Legacy code: lessons from NA61/SHINE offline software upgrade adventure

Oskar Wyszyński¹, Andras Laszlo², Antoni Marcinek¹, Tom Paul³, Roland Sipos², Marek Szuba⁴, Michael Unger⁴ and Darko Veberic⁵

¹ Jagiellonian University, PL
² Wigner Research Center for Physics, HU
³ Department of Physics-Northeastern University, US
⁴ KIT - Karlsruhe Institute of Technology, DE
⁵ University of Nova Gorica, SI

E-mail: Oskar.Wyszyński@cern.ch

Abstract. Shine is the new offline software framework of the NA61/SHINE experiment[1] at the CERN SPS for data reconstruction, analysis and visualization as well as detector simulation. To allow for a smooth migration to the new framework, as well as to facilitate its validation, our transition strategy foresees to incorporate considerable parts of the old NA61/SHINE reconstruction chain which is based on the legacy code of the NA49 experiment[2]. Such a reuse of parts of old code, written mostly in C and Fortran, is an often arising problem in HEP experiments. Apart from the need to properly interface the old and new code, the migration task is complicated in our case due to the use of nonstandard commercial compilers in the NA49 code. In this presentation we will describe the challenges faced during the porting of legacy code and discuss solutions that can help developers embarking on a similar adventure. In particular, we will describe the transition from scattered Makefiles to a monolithic CMake built system, the design of C++ interfaces to the legacy code and the semi-automatic conversion of non-standard PGI-Fortran constructs to code that compiles with GFortran. In addition, the validation of the physics output of the new framework will be discussed.

1. Introduction

NA61/SHINE[1] is a fixed-target hadron spectrometer experiment at the CERN SPS accelerator. The main parts of the tracking detector were, after some upgrades, inherited from the predecessor experiment NA49[2]. The key components of the experimental facility are the Time Projection Chambers along with superconducting magnets for particle momentum and charge measurement, and Time of Flight detectors for particle type identification. This setup provides a large acceptance hadron spectrometer with excellent capabilities for momentum, charge and mass measurements.

The new Projectile Spectator Detector is complementing the setup for centrality measurement in ion collisions. The physics program of NA61 is the systematic measurement of hadron production properties in hadron-nucleus collisions as reference for neutrino (T2K) and cosmic-ray (Pierre Auger Observatory) experiments, furthermore the search for the critical point of strongly interacting matter in nucleus-nucleus collisions via different hadron production observables, such as fluctuation measures.
2. Problem
The current software, similarly to the most expensive parts of the hardware, has been inherited from the NA49 experiment. The core of the present framework, DSPACK, was developed in the early 1990s. Significant effort was invested to reuse the existing code and to develop missing parts. Difficulties in maintenance and development largely impeded data production and analysis. The reconstruction and simulation chain was divided into so-called “clients”, which are stand-alone programs that read and write data following the client-server model. Clients are managed by overcomplicated shell script which is using command-line arguments as well as environmental variables to command the clients. The steering script extends over 2000 lines with rather complicated, but mostly obsolete features. This approach caused considerable amount of errors during production runs.

A hybrid programing technique was used for simulation and reconstruction clients, written mostly in FORTRAN and C. Furthermore, the FORTRAN code is particularly difficult to maintain due to the non-standard features, specific to the Portland Group’s commercial compilers used in NA49.

Documentation is scant and mostly out of date. Consequently, analysis of source code is needed in order to rebuild the most relevant part of the documentation.

The used event data model was highly rigid. Some of the inherited detectors, which are not used in NA61 anymore, have to stay supported. Furthermore, introduction of new detectors was not foreseen during designing of the data structures and therefore some of the old data structures were used in order to accomodate new detector data, leading to large inconsistencies in the code.

In the old framework software testing was not used. All differences in results were debugged manually, consuming abundant amounts of human resources. Besides, without testing, we were exposed to non-fatal but possibly significant bugs persisting in the code.
3. Solution
The new software framework for simulation, reconstruction and data analysis is based on the Offline Software Framework of the Pierre Auger Observatory\cite{3}\cite{4}.

