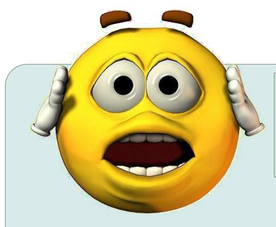
Legacy code: lessons from NA61/SHINE offline software upgrade adventure

Primary authors:
Oskar Wyszynski (Jagiellonian University (PL))

Co-authors:
Andras Laszlo (Hungarian Academy of Sciences (HU))
Antoni Jerzy Marcinek (Jagiellonian University (PL))
Tom Paul (Department of Physics-Northeastern University)
Roland Sipos (Hungarian Academy of Sciences (HU))
Marek Szuba (KIT - Karlsruhe Institute of Technology (DE))
Michael Unger (KIT - Karlsruhe Institute of Technology (DE))
Darko Veberic (University of Nova Gorica (SI))

on behalf of the NA61/SHINE collaboration

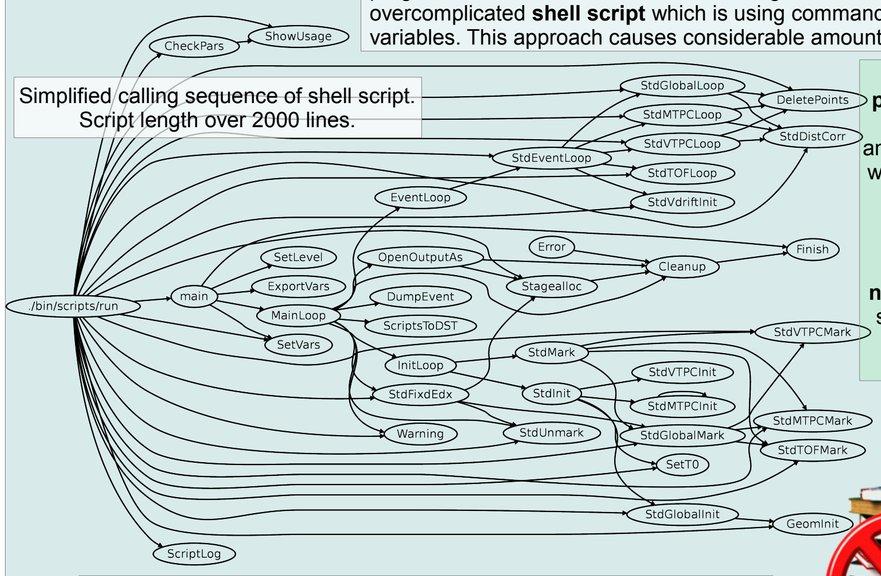


The **current software,** like hardware, has been **inherited from the NA49** experiment.
The core of the present framework, DSPACK, was developed in the early 1990s.
Significant effort was invested to reuse the existing code and to develop missing parts.
Difficulties in maintaining and developing distinctly impede data production and analysis.

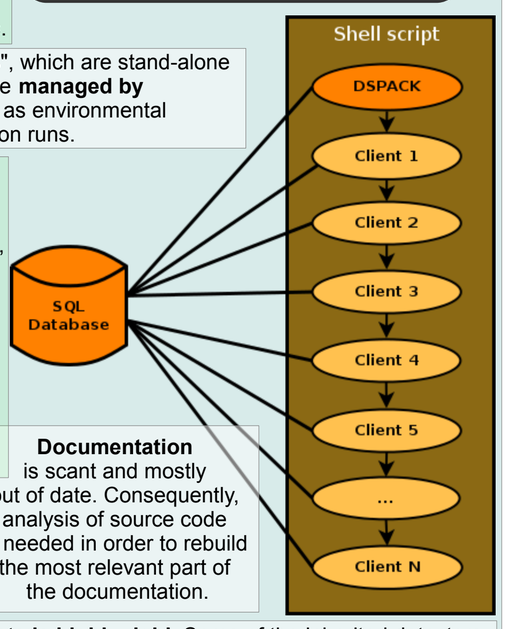
Problem

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. This approach causes considerable amount of error during production runs.

Simplified calling sequence of shell script.
Script length over 2000 lines.



Hybrid programing technique is 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.



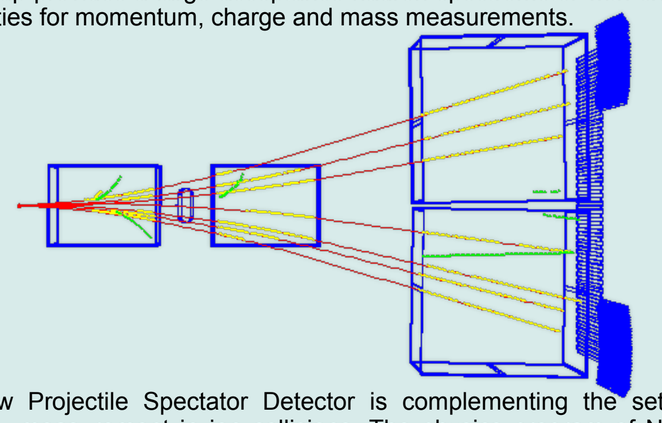
Documentation is scant and mostly out of date. Consequently, analysis of source code is needed in order to rebuild the most relevant part of the documentation.

