Feature article

REST-for-Physics, a ROOT-based framework for event oriented data analysis and combined Monte Carlo response ☆

Konrad Altenmüller [a], Susana Cebrián [a], Theopisti Dafni [a], David Díez-Ibáñez [a], Javier Galán [a,*], Javier Galindo [a], Juan Antonio García [a], Igor G. Irastorza [a], Gloria Luzón [a], Cristina Margalejo [a], Hector Mirallas [a], Luis Obis [a], Oscar Pérez [a], Ke Han [b], Kaixiang Ni [b,*], Yann Bedfer [c,d], Barbara Biasuzzi [c,d], Esther Ferrer-Ribas [c,d], Damien Neyret [c,d], Thomas Papaevangelou [c,d], Cristian Cogollos [d,e], Eduardo Picatoste [d,e]

[a] *Center for Astroparticles and High Energy Physics (CAPA), Universidad de Zaragoza, 50009 Zaragoza, Spain*
[b] *INPAC, Shanghai Laboratory for Particle Physics and Cosmology, Key Laboratory for Particle Astrophysics and Cosmology (MOE), School of Physics and Astronomy, Shanghai Jiao Tong University, Shanghai 200240, China*
[c] *IRFU, CEA, Université Paris-Saclay, F-91191 Gif-sur-Yvette, France*
[d] *Institut de Ciències del Cosmos, Universitat de Barcelona, Barcelona, Spain*
[e] *Departament de Física Quàntica i Astrofísica, Universitat de Barcelona, Barcelona, Spain*

ARTICLE INFO

ABSTRACT

The REST-for-Physics (Rare Event Searches Toolkit for Physics) framework is a ROOT-based solution providing the means to process and analyze experimental or Monte Carlo event data. Special care has been taken to the traceability of the code and the validation of the results produced within the framework, together with the connectivity between code and stored data, registered through specific version metadata members.

The framework development was originally motivated to cover the needs of Rare Event Searches experiments (experiments looking for phenomena having extremely low occurrence probability, like dark matter or neutrino interactions or rare nuclear decays). The framework components naturally implement tools to address the challenges in these kinds of experiments. The integration of a detector physics response, the implementation of signal processing routines, or topological algorithms for physical event identification are some examples. Despite this specialization, the framework was conceived thinking in scalability. Other event-oriented applications could benefit from the data processing routines and/or metadata description implemented in REST, being the generic framework tools completely decoupled from dedicated libraries.

REST-for-Physics is a consolidated piece of software already serving the needs of different physics experiments - using gaseous Time Projection Chambers (TPCs) as detection technology - for detector data analysis and characterization, as well as generic R&D. Even though REST has been exploited mainly with gaseous TPCs, the code could be easily applied or adapted to other detector technologies. We present in this work an overview of REST-for-Physics, providing a broad perspective to the infrastructure and organization of the project as a whole. The framework and its different components will be described in the text.

---

1. Introduction

REST-for-Physics[1] (Rare Event Searches Toolkit for Physics) is a collaborative software effort providing a common framework and tools for acquisition, simulation, generic data analysis, and detector response in experimental particle physics. An ambitious feature of REST-for-Physics is its capability to analyze and compare both Monte Carlo and experimental data using the same *event processing* routines upon a unified *event data - metadata* architecture. The framework was born to bring together different software requirements related to gaseous Time Projection Chambers (TPCs) in the context of Rare Event Searches, and to unify and coordinate various independent developments in a common modular infrastructure with potential for scalability and reusability. Special care has been taken to ensure the traceability and reproducibility of the results obtained after the data processing, linking the code version with the metadata version stored on disk, and protecting such relation. Any user local changes to the code are identified at compilation time. This is used to guarantee that the executed version and the results written to disk correspond to an unmodified official public release. This fact is extremely relevant when planning to register official experimental data and preserve it for historical reasons, such as covering the data management plan of scientific collaborations, including the release of data to be publicly exploited outside the collaboration domain. The code updates are periodically published in the Zenodo citations system, where a reference to the latest official release is found [1].

REST-for-Physics is the result of several years of experience on detector physics and research, motivated originally to cover the software needs of the T-REX project for neutrino and dark matter searches [2,3]. The REST-for-Physics code has benefited from several academic works, as it becomes apparent in several PhD thesis publications [4–10] that have contributed to shape and define the final project that is described in this manuscript. This project has contributed to the development of different but interconnected research activities in a coherent way, unifying common tools that are regularly used today not only in research but also at all the academic levels, from undergraduate to master students.

Different experimental projects have seen REST-for-Physics growing from its preliminary stages to the mature project we present in this work. REST-for-Physics has been evolving within, and it is being used to produce results at, CAST [11], TREX-DM [12], PandaX-III [13–15], and IAXO [16]. Those projects have benefited from the consolidation of REST as a common tool widely used among collaborators to process, register and analyze detector data. The use and development of REST in other experiments is encouraged in a community effort to maintain appropriate tools for related tasks. In addition to sharing the know-how and experience in our physics domain, the motivation to release a public common framework resides in providing the possibility to distribute the experimental data following a unique format readable with REST-for-Physics, or any other ROOT I/O compatible code, in a future open-data program of the experiments. The code is open-source and it is distributed under a GNU public license at *GitHub* [17].