In order to facilitate migration, module wrappers for clients enable us to run the old reconstruction chain in the new framework. Additionally, DSPACK Interface is created to provide the necessary conversions between old and new data structures. Once the framework passes validation, modules are planned to be replaced one by one, giving opportunity to implement new algorithms.

Most of the generation of documentation is automatized via generating it from the source code comments. Doxygen\cite{5} is used to provide this feature.

Integrated rigorous automatic testing was implemented with CppUnit\cite{6}. Each new module is required to provide appropriate tests.

Modularity is a basic concept of the framework, implemented as a sequence of processing modules. New software consist of three general parts:

- processing modules configurable by XML files,
- event data model to store and share data between modules,
- detector description, providing non-event data of the experiment.

In the new framework C++ language is used exclusively instead of hybrid programming. Standard Template Library\cite{7} (STL) data containers are used to unify data structures. Furthermore, the event data is streamed using ROOT\cite{8} to ensure flexible and platform independent input/output. Different levels of detail are realized by selective omission for streaming.

4. Reality
Unfortunately, there were some obstacles to overcome during the realization of the above proposal...

4.1. Changing compiler
In GNU Fortran\cite{9} records and structures are not supported. Instead, one can use type declaration, namely \texttt{RECORD /item/} becomes \texttt{TYPE(item)} and \texttt{STRUCTURE /item/} ... \texttt{END STRUCTURE}
becomes TYPE item ... END TYPE. Furthermore, access operator . must be changed to %, e.g. 
product.price = 3.15 to product%price = 3.15. In order to use pointers -fcray-pointer 
flag must be set.

4.2. Intrinsic functions

The next problem was the impossibility to call IARGC() and GETARG(i,s) directly from C code. 
Surprisingly we discovered that in GNU Fortran these functions were implemented not as library 
functions like in PGI Fortran, but as intrinsic functions. Two simple wrappers were able to solve 
this particular problem.

Listing 1: IARGC wrapper

```fortran
integer function iargc_wrapper()
    implicit none
    intrinsic IARGC
    iargc_wrapper = IARGC()
    return
end
```

Listing 2: GETARG wrapper

```fortran
subroutine getarg_wrapper( i, s )
    implicit none
    integer i
    character(*) s
    intrinsic GETARG
    call GETARG(i,s)
    return
end
```

4.3. Uncovering bugs

DSPACK objects are defined by function calls, by reading data files containing object definitions 
or by reading Object Description Files. Object description file is an ASCII file which may be 
edited by the user in order to add or modify an object. One of many of the unwritten rules tells 
us that comments in the ODF file must be shorter than 80 characters. Unfortunately, young 
members of the collaboration did not know about this rule and veterans forgot...

Listing 3: Declaration of structure

```fortran
typedef struct bpd_data_t DS_TEMPLATE {
    int_t iflag; /* quality flag (several clusters, missing ADC) */
    int_t adc[32]; /* ADC values from strip 1 to 32 (for 16 strip BPDs
        only first 16 elements have any meaning) */
    float_t mean; /* Centroid of highest cluster in NA61 coord. system */
    float_t rms; /* RMS width of cluster in cm */
    int_t maximum; /* maximum ADC value of cluster */
    int_t charge; /* sum of (ADC - ped)*gain in highest cluster */
    int_t sum_of_all; /* sum of (ADC - ped)*gain>0 over all 32 (16) strips */
    int_t detector; /* H1=1, V1=2, H2=3, V2=4, H3, V3, H4, V4=8 */
} bpd_data_t;
```
/* ADC values from strip 1 to 32 (for 16 strip BPDs only first 16 elements have any meaning) */

This comment has 95 characters in length. One could ask why do we face this bug just now? The answer is that different compilers imply different memory layout of the same executable.

Figure 3: Different memory layout.

4.4. Validation and numerical differences
Validation is a process of comparing results of new and old software, in order to confirm that it fulfills intended purpose. In other words, this process ensures that new software can be used instead of the old one.