Model of data is 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 accommodate new detectors data, leading to large inconsistencies in the code.

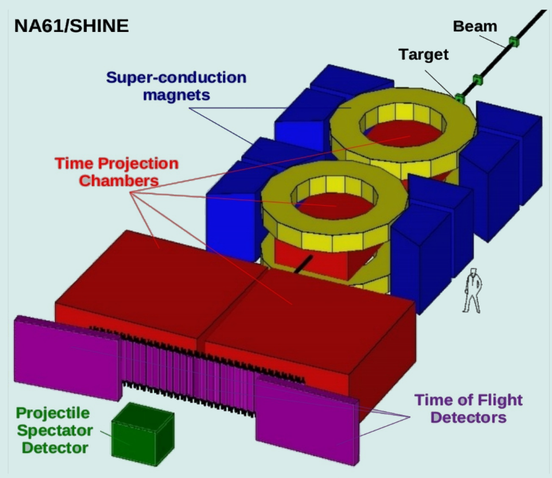
In the old framework **software testing was not used**. All differences in results were debugged manually, consuming abundant amounts of human resources. Besides, without testing, we were exposed to non-fatal but possibly significant bugs persisting in the code.

NA61/SHINE

NA61/SHINE 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. 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.



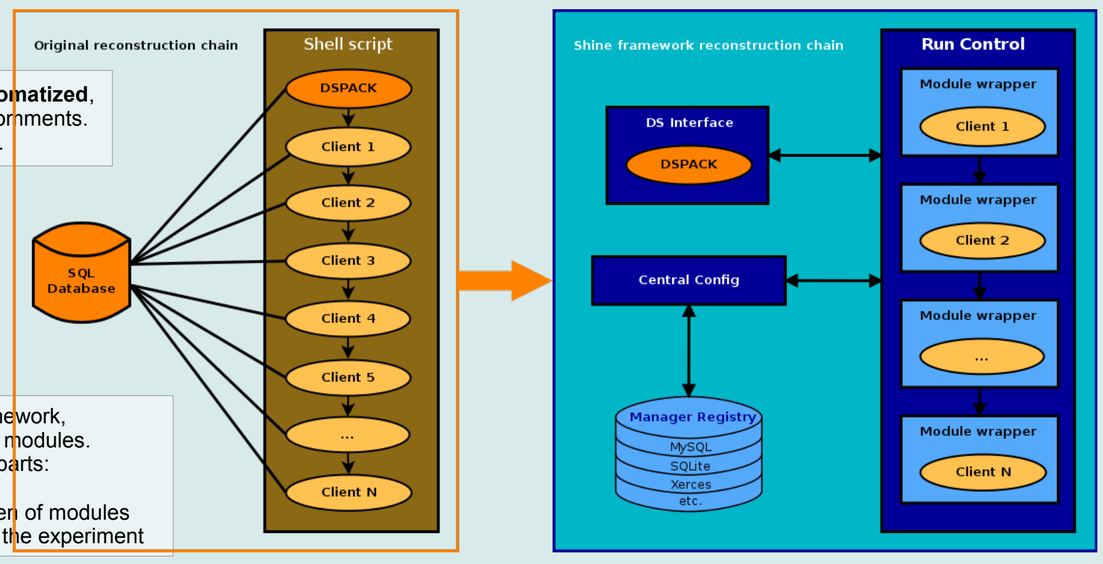
The new Projectile Spectator Detector is complementing the setup for centrality measurement in ion collisions. The physics program of NA61 is the systematic measurement of hadron production properties in hadron-nucleus collisions as reference for neutrino (T2K) and cosmic-ray (Pierre Auger Observatory) 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.



Solution

A new software framework for simulation, reconstruction and data analysis is **based on** The Offline Software Framework of The Pierre Auger Observatory.

In order to expedite migration, **module wrappers** for clients are enable us to run 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 will be replaced one by one, giving opportunity to implement new algorithms.

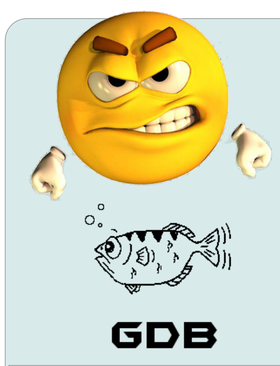


Most of the **generation of documentation is automatized**, achieved by generating it from the source code comments. **Doxygen** is used to provide this feature.

Integrated rigorous automatic **testing system** implemented with **CppUnit**. Each new module is required to provide appropriate tests.

Modularity as 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 of 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 (**STL**) data containers are used to unify data structures. Furthermore, the event data is streamed using ROOT to ensure flexible and platform independent input/output system. Different levels of details are realized by selective omission.



