Finanziato dall'Unione europea
NextGenerationEU
Ministero dell'Università e della Ricerca
Italiadomani
PIANO NAZIONALE DI RIPRESA E RESILIENZA
ICSC
Centro Nazionale di Ricerca in HPC, Big Data and Quantum Computing

Declarative paradigms for analysis description and implementation
-
Update

Alberto Annovi, Tommaso Boccali, *Paolo Mastrandrea*, Andrea Rizzi

(INFN e Università di Pisa)

ICSC-S2-WP2 Meeting
10/09/2024

Analysis paradigm: declarative vs imperative

- These is a correlation between the paradigm used for the description of the analysis algorithm and the *programming paradigm* used for its implementation in a *software program*.

Main paradigm approaches

(from Wikipedia)

There are two main approaches to programming:[1]

- Imperative programming – focuses on how to execute, defines control flow as statements that change a program state.
- Declarative programming – focuses on what to execute, defines program logic, but not detailed control flow.

- So far mainly imperative paradigms have been used for analysis description and implementation
  - More straightforward application for "simple" tasks and linear/serial tools

- What has changed in the last decade?
  - **HW** : parallelism/multithreading
  - **SW** : more expressive programming languages (Python, C++ 17/20/23)
  - **Tasks** : increased complexity, increased data size (analyses, combinations)

Why a declarative approach?

- Benefits of a (***more***) **declarative paradigm**:
  - ***Deeper decoupling*** between algorithm and implementation
    - Faster analysis development
    - Wider portability of an analysis (different datasets/experiments)
    - Stronger preservation of the results
  - ***Better scaling*** of development and preservation for increasing complexity of the algorithms and size of the data
  - Simpler parallelization of the tasks
  - Better support for automatic (technical) optimisation
  - More flexibility: e.g. different backend processors

- The development of analysis frameworks based on (more) declarative paradigms is growing momentum in the last years across the whole HEP community (e.g. Analysis Description Languages for the LHC)

- Risks assessment:
  - *"It's just another framework"* : Too specific / not general enough
  - *"It would be great if it worked"* : Too general / not customizable enough

Plan and activity

- ***Target*** : a framework/toolbox able to:
  - support a declarative approach for the analysis description
  - interact smoothly with the standard tools (e.g. ROOT) and data samples (e.g. nanoAOD, PHYSLITE)

- ***Plan*** :
  - ***Development*** of a toolbox able to support declarative approach in HEP analysis:
    - Re-implementation from ***NAIL***
    - Improved modularity and scalability
  - ***Extensions*** to:
    - **multiple input data-forma**
    - **full analysis chain**
  - Test and optimisation phases will benefit from cutting-edge HW resources and community feedback

- ***Activity*** :
  - Development ongoing in **ICSC-S2-WP2** - beneficial interactions with other spokes/experiments
  - Documentation (starting)
  - Presentation/interaction: workshop + conferences + discussion starting with ATLAS and ROOT teams

References & documentation (in progress)

- NAIL

- ICSC-S2 Annual meeting 2023 (CINECA, Casalecchio di Reno, 18-20/12/2023)
- ACAT 2024 (11-15/03/2024, Stony Brook, USA, proceedings submitted for publication)

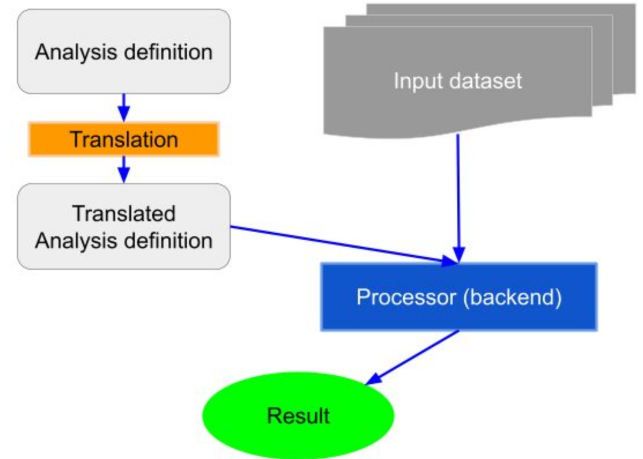
- CHEP 2024 (Krakow, Poland, 19-25/10/2024)

