made a catalog for these files, handled by a database. Information about proper filenames and their path for given runID and time can be fetched via the database interface.

In the ShineOffline Framework a tool called *CalibrationFileHandler* parses the XML dump of the database and rebuilds the catalog with containers from the standard library. Thanks to this, users are unbounded from the public services and a robust dependency, the database connection. The tool is required to fetch the correct filenames for the actual state and load the content to DSHACK memory.

5.3 Problems

Based on the nomenclature of the porting we distinguish clients as *compiled, ready to process* and *valid*. Compiled means that all dependencies were added to the NA49 Legacy library and the client was modularized. A client is callable, if it is able to make process without errors, so external file reading and DSPACK communication was successful. A client becomes validated if the manipulated data is correct. Similar to the result of the original client executables or differences are explained.
5.3.1 Compilation difficulties

As clients are standalone programs, global name collisions may occur when two or more client use the same function or variable name. In these cases we supplemented the source of problem with the clientname prefix. Several clients use external programs and libraries as dependencies what are need to be found first and then to be inserted into the Framework.

As the NA49 Legacy software contains ten thousands of source code lines, most of the compilation issues come from the different Fortran compiler. Unique features of PGI Fortran such as STRUCTURE and RECORD directives are not supported in GNU Fortran. Also the access operator is distinct, as PGI uses percent (%) and not dot (.) what GNU does. Usage of POINTER is not default with gfortran and need to be forced by the -fcray-pointer compiler option, and some of these options are different in the two case, such as -Mextend for PGI and -ffixed-line-length-132 for defining characters column maximum number. The following Fortran chunk presents the conversions of sources from PGI to GNU (5.6. Fortran source):

```
c Structure in PGI
STRUCTURE /foo/
    INTEGER id
    REAL num
END STRUCTURE

c Structure is converted into TYPE for GNU:
TYPE foo
    INTEGER id
    REAL num
END TYPE

c Record in PGI
RECORD /foo/ item, array(100)

c Record converted into TYPE for GNU:
TYPE( foo ) item, array(100)

c Access operators in PGI:
item.id = 42
```
One needs to keep in mind that it is impossible to call \texttt{GETARG} and \texttt{IARGC} functions directly from C source, because PGI functions are contained in the library, so \texttt{getarg\_wrapper} and \texttt{iargc\_wrapper} functions were implemented.

The presence of cross dependencies made the porting really difficult, as some clients have unique libraries that other clients use already. Due to this phenomenon the build system was need to be modified to support the continuously changing dependency structure along the porting.

\subsection*{5.3.2 Runtime errors}

After the compilation problems more errors occurred during the running the legacy reconstruction application. Hidden memory violations caused sequence breaking errors and segmentation faults.

The source of these comes from the following:

\begin{itemize}
\item \textbf{Missing pointers}: Because of cross dependencies and the client feeds client philosophy a bug may result with not correct data what the following modules would use.
\item \textbf{Memory leaking}: As the modules are not standalone programs, the operating system will not free the allocated memory after the function returns. In some cases this may lead to side effects.
\item \textbf{Hidden bugs}: Some wrong implementation may remain hidden in the legacy
\end{itemize}
software, but after the porting unexpected errors may occur, because of the different processing mechanism.

Most of the problems were fixed, but still made huge effects on data integrity. In the validation phase most of the differences may come around because of these issues. To find the proof about the correct mechanism and see that the legacy modules results with appropriate data, is really hard.

5.4 Validation

The validation phase aims to prove the correctness of the results from the ported legacy applications. Currently event-by-event comparison based validation is in progress, and is complemented by statistical validation.

Demonstration of difference in floating point number representation between PGI and GNU is done and continued with deeper investigation on each client is in progress.


Chapter 6

Results

In summary this thesis describes the newly completed system and points out, how The Shine Offline Framework became a stable and great software tool for the NA61 Collaboration. All the relevant aspects of the upgrade are covered and due to the in-depth description of the technical details, this work could serve as a future reference for new SHINE developers.

The architectural design follows the framework ideology which is a perfect choice in the case when users are also developers. It provides the application programming interface (API) via Modules, and manages the proper flow of control and supervision of default behavior along the processing.

Data formats are unified and use the Shine Offline Event (SHOE) what is a STL plus ROOT based, streamable collection of classes. The framework was implemented in C++ to be the only and unified programming language. Configuration handling done via XML files to provide a user friendly approach.

The system is portable and can be installed with ease, and external dependencies are controlled in an appropriate way. Collaborators can make offline related tasks not bounded to the CERN services anymore, so offline software became literally offline thanks to the Shine Offline Framework.

The legacy integration makes available to profile and test the old software consistency and correctness, and create a production ready tool with this option.

The Framework comes with several already implemented, frequently used mod-
ules that are necessary for applications, such as EventFileReaderSG for reading events or the ShineFileExporterSG for saving the output of the application into ROOT files. Also for simulation and reconstruction, new algorithms were implemented.

In order to facilitate learning process ExampleApplications were added, such as conversion (DS2SHOE) or offline applications (SimpleAnalysis or LMPDReconstruction). Collaboration members already started to use the framework and develop their applications within it.

The author is grateful that he was able to participate in the Shine Offline Framework development, and subsequently work on new challenges. Original contributions by the author are described below:

- **UnitParser**: The skeleton of the framework relied on the CLHEP (A Class Library for High Energy Physics) package in order to provide unit parsing to the Reader utility that processing XML files. The framework uses various features from the Boost C++ Libraries that contains elements with parsing functionalities such as the Spirit library for Phoenix object definitions. With the developed UnitParser utility the heavy-weight CLHEP dependency was removed as it’s support is already offered by the Boost features. Due to this advantage the tool was also integrated to Auger Offline Framework.

- **Client porting**: Original standalone clients were need to be added to the supervised reconstruction module sequence in the Shine Offline Framework. This required modifications on client code in order to get rid of the compilation and runnability issues. Also several necessary tools and libraries were added to the framework and part of the build system was modified several times to satisfy NA49 Legacy needs. This task resulted in the proper disentangling of the dependency tree.

- **Dependency control**: There are several data files used in the old software environment. These were categorized by a database table that gives unique identity and path to the files. For different time and data taking periods there is a matched set of files. As one of the key feature of the Shine Offline Frame-
work is the portability, it was a goal to avoid database connection handling. The \textit{CalibrationFileHandler} tool parses the database XML dump files to build up the structure of the key system, and finds the proper set of files for given time and data taking periods.

- \textbf{Calibration manager:} The calibration factors are need to be accessible in the Detector User Interface and the provision should follow the \textit{Manager} ideology. This required the modification of \textit{Components} with calibration class members and a supplementary manager that fetch the data. In order to find the best way for the calibration factor integration the \textit{Slow Control} data set is acquirable in the new system.

The \textbf{Shine Offline Framework} will be presented in the \textit{CHEP Conference 2012 in New York} by two poster presentations[16, 17].
Chapter 7

Acknowledgments

I would like to express my gratitude to Dr. András László, my supervisor for help in every questions and for the guide to the world of high energy physics offline software related tasks in CERN.

I would like to thank Dr. Ágnes Fülöp, my supervisor for the support from her and the Eötvös Loránd University, Budapest.

I would like to thank to the Shine Group, in particular to the ‘seniors’, Dr. Michael Unger, Dr. Darko Veberič, Dr. Tom Paul and Dr. Marek Szuba and for the ‘young generation’, Oskar Wyszyński for the experience and knowledge what I acquired during the cooperative work.

I would also like to thank to Dr. György Vesztergombi and Dr. Zoltán Fodor for the support of the Wigner Research Center for Physics, Budapest.

I would like to thank to the NA61 Collaboration, in particular for Dr. Marek Gazdzicki for the support and opportunity of participating in this interesting experiment.

Last, but not least, I would like to thank to my family and for my love for their patience, encouragement and support.
# List of source code examples

2.1 Object definition in PGI Fortran. .............................................. 13
2.2 Object definition in C. ............................................................ 13
2.3 DSPACK dummy client in C. ..................................................... 15
2.4 Dummy chain of DSPACK clients. .............................................. 16
4.1 Accessing configuration. .......................................................... 29
4.2 Bootstrap configuration passing and logging. ............................ 29
4.3 The configLink element. ........................................................... 30
4.4 Example of schemaPathReplacement. ....................................... 30
4.5 Example of parameterOverrides. .............................................. 31
4.6 Alias for configurations. ......................................................... 32
4.7 Usage of example configuration. .............................................. 32
4.8 Example of standard configuration. ......................................... 33
4.9 A basic sequence file. .............................................................. 34
4.10 Mixed looping in a sequence file. .......................................... 34
4.11 Usage of detector interface. ..................................................... 35
4.12 The virtual GenericGetData. ................................................... 40
4.13 An example implementation of GenericGetData. ....................... 40
4.14 The VModule’s virtual functions. ............................................ 41
4.15 Use of registration macro in modules. .................................... 43
4.16 The VModule’s registration macro. ......................................... 43
4.17 Module example. ................................................................. 44
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.18</td>
<td>Chunk of the Event class.</td>
<td>48</td>
</tr>
<tr>
<td>4.19</td>
<td>Chunk of the RevEvent class.</td>
<td>48</td>
</tr>
<tr>
<td>4.20</td>
<td>Navigation in SHOE.</td>
<td>49</td>
</tr>
<tr>
<td>4.21</td>
<td>Time conversions example.</td>
<td>51</td>
</tr>
<tr>
<td>4.22</td>
<td>Unit conventions in Shine.</td>
<td>52</td>
</tr>
<tr>
<td>4.23</td>
<td>The foo.xml file.</td>
<td>53</td>
</tr>
<tr>
<td>4.24</td>
<td>Processing of the foo.xml file with the Reader utility.</td>
<td>53</td>
</tr>
<tr>
<td>4.25</td>
<td>Unit conventions in Shine.</td>
<td>54</td>
</tr>
<tr>
<td>4.26</td>
<td>Units in XML.</td>
<td>55</td>
</tr>
<tr>
<td>4.27</td>
<td>Unit conventions in Shine.</td>
<td>55</td>
</tr>
<tr>
<td>5.1</td>
<td>Example of DSInterface usage.</td>
<td>59</td>
</tr>
<tr>
<td>5.2</td>
<td>Extern function in a client header.</td>
<td>60</td>
</tr>
<tr>
<td>5.3</td>
<td>Ported client source.</td>
<td>60</td>
</tr>
<tr>
<td>5.4</td>
<td>Client module configuration.</td>
<td>61</td>
</tr>
<tr>
<td>5.5</td>
<td>Example of client wrapper modules.</td>
<td>61</td>
</tr>
<tr>
<td>5.6</td>
<td>Fortran conversions.</td>
<td>65</td>
</tr>
</tbody>
</table>
List of Figures

1.1 The schematic of the setup of NA61 Experiment detector. .......................... 5
1.2 The schematic top view of the NA61 Experiment detector setup. .................. 6
4.1 Detector description logic. ........................................................................ 36
4.2 The inheritance UML diagram of the Component class. ............................ 37
4.3 The schematic of RecEvent. ................................................................. 47
Bibliography


[12] Buildbot (Continuous integration system to automate the build/test cycle) [http://trac.buildbot.net].


[14] Installation instructions [https://twiki.cern.ch/twiki/bin/viewauth/NA61/NewOFInstallation].