Unfortunately, one cannot expect to get identical results. Even if all bugs are eliminated, still one can get differences related to numerics. This kind of differences are particularly difficult to separate from other bugs. There is no sharp boundary between seemingly numerically right or wrong results. In particular, when the code is of low quality, one can experience large differences.

Listing 4: Example code to illustrate calculation differences

```fortran
program test
    real tmp, tmp1
    tmp = 50.7**3
    tmp1 = 130323.85
    if(tmp .ge. tmp1) then
        write (*,*) 'Bigger or equal than 130323.85: YES!!!'
    else
        write (*,*) 'Bigger or equal than 130323.85: NO!!!'
    endif
    print *, 'Value of tmp variable: ', tmp
    print *, tmp
    write (*, '(E20.10)') tmp
    write (*, '(D20.10)') tmp
    write (*, '(F20.10)') tmp
end
```

One can construct a trivial example test program to see such an effect: calculate $50.7^3$ and compare it to the right constant in order to choose an action. Results of this simple calculations are different, consequently different action is taken depending on the exact realization of the pertinent algebraic operations.
Listing 5: Output of PGI Fortran code

1 Bigger or equal than 130323.85: NO!!!
2 Value of tmp variable:
3 130323.8
4 0.1303238400E+06
5 0.1303238400D+06
6 130323.8437500000

Listing 6: Output of GNU Fortran code

1 Bigger or equal than 130323.85: YES!!!
2 Value of tmp variable:
3 130323.9
4 0.1303238516E+06
5 0.1303238516D+06
6 130323.8515625000

One may assume that these differences could get potentially large when a complex multi-step arithmetic is processed. However, the currently observed differences in reconstruction variables is fortunately rather small. As a demonstration, two validation plots can be seen in Figs. 4 and 5. The event-by-event differences in the number of clusters is most of the time ±1 corresponding to a sub-permille difference in the total number of clusters per event. There is also good agreement in the reconstructed Y-positions of the clusters with a standard deviation one order of magnitude smaller than the detector resolution.

Figure 4: Differences of number of clusters in vertex TPCs.

Figure 5: Differences in calculation of Y position in vertex TPCs.

4.5. Passing arguments

Neither Fortran nor C compilers check for correct match of types of passed arguments. In order to avoid run-time errors, special care must be taken when arguments are handled. Moreover, it should be kept in mind how arguments are actually passed. Namely, in Fortran all passing is by reference whereas in C, if not indicated differently, passes are by value. Furthermore, arrays as arguments provide additional problems since Fortran keeps arrays in column-major order and the C arrays are in row-major. On top of this nuisance, in Fortran the array index starts with one and not with zero like in C. Passing strings is a particular problem since strings in C are null
terminated and in Fortran length is declared and passed as an hidden argument which needs special handling on the C side of the code.

5. Status and outlook
The reconstruction chain consist of 24 module wrappers of legacy clients. All these modules are ported by now and they are technically functional. Currently, we are in the validation phase in order to assure the correct physics functionality of each of the modules. For this purpose, high level output variables such as cluster positions and track momenta are compared. 14 modules are already validated and we expect the final validation of the full reconstruction chain within the next months. The next step will be the replacement of the legacy algorithms by native modules written in C++.

References
[1] NA61/SHINE experiment, http://na61.web.cern.ch
[2] NA49 experiment, http://na49info.web.cern.ch/na49info
[3] S. Argiro et al.: The Offline software framework of the Pierre Auger Observatory http://www.auger.org/technical_info/pdfs/icrc2005/icrc_2005_116.pdf
[4] Review of NA61 Software Upgrade Proposal, https://indico.cern.ch/conferenceDisplay.py?confId=125760
[5] Doxygen, http://www.doxygen.org
[6] CppUnit - C++ port of JUnit, http://sourceforge.net/projects/cppunit
[7] STL - Standard Template Library for C++, http://sgi.com/tech/stl
[8] ROOT - An Object-Oriented Data Analysis Framework, http://root.cern.ch
[9] Extensions not implemented in GNU Fortran, http://gcc.gnu.org/onlinedocs/gfortran/Extensions-not-implemented-in-GNU-Fortran.html