Data-format interface

Scalar	Obj_x	Obj_y	nVec	Vec	nCol1	Col1_x	Col1_y	nCol2	Col2_x	Col2_z	Col2_q
				Vec		Col1_x	Col1_y		Col2_x	Col2_z	Col2_q
				Vec		Col1_x	Col1_y		Col2_x	Col2_z	Col2_q
				Vec					Col2_x	Col2_z	Col2_q

- In principle 3 *equivalent* - but in general distinct - data-formats are involved in an analysis definition:
  - **a.** data-format used inside the framework for variables manipulation
  - **b.** data-format used in the description of the algorithm by the user
  - **c.** data-format used in the encoding of the input data to be processed
- **a** and **b** can - in principle - be unified for most applications
- **c** is *experiment dependent* : a translation is needed **a ↔ c**

- Strategy:
  - Translation via a configurable dictionary tool (Python)
  - Encode all the data-format specific information (and configurations needed) in a dictionary (JSON file)



Toolbox development status

- Tools:
  - ✓ Graph handling
  - ✓ Sample Processing : event loop definition
    - ➡ handling of systematic uncertainties to be added
  - ✓ Interface Dictionary : translation interface + variable definitions
  - Back-end processors (for event loop):
    - ✓ Basic loop processor (C++ compiled)
    - ✓ RDF-based processor (C++ compiled)
    - ➡ Direct python processor
  - ➡ Processors for full analysis chain : ***in development***

- Supported interfaces:
  - ✓ nanoAOD (CMS)
  - ➡ ✓ PHYSLITE (ATLAS)

Example: (over-) simplified energy calibration scheme

Configuration and extensions

- Stand-alone Python package

- Dependences/needs:
  - g++ for back-end C++ compilation
  - Environment with ROOT available
    - Used in most of the analysis
    - Math tools used frequently
    - “Magic” type identification - needed to build the back-end processor
  - Dictionary (used both in analysis description and back-end building stages)

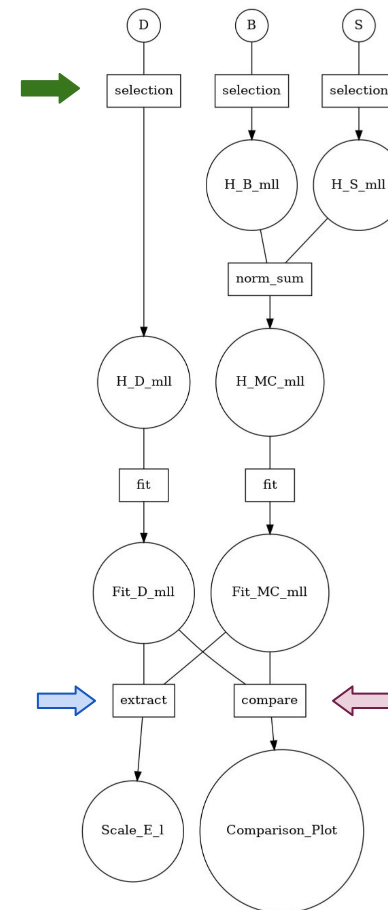
- Analysis description in **one JSON file** :
  - contains both analysis description and actual dictionary

Features

- Hash code tagging:
  - Avoid repeating the steps already available
  - Feature implemented (but not tested yet)
  - Useful application to both event loops and full chain analyses
- Graph tool
  - DAG (Directed Acyclic Graph) custom implementation - used in both event loop and full-chain analysis
- Sample Processor (event loop)
  - Full set of declarations (under test - inherited from NAIL)
  - Interface dictionary used by the definitions (to be re-evaluated for the next iterations)
  - Event “Regions” defined according to selections
  - Systematic uncertainty handling (to be implemented)
- Back-end processor (event loop):
  - C++ code generated by the Sample Processor and compiled (g++)
  - Plain loop more readable than RDF - RDF ~30% faster processing