GDB

Reality

Impossibility to call GETARG and IARGC function directly from C code

Reason:

PGI Fortran functions contained by library **GFortran** functions are intrinsic

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

Solution:

Fortran wrappers (getarg_wrapper and iargc_wrapper)

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

Changing compilers delivered by PGI to GCC

PGI Fortran	GNU Fortran
STRUCTURE	None
Record	None
"." as access operator	"%" as access operator
POINTERs are default available	POINTERs are forced by option -fcray-pointer
132 characters column by compiler's option –Mextend	132 characters column by compiler's option -ffixed-line-length-132

```
RECORD /item/ apple, products(100)
```

```
TYPE(item) apple, products(100)
```

```
apple.price = 0.15
products(7).price = 7831
```

```
apple%price = 0.15
products(7)%price = 7831
```

```
STRUCTURE /item/
  INTEGER id
  CHARACTER(LEN=200) description
  REAL price
END STRUCTURE
```

```
TYPE item
  INTEGER id
  CHARACTER(LEN=200) description
  REAL price
END TYPE
```

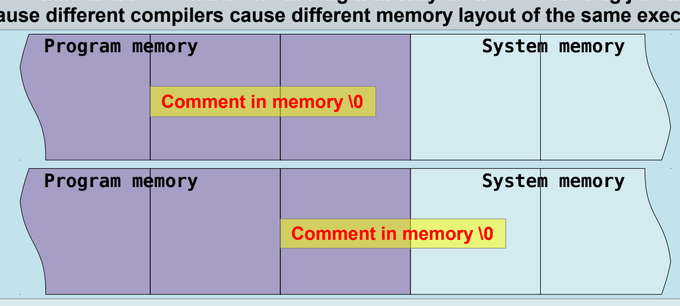
Segmentation Fault during reading text file

DSPACK objects are defined by function calls, by reading data files consisting of 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 tell 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...

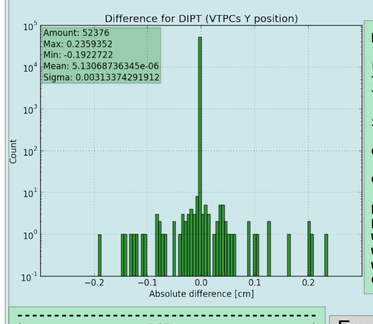
```
typedef struct bpd_data_t DS_TEMPLATE {
  int_t iflag; /* quality flag (several clusters, missing ADC) */
  int_t adc[32]; /* ADC values from strip 1 to 32 (for 16 strip BPDs only first 16 elements have any meaning) */
  float_t mean; /* Centroid of highest cluster in NA61 coord. system */
  float_t 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 !!! Why do we see this bug just now?? Because different compilers cause different memory layout of the same executable.



Validation & Numerical differences



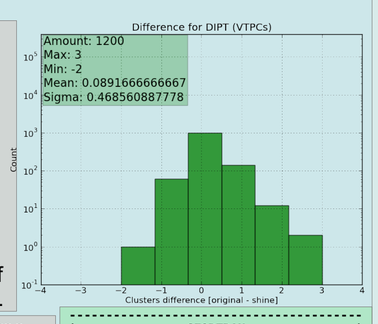

```
program test

real tmp, tmp1
tmp = 50.7**3
tmp1 = 130323.85

if(tmp .ge. tmp1) then
  write (*,*) 'Bigger or equal that 130323.85: YES!!!'
else
  write (*,*) 'Bigger or equal that 130323.85: NO!!!'
endif

print *, 'Value of tmp variable: '
print *, tmp
write (*,"(E20.10)") tmp
write (*,"(D20.10)") tmp
write (*,"(F20.10)") tmp
end
```

Validation is a process of comparing results of new and old software, in order to conform that it fulfills intended purpose. In other words, this process ensures that new software can be used instead of the old one. Unfortunately, we cannot expect to get identical results. Even if we eliminate all bugs, still we can get differences related to **numerics**. This kind of **differences** are particularly difficult to separate from other **bugs**, there is no sharp line between what's right or wrong. Especially, when code is of low quality, you can experience large differences.




```
|                PGI                |
Bigger or equal that 130323.85: NO!!!
Value of tmp variable:
   130323.8
   0.1303238400E+06
   0.1303238400D+06
  130323.8437500000
```

For **example**, as you can see from a trivial test program, which **calculates 50.7^3** and compares to constant in order to choose an actions. Surprisingly, results of this simple calculations are different, consequently different action was chosen Imagine how large the differences can be, when multi-step complex arithmetic is processed. As an example of results two plots were chosen, they show differences in number of clusters and their Y position.

```
|              GFORTRAN             |
Bigger or equal that 130323.85: YES!!!
Value of tmp variable:
  130323.9
   0.1303238516E+06
   0.1303238516D+06
  130323.8515625000
```

Segmentation fault during passing arguments

Reason:

```
Wrong calling sequence in com_h.F : 36 & 55
call ds_namtok(command,' ',com,n,arg,mm,ierr)
```

Solution:

```
Correct calling sequence:
call ds_namtok(command,' ',com,arg,n,mm,ierr)
```

Neither Fortran nor C compilers **are checking for correct match** of types of passed arguments. With the purpose of avoiding 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 references** 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**. Except this nuisance, **in Fortran 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.

Status & 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 to assure the correct physics functionality of each of the modules. For that purpose, high level output variables like 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.