The aim of this document is to give the reader a broad perspective of the purpose of the software project, its organization and contents, and the basic instruments that shape the whole infrastructure, giving an idea of its scalability potential, and in addition, showing the code validation strategy and continuous integration philosophy. For further reference, detailed information is provided, including an Application Programming Interface (API) class documentation for developers [18] synchronized daily with the latest development version, and a comprehensive guide for first time users [19]. An additional communication channel is available in the form of a public forum [20] to encourage discussion about topics related to our field, help others on their first steps using REST and/or integrate their first routines inside the framework, and discuss about new or existing feature upgrades.

The REST-for-Physics potential resides on its capability to be used with Monte Carlo or experimental data, or a combination of both in an event processing chain. REST is used for modeling, simulation and/or detector response, but not exclusively. The aim of this document is not to provide a detailed description of particular calculations, but to provide an overall description of the existing tools that are used frequently on such duty. This document is distributed as follows. In section 2 we provide the framework philosophy from a conceptual perspective, the contextualization of the environment where REST was born and the scope of the project itself. In section 3 we give a broad description of the main framework infrastructure, the basic concepts and/or elements that shape its behavior, common analysis and visualization tools, and job management. In section 4 we introduce the most common libraries that implement dedicated algorithms for specific tasks in the aforementioned duties.

2. REST conceptual design and scope

REST-for-Physics defines common data structures for event-based data processing. As it will be seen later in the general description (in section 3), this entails a prototyping of the event data holder, the processes that transform or operate those data holders, and the description of the metadata information giving a meaning to the data being processed: initial data taking conditions, input processing parameters, output results written to disk in the form of metadata, etc. The prototypes of event data, processes and metadata are complemented with basic analysis tools that are frequently used on event-based data analysis. Another important structure, named tree, is used to gather relevant event information during the data processing. This analysis summary tree contains a set of variables defined during the event data processing to be used in subsequent, higher level analysis.

REST-for-Physics defines a framework, or code development space, that centralizes event processing and analysis routines. These routines contributed by the same experts that work on the analysis of the data. The REST community keeps a strong link between algorithm design and the framework design, since there is an implicit connection between the algorithm development, analysis interpretation, and framework design requirements. REST provides already existing processes that can be used directly to define a given event processing task. REST has been designed to provide the means to be extended with new processes, metadata or event data types.

The development of REST emerges in a strong academical environment. In such context, REST intends to provide the means for academic works to materialize in the form of a piece of code that can be re-used within an already consolidated software infrastructure. A major goal for REST is to make it more accessible to non-computing experts that have a high level for algorithm coding abstraction and comprehension of the physics context.

REST-for-Physics does not replace nor does it compete with other dedicated simulation packages which provide high accuracy physics description on dedicated problems; it seeks to integrate those packages, such as Geant4 [21] or Garfield++ [22], and exploit them inside the framework as needed on the processing of the event data. In addition, REST-for-Physics includes dedicated libraries (described in section 4) that implement specialized algorithms for signal processing or physical track reconstruction. REST has its own algorithms for known mathematical problems, e.g.

---

[1] Along the text we may refer to REST-for-Physics as simply REST. The REST-for-Physics naming is preferred to avoid naming conflict with other unrelated but popular software packages.

time signal processing, to have full control over those and adapt them to our experimental needs, while still linking to consolidated libraries when possible, as it is for example the case for high-precision numbers implementation at the mpfr library [23] or the use of graph theory methods [24,25].

3. General description

REST-for-Physics is composed of a set of libraries written in C++ and it is fully integrated with ROOT [26–28], i.e. most C++ classes inherit from a ROOT TObject and therefore they can be read, accessed or written using the ROOT I/O interface. The only structural dependence is related to ROOT libraries, while other packages, as Geant4 [21] or Garfield++ [22], can be optionally integrated and used within the REST-for-Physics framework when generating or processing Monte Carlo data. Since REST-for-Physics is a natural extension of ROOT, the same naming conventions are followed: the Taligent rules [29]. On top of those standard naming conventions, any REST-for-Physics C++ class will always start with the *TRest* prefix. In this paper we will highlight the words when they clearly make a reference to an existing REST-for-Physics C++ class: a class named *TRestEvent* will be written as *event* and a class named *TRestAnalysisTree* will be written as *analysis tree*. Therefore, a highlighted word, within context, expresses a deep connection with the existing C++ classes in the project.

3.1. The REST-for-Physics framework

Inside the REST-for-Physics ecosystem a core library or framework is found. This core library prototypes and fixes the implementation of most of the REST-for-Physics C++ classes. Those base classes serve to define common methods and data members for specific[2] classes (see Fig. 1). We shall briefly introduce those basic elements:

- The *event* class encapsulates any specific *event* data inside REST. It defines common fields, such as timestamp or event id, and it prototypes common methods used for printing or drawing event information. A particular *specific event* implementation defines a type. Thus it is important to note that in what follows we will distinguish between the *event* data as the explicit contents of a particular *specific event*, and the *event* type as the format, or structure, of the *specific event*. A *specific event* representation is typically a physical quantity that needs to be described in a physical coordinate space or physical time, as it can be the time signals registered by an electronics acquisition system, or the energy deposits distribution produced by a Geant4 simulation.
- A *metadata* class may be used as a mere information container, storing relevant parameters, such as the description of the simulation conditions in *restG4* (see section 4.3). Or it might also adopt the shape of a complex object definition that implements advanced methods, such as the construction of a *detector readout*, or a *magnetic field* volume including interpolation routines. Those advanced *metadata* classes will be found in specialized libraries. Conceptually we understand by *metadata* any information required to give meaning to any *specific event* data. Therefore, any input or output parameters required during the processing or transformation of *event* data, or type,

Fig. 1. The base REST-for-Physics framework abstract classes, *metadata*, *event process* and *event* together with few examples of specific class implementations.

using *event processes*, is also regarded as *metadata*. Any metadata class can be initialized through an Extensible Markup Language (XML) configuration file.

- The *event process* class defines an input/output *event* protocol allowing to interconnect different *specific event process* implementations into a sequential processing chain. This object (as an instance of a specific class) will be able to perform operations with the input *specific event* transforming its type and/or its contents; the changes being returned in the output *specific event*. The *event process* itself inherits from the *metadata* class, since a process usually requires initialization to define input parameters that control the behavior of the process.

Other elemental tools are found inside the main framework, such as string helper methods, fundamental physics constants and units system or other basic mathematical tools useful for the development of any specific *event process*. Any *specific metadata* or *specific event process* class that does not require specialization will likely be hosted inside the framework domain. In addition, the framework repository [30] centralizes other REST-for-Physics components, such as libraries or packages. Those components will be introduced in section 4.

3.2. I/O access and metadata storage

REST-for-Physics uses the ROOT I/O interface to write *event* and *metadata* objects to disk. A ROOT file generated with REST may contain any number of *specific event* and *specific metadata* objects, including any *specific event process* (being a *metadata* object itself). Those objects are stored in a unique file, together with the *run* metadata object and the *analysis tree* that are always present in any file that has been processed with REST (see Fig. 2). The *run* object registers values to identify the data file and the conditions the data were registered, such as the start time, duration, run number, etc, while the *analysis tree* collects per-event information, named observables, at any stage of the data processing. The *run* object takes also an active role when accessing the different objects stored on disk by implementing helper methods to access the data; for example, getting a list of events from the *analysis tree* fulfilling particular conditions or retrieving directly the pointer to a given event id number updating simultaneously its corresponding *analysis tree* entry.

The framework philosophy is to create *specific metadata* classes with dedicated data members to store any information crucial for the final analysis, and/or to fully determine the nature of the data being stored. All the *metadata* inherited objects gain data member reflection support, thus creating a relation between the C++ conceptual class members and the text fields used in the configuration files. Through the implementation of the *metadata* object, a dedicated configuration file format for REST has been designed, based on Extensible Markup Language (XML). This upgrade allows the reading of XML files with additional features, such as system environment variables, complex programming instructions, including *if* conditions or *loops*, evaluation of mathematical expressions or
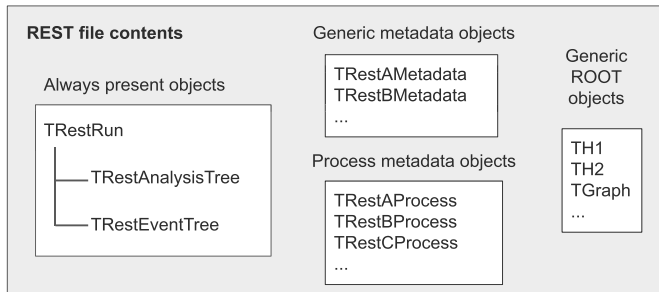
---

[2] The reader should note that when we refer to specific classes, we refer to classes which inherit, in the strict sense of C++ class inheritance, from the base abstract classes, such as *event*, *metadata* and *event process* classes. In the text, we will highlight the keyword *specific* to refer to those inherited classes in a generic way, e.g. *specific event* will be connected to any *TRestSpecificEvent* inheriting from *TRestEvent*.

Fig. 2. A schematic showing the different components that may be present inside a REST data file. The *analysis tree* and *event tree* objects are independent objects accessed through the *run* interface which ensures coherent access to a particular event entry linked to its corresponding analysis entry.

even support for arbitrary physical units conversion inside any parameter. A new extension, *rml*, is assigned to this upgraded XML format (see also the TRestMetadata class description in the API documentation [18]).

Using the *metadata* philosophy, a unique relation between the configuration files and the C++ objects is created. One XML element is identified with one *metadata* object, with its attributes associated to the data members inside the object. If there is an embedded element inside the XML element, it is associated in-chain to the corresponding *metadata* member. In this way, the automatic initialization of *metadata* objects is achieved, without any file reading methods in the class.

3.3. Event data processing and analysis

The framework allows to build an event data processing chain in a modular way by interconnecting already existing *specific event processes*, or developing new ones with the potential to plug them directly to an existing processing chain. Each *event process* has access to the input *specific event*, the *analysis tree* and any *metadata* object that is accessible by the *run* object. Depending on the input/output *specific event* interaction inside the process, we may attempt to classify the *event processes* into different groups, as illustrated in Fig. 3:

- An *external process* is a process that reads an external data source, usually at the beginning of a REST processing chain. It might be binary data generated by an acquisition system, or Monte Carlo data generated by an external simulation package. The process will be in charge of understanding the format of that external data serving to initialize a REST *specific event*.
- An *internal transformation process* is a process in which the *specific event* input is the same type as the *specific event* output. The *event* data will be transformed but not the *event* type.
- A *pure analysis process* accesses the information of a *specific event* type and produces observables that will be added to the *analysis tree* but it will not modify the *specific event* contents in any sense. A pure analysis process might serve, for example, to implement a complex physics model that uses the *specific metadata* and *specific event* information to elaborate some results that will be exported to the *analysis tree*, or a *specific metadata* object.
- A *general process* is a process that does not access the information inside the *specific event* type. It will only need access to the basic *event* information common to all *specific events*, and/or the *analysis tree*. Therefore, this process may be plugged at any point of a data processing chain without restrictions. Processes of this kind may have many different purposes, for example; to visualize online *analysis tree* observables on real time, implement a *summary* process to calculate averages (or

any other statistical variable) from the *analysis tree*, or perform a generic fitting of a variable from the *analysis tree* storing the fitting results in a dedicated metadata object, among many other basic analysis tasks.

- A *transformation process* is a process that receives as input a *specific event* and transforms it to a different *event* type. This kind of processes will all be placed at the *connectors* library, described in section 4.5, in order to encapsulate all library inter-dependencies in a single entity.

A *specific event* might be transformed during the event processing and, in that transformation, relevant information might not be available anymore in the final transformed output event. The reason is that the role of the *specific event* object is to provide a faithful or significant representation of the data at the state of processing inside the processing chain. At different processing stages, the event data might be made of time signals registered at an electronics setup, or it might be in the shape of discrete energy deposits in a physical coordinate system. Therefore, the transformation from one event data representation, or *specific event*, into another, means that a relevant parameter available at a particular stage, is not available anymore.

The *analysis tree* comes into play as an instrument to collect all those parameters extracted or calculated from the *specific event* information which will be relevant for the final analysis. Any *specific event process* in the processing chain is allowed to add new observables to the *analysis tree*. Once a process adds an observable to the *analysis tree*, this observable will always be available even if the *event* data are transformed or the processing chain happens in several steps using different input/output REST data files. The information in the *analysis tree* is always accumulative, and therefore it will contain a full summary of the observables added by each process (see Fig. 4).

In brief, the *analysis tree* provides a way for *specific event processes* to export an analysis result, extracted from each event, to the framework. It must be noted that the processes have two ways to export results, an event-per-event based observable inside the *analysis tree*, or a given result common to all the events in a particular *run*, that will be exported in the form of a *specific metadata* object.

The information extracted by a process and added to the *analysis tree* might be as simple as just a registered value available at the *specific event* at a given stage of the processing chain, or it might be the result of a complex calculation in the context of a physics model, including complicated input *metadata* objects or parameters. Of course, even in the case of basic observables extracted directly from the *specific event*, the user might be interested to know the evolution of such observable after an intermediate processing. In order to do that, it is possible to define the same *specific event process* at different positions in the sequential processing chain.

The *event process* class implements a method to facilitate the addition of observables to the *analysis tree*. This method allows to directly create or set the value for an observable from any C++ variable[3] (supporting the most common C++ types, from base types to proper *stl* containers, and either global or local variables). This method simplifies the coding of REST *event processes* by avoiding users to directly interface the branch and tree ROOT objects, and at the same time it is used to encapsulate common naming conventions for observable names, or other analysis REST standard definitions.

---

[3] In principle custom data types can be implemented inside the *analysis tree*, as this feature is supported in a standard ROOT tree. However, restricting ourselves to the use of standard C++ types is convenient to avoid additional dependencies and facilitate exporting the tree data outside of the framework domains.
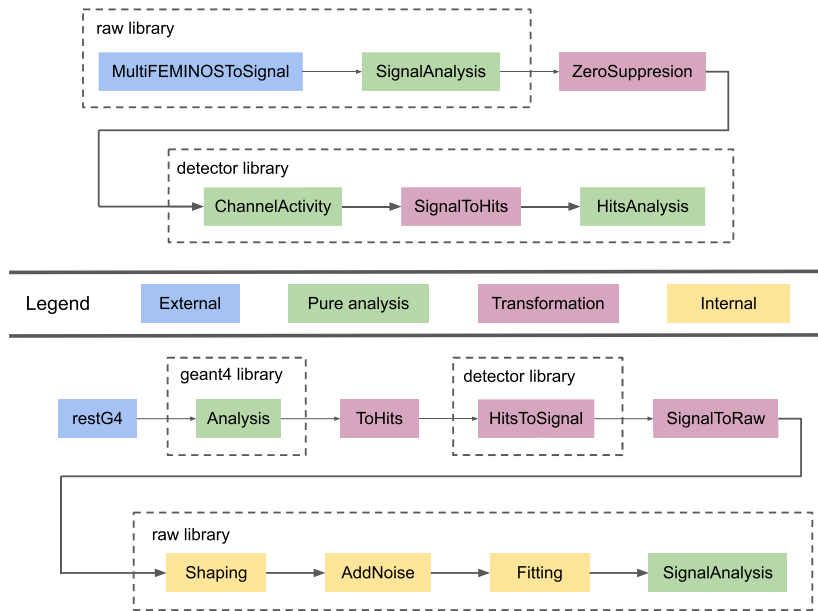
Fig. 3. A schematic showing the event data flow for two different data chain implementations in order to illustrate the different processes classification (using a color legend). On the *top*, an experimental detector data processing chain reading a binary file, analyzing, and post-processing the rawdata for event reconstruction. On the *bottom*, a Monte Carlo generated data processing chain, where the data are analyzed and transformed to match the data format in a raw electronics acquisition system, and subsequently the data are conditioned using *shaping*, *add noise* and *fitting* internal processes belonging to the raw library. The schematic shows how different libraries (geant4, detector, raw, described later on section 4) intervene at different stages, and how those play a role in both, Monte Carlo and experimental data processing. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)
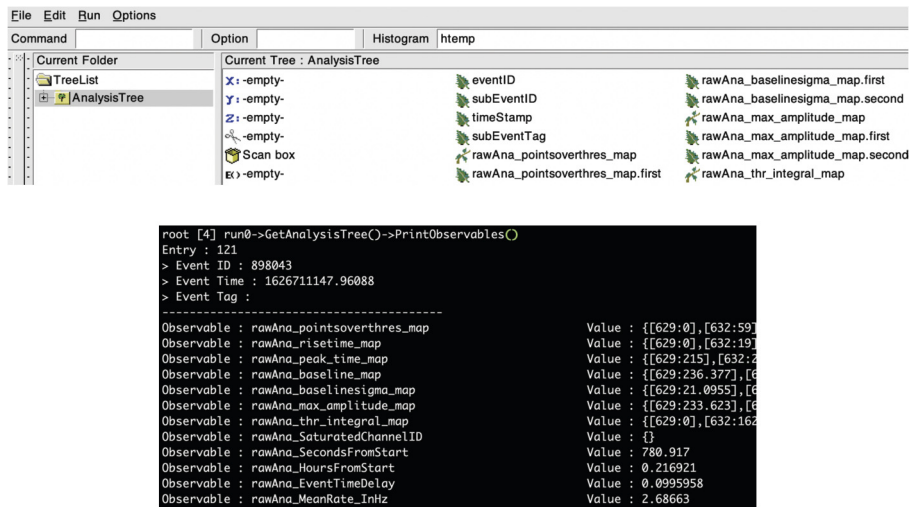

Fig. 4. Two snapshots showing the observables registered at the analysis tree. At the *top*, the observables are inspected using a ROOT browser object. At the *bottom*, the analysis observables are inspected at a particular event entry using the ROOT command line interface.

Our framework design is completely adapted to the processing of experimental data, or Monte Carlo simulated data. The reason is that a *specific event process* implementation may operate on both scenarios. The only requirement is that the experimental or simulated input *event* must be given to the process in the form of a *specific event* type. If both, simulation or experimental data, are conditioned to fit in a common *specific event* type, it will be possible to build a processing chain that not only processes simulated data or experimental data, but that fully combines both. For example, one could integrate a process simulating the signal shaping of electronics into experimental data to assess the benefit of applying such electronics setup in our experiment. Furthermore, a proper conditioning of the generated Monte Carlo *event* data will allow the evaluation of the algorithms for analysis to be used with the experimental data even before the start of the physics data taking program.

Event processes are executed through an efficient engine, or *process runner*, with multi-thread support. The data processing chain is cloned into multiple instances and kept in different threads respectively. During execution, the input event is in turn dispatched to each thread for processing, while the output event is redirected to the global output file for writing, leading to an increase of processing speed. Fig. 5 summarizes the input/output processing logic and the different concepts already described in this section.

3.4. Visualization and plotting

REST-for-Physics implements routines for event visualization and observable plotting based on ROOT drawing classes and methods. ROOT graphical interface classes are used to create basic tools, such as an *event browser* with a control panel and a drawing pad
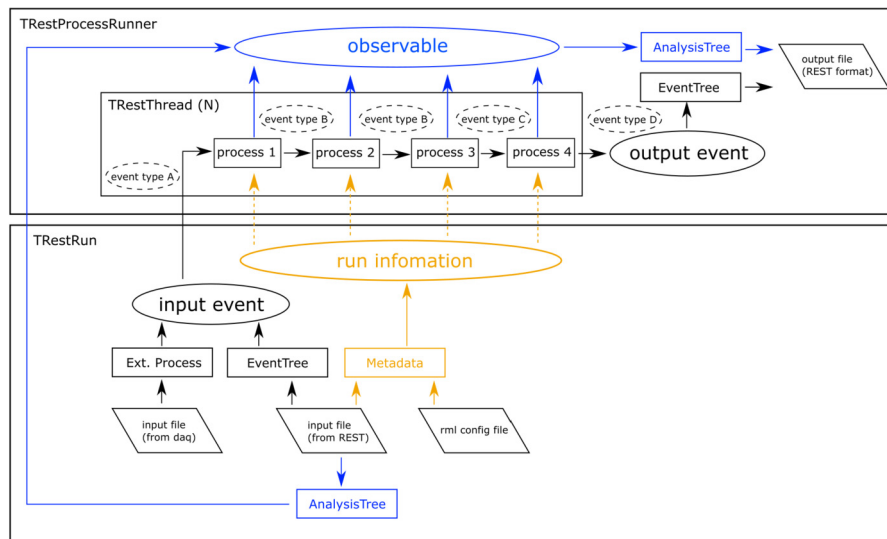
Fig. 5. A schematic diagram showing the event data flow inside a REST processing chain. The *run* object is initialized, and it has access to any *specific metadata* or *event* data available at the input REST file, or any additional objects described through *rml*. The data are then processed using the implementation inside the *process runner* object. Different event types (A,B,C,D) make reference to different *specific event* implementations. The resulting output REST file will contain all the *metadata* information available to the chain, including any previously available, together with the transformed output *specific event*, and the updated *analysis tree*.
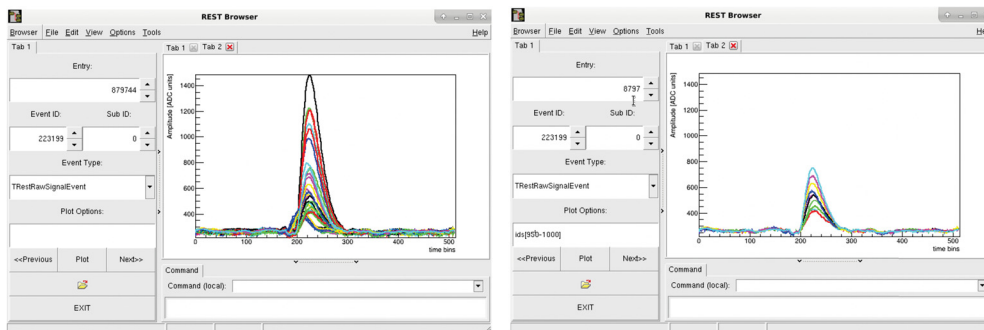

Fig. 6. Two snapshots from the REST *event browser* where the control panel and the drawing pad showing an event entry for a *raw signal event* is shown (see section 4.2). On the left figure the complete event is presented, while on the right figure the pulses have been filtered using an option passed to *draw event* method, which is implemented at any *specific event*.

(see Fig. 6). The drawing pad itself is the target of the *draw event* method implemented at each *specific event*. If enabled, different output *specific event* trees - from different stages in the data processing - will be stored in the same file. In that case the *event browser* will be able to switch between the different event data representations.

The *analysis tree* class inherits directly from the ROOT tree class, and therefore one may exploit all the resources provided by ROOT when analyzing the observables that have been added to the *analysis tree* by the different *specific event processes* at the processing chain: i.e. one may use a ROOT browser to explore the REST data files, and quickly draw and inspect variables from the *analysis tree* (as shown previously in Fig. 4).

Furthermore, REST implements dedicated tools for automatic and systematic plot generation, such as the *analysis plot* or the *metadata plot* classes. The *analysis plot* will efficiently integrate the capability to merge thousands of files through an *rml* file in which the desired plots will be assembled using the combined datasets. An *analysis plot* object allows the creation of systematic plot definitions that can be used, for example, to produce quick analysis reports in a Portable Document Format (PDF), as the one produced in Fig. 7, or to export histogram data in any other file format supported by ROOT. A *metadata plot* object allows to read many REST generated files and draw any *specific metadata* member as a function of another *specific metadata* member extracted from each of

the REST files provided. This enables the study of the correlation between any two metadata parameters, or the evolution of a metadata parameter as a function of the *run* time, or the associated run number, for example.

3.5. Execution and job management

Two executables are provided at the top level of the REST-for-Physics framework and are always available to any REST user, *restRoot* and *restManager*: the former provides a ROOT interactive prompt with REST libraries loaded, and optionally, with all the available REST macros preloaded; *restManager* manages the execution of jobs. It may launch a processing chain defined through the *process runner*, execute a method defined by any REST object available to the *run* object or launch a ROOT C++ macro file.

ROOT C-macros can be used to execute very specific but common tasks accessing the information inside REST data files. Official REST macros distributed with the framework may have been assigned an alias to facilitate its execution at the command line. Packages, or applications, that link to REST libraries will also provide their own executables, such as *restG4* or *restFileIndexer* (see Fig. 8). *restManager* allows the definition of all those actions through a configurable *rml* file. The *manager* class, executed through the *restManager* executable, guarantees that the event
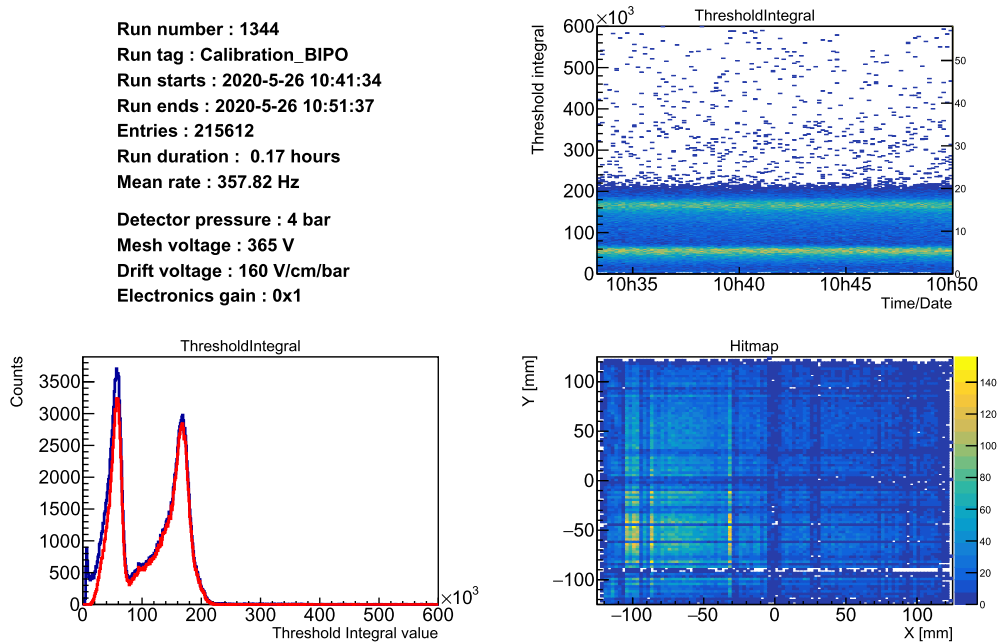
Fig. 7. A summary report produced by the quick analysis system integrated at TREX-DM using REST. The plots are generated using *analysis tree* observables. A panel with *run* details and other *specific metadata* information (top-left), an energy spectrum (bottom-left), an energy spectra evolution along the run duration (top-right) and a distribution of the mean positions, or hitmap, where the event interactions took place (bottom-right) are shown. The *ThresholdIntegral* is an observable produced by the raw library that represents the detected energy.
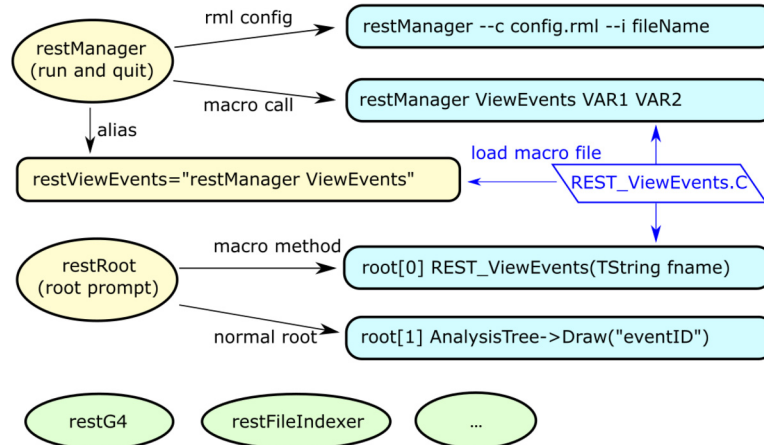

Fig. 8. REST executables running logic. *restManager* and *restRoot* work together to provide full access to the REST framework functionalities. Pre-defined ROOT C++ macro files are accessible through different interfaces, as it is shown through the *REST_ViewEvents.C* macro. Applications based on REST framework (green bubbles) extend the scope of the framework by providing additional functionalities, such as *restG4* or *restFileIndexer*.

data processing flow follows the standards previously described in Fig. 5.

A bash script, *rest-config*, is generated at each project compilation to provide information on the configuration of a particular build and to facilitate the linking of REST with external applications. It is important to remark that once REST has been compiled with a particular version of ROOT, Geant4 or Garfield++, that compilation of REST must only be used with those versions. The shell script *thisREST.sh* will be responsible to load the ROOT, Geant4, Garfield++, or any other packages required, so that they match the correct versions used to compile REST at runtime.

3.6. Project structure, versioning and code validation

The main framework defines the basic functions, and describes the behavior of the main elements of REST. As previously mentioned, it also serves to centralize all the REST-for-Physics com-

ponents, such as packages or libraries, and eventually dedicated projects. We have adopted a *git submodule*[4] strategy to integrate those components in a modular way inside the main framework repository. This scheme allows to independently monitor the development activity at each of those components, to isolate technical issues, and to focus on their functionality. Each component evolves independently with its own version or tracking system. A particular state of the code at each of those components is fixed at the main framework through a *git commit* hash, or a unique number. When that happens, the corresponding *git commit* becomes the official component version of REST.

---

[4] From this point we introduce a few concepts connected with the code versioning system, *git*, that are broadly available online, such as *commit* or *submodule*. When we refer to those alien concepts we will highlight them using the *git* keyword followed by the specific concept name.
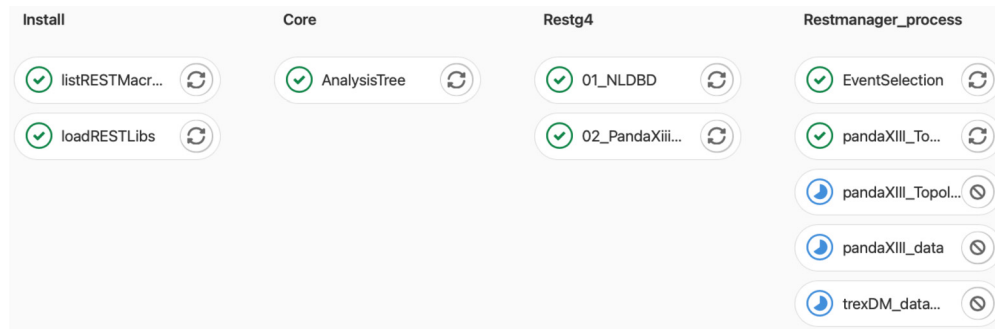
Fig. 9. A snapshot from a validation pipeline at *gitlab.cern.ch* running different tests triggered by an update to the code at the main framework repository. Different validation stages are shown, from the most basic tests on the left, including compilation and installation to complex data chain processing tests on the right.

The framework repository fully centralizes the versioning system of REST, understood as the state of the code at a given period of time, including the state of the official *git submodules* attached to it. Any REST *metadata* object written to disk using the ROOT I/O scheme will be stamped with metadata values (e.g. the REST release number, latest commit hash, release date, etc) that ensure that the data written to disk has been processed with a given version, or state of the code.[5] In order to certify that, two of those metadata members will be initialized at the code compilation time. The first metadata member will guarantee the source code was built from a clean, unmodified state with respect to the *git remote* repository, and the second metadata member will certify that the corresponding framework code state is associated with an official *git tag* release,[6] where each *git tag* generated at the main framework repository will automatically produce a code release referenced and citable at the Zenodo system [1].

On top of that versioning strategy, it is important to mention that REST properly implements the ROOT schema evolution and ensures backwards compatibility for objects that have suffered changes in their data members.

To ensure the code quality and stability with time, each repository integrates a validation pipeline where basic tests on the code are performed: some examples are code formatting and style validation, testing the proper libraries integration and building of executable programs or, even more important, testing basic results from complex data processing chains (see Fig. 9). Each modification to the code, or *git commit*, will be verified by running those validation pipelines. If a modification to the code produces an unexpected value on a consolidated data processing routine, the contributor will be notified, and changes will only become official after peer reviewing the code. This fact is extremely relevant to guarantee that the algorithms keep producing the expected results, or in the undesired case of a bug code identification, promptly identify the affected routines after its correction. Moreover, validation pipelines might serve as running examples to show the integration or use of a specific tool or element operating inside the framework.

4. REST-for-Physics libraries

The main framework contains common tools required for centralized data access, visualization, and basic analysis routines, including generic REST-for-Physics *metadata* classes and *processes* that do not require *event* specialization, i.e. they only need to access information at the *analysis tree* level. More specialized routines, requiring a dedicated *event* data type, such as time signal processing or detector event reconstruction, are organized into libraries; all classes belonging to the library keep a closer relation and therefore enhanced connectivity.

A library is usually associated only with one or two *specific event* types, increasing the connectivity between different *specific event processes* inside the same library. In this way, any combination of processes belonging to a particular library can be connected inside a data processing chain within its library domain. A dedicated library, the *connectors* library, hosts those *specific event processes* or *specific metadata* objects that need to interconnect different libraries, keeping all inter-library dependencies bound together into a single entity and allowing each library to be fully operational in stand-alone mode.

A class belonging to a particular library will have its library name as a prefix at the class name. Therefore, the *TRest* naming convention is extended in the case of the libraries to *TRestLibName*, enabling the prompt identification of the library an object belongs to.[7]

Even though new libraries might be added in the future to the framework, this section briefly describes those fundamental libraries that gave REST-for-Physics enough functionality and versatility to be used in different aspects of rare event searches experiments.

4.1. The detector library

The detector library [31] has been designed to be used for event reconstruction inside a Time Projection Chamber (TPC) filled with a gaseous medium.[8] This library contains metadata class definitions that allow to describe the detector configuration: these can be *drift volume* description, the *detector readout* topology, the particular *gas* properties (extracted using the Magboltz interface implemented by Garfield++) or others. It also integrates processes implementing routines for event reconstruction from real detector data and/or emulation of different physical response effects, e.g. including *electron diffusion*, or artificially introducing the detector energy resolution by means of a *smearing* process.

The *readout* construction (see Fig. 10) is a crucial element of the detector library. This element permits the definition of an arbitrary number of *readout planes*, containing an arbitrary num-

---

[5] It is important to note that any new *metadata* inheriting class, created in an existing or a new library, will contain these official version members. Therefore, a new REST library must integrate *metadata* classes that define the relevant information.

[6] Therefore, to become official, a new library needs to be integrated as a *git submodule* inside the main repository. Otherwise, the metadata member stating that it is an official release will not be activated.

[7] In this context, we will continue highlighting the words that make reference to C++ objects using that pattern, such as *TRestDetectorReadout* being written as *detector readout*, or even omitting the library keyword, writing, for example, *TRestDetectorGas* simply as *gas*.

[8] The currrent version of REST-for-Physics has only been exploited with gaseous TPCs. However, a liquid TPC or even other detector technologies will probably share common detection elements, like the generic *detector readout* implementation, or several detector physics processes.
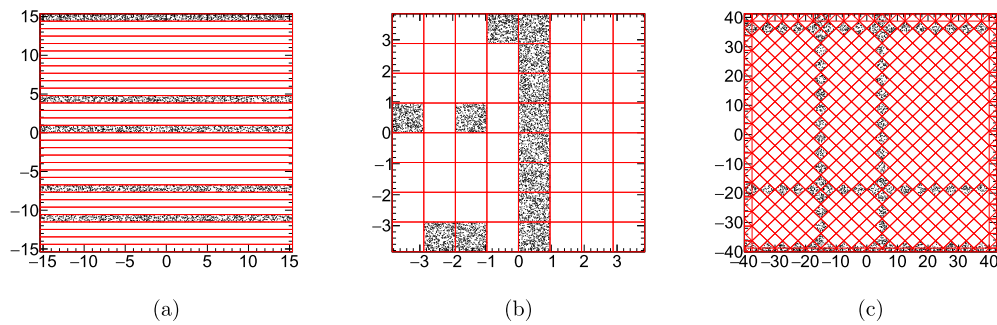
Fig. 10. Basic readout topologies that can be found at the *basic-readouts* repository [32]. (a) A stripped *readout channel* layout. (b) A pixel layout. (c) A more complex layout where each *channel* is composed of a few interconnected *readout pixels* that create a stripped pattern. The red lines represent the boundaries of the *readout pixels*, while the black dots are produced by launching a randomly spatial distribution and drawing only those points that fall inside dummy user-enabled *channels*, serving for *readout* design validation.

ber of *readout modules*, composed of physical *readout channels* that identify unambiguously with the acquisition channels of an electronics setup. The *readout channels* are at the same time built with *readout pixels*, the most basic element of a *detector readout*. Such scheme allows to create any arbitrary and complex topology, with the capability to efficiently translate - back and forward - physical coordinates and electronic channels for readouts containing a few millions of pixels.

As any other library, the detector library provides an event type to encapsulate the detector data. Currently, and for convenience, it is the only library that defines two event types. The *detector hits event* type, and the *detector signal event* type. The *hits event* defines a physical quantity, the energy deposits at the detector physical volume, using a 3-dimensional spatial coordinate representation. The *signal event* describes the energy deposits as a function of the arrival time to the *readout plane* associated to each detector electronics channel. The *readout* implementation works as a dictionary between those two types; it is used to translate one event type into another by projecting the energy deposits into the *readout channels*, or by recovering back the physical coordinate description from the readout channels information.

This library plays a central role in the characterization of the detector data and thus naturally includes connections to REST libraries related