Extension to full analysis chain

- Example of a full analysis chain (dummy-style calibration task):
  - Sample preparation
  - **Event Loop (NAIL - fully re-implemented now)**
  - Snapshot/data reduction
  - Combination / comparison of distributions
  - Statistical analysis / Extraction of results
  - Selective / incremental execution
- Status: development of a base structure, check for completeness
- Plan for demonstrator:
  - Single task prototype (e.g. Event Loop)
  - Incremental extension to other tasks

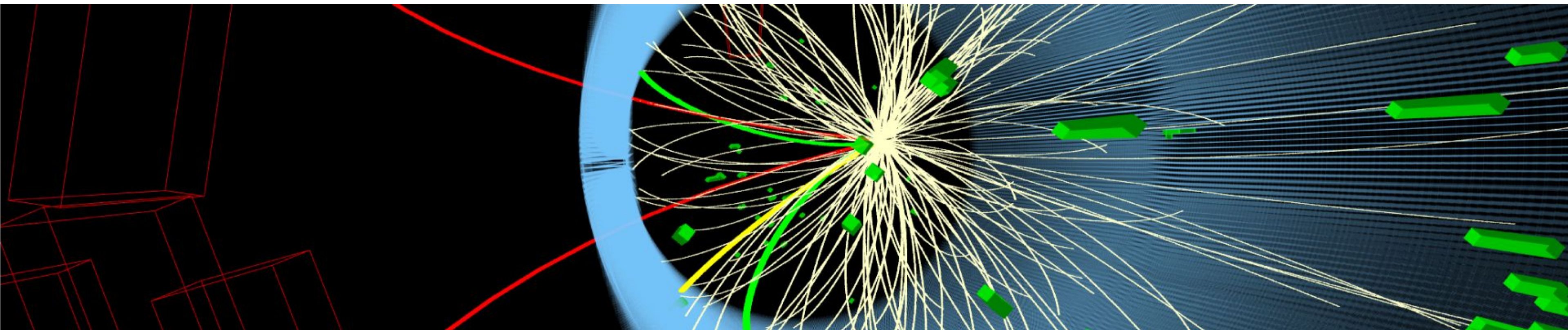


Example: (over-) simplified energy calibration scheme

Summary

- The application of (more) declarative paradigms in analysis description and implementation
  - Can boost analysis' speed (development and execution), preservation and portability
  - Is growing interest through the whole HEP community
- Plan:
  - Build on modern and recent developments (e.g. CMS' NAIL with nanoAOD data-format)
  - Extend the **data-format interface** to
    - Development, integration and validation (e.g. ATLAS' PHYSLITE)
  - Extend the framework to a **full analysis chain**
  - Test and optimisation phases will benefit from cutting-edge HW resources and community feedback
- Activity in full swing - looking forward to comments/ideas/feedbacks

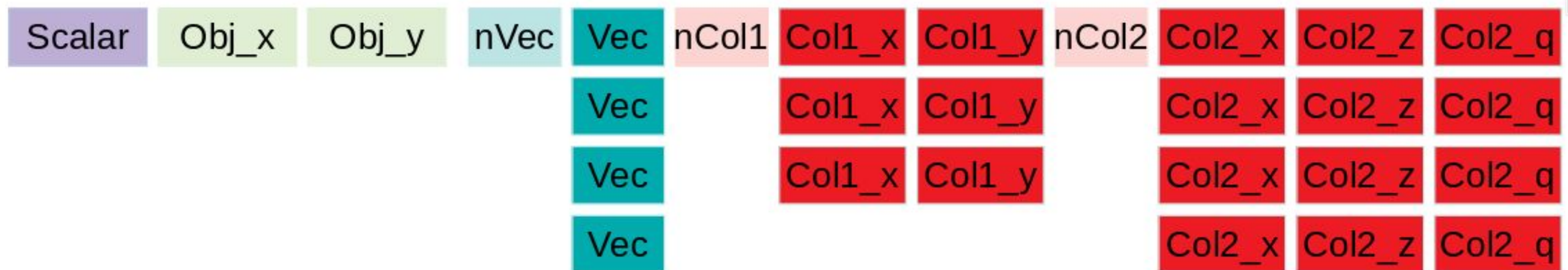
Backup



NAIL (Natural Analysis Implementation Language)

- *"NAIL is an analysis language that should allow to define in an abstract way a data analysis of a typical HEP experiment such as CMS or ATLAS. NAIL assumes an input data model for the event to process (...) and allow to specify the event by event processing actions in a declarative form. Analysis variations for optimizations and systematics do not need to be explicitly coded but are automatically derived from the event processing computational graph. Currently ROOT's RDataFrame is used as backend for a concrete implementation of the event processing as it allows parallelization and lazy evaluation."* (from the README file of the NAIL package)

- Developed in the CMS collaboration, main developer Andrea Rizzi
- Based on CMS' **nanoAOD** (*columnar*) data format, written in Python, heavy lift in C++ (RDataFrame)



AoS vs SoA

- From Wikipedia : *"In computing, an **array of structures (AoS)**, **structure of arrays (SoA)** … are contrasting ways to arrange a sequence of records in memory, with regard to interleaving, and are of interest in SIMD and SIMT programming."*

AoS

```
1  struct point3D {
2      float x;
3      float y;
4      float z;
5  };
6  struct point3D points[N];
7  float get_point_x(int i) { return points[i].x; }
```

SoA

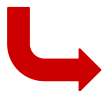
```
1  struct pointlist3D {
2      float x[N];
3      float y[N];
4      float z[N];
5  };
6  struct pointlist3D points;
7  float get_point_x(int i) { return points.x[i]; }
```

- CMS: SoA (e.g. nanoAOD)
- ATLAS: AoS interface with SoA memory storage (e.g. xAOD, PHYSLITE)

Where the increased speed comes from?

- **RVec**
  - *“A "std::vector"-like collection of values implementing handy operation to analyse them.”*
  - [Documentation]
  - Optimised for **speed**
  - Its storage is contiguous in memory
  - **Automatic internal loop**

```
std::vector<float> goodMuons_pt;
const auto size = mu_charge.size();
for (size_t i=0; i < size; ++i) {
   if (mu_pt[i] > 10 && abs(mu_eta[i]) <= 2. &&  mu_charge[i] == -1) {
      goodMuons_pt.emplace_back(mu_pt[i]);
   }
}
```



```
auto goodMuons_pt = mu_pt[ (mu_pt > 10.f && abs(mu_eta) <= 2.f && mu_charge == -1) ]
```

Where the increased speed comes from?

- **RDataFrame**
  - *"ROOT's RDataFrame offers a modern, high-level **interface** for analysis of data stored in TTree , CSV and other data formats, in C++ or Python.*

    In addition, multi-threading and other low-level optimisations allow users to exploit all the resources available on their machines completely transparently."
  - Documentation
  - Optimised for **speed**
  - **Lazy** evaluation and **automatic internal loop**

```
TTreeReader reader("myTree", file);
TTreeReaderValue<A_t> a(reader, "A");
TTreeReaderValue<B_t> b(reader, "B");
TTreeReaderValue<C_t> c(reader, "C");
while(reader.Next()) {
   if(IsGoodEvent(*a, *b, *c))
      DoStuff(*a, *b, *c);
}
```

```
ROOT::RDataFrame d("myTree", file, {"A", "B", "C"});
d.Filter(IsGoodEvent).Foreach(DoStuff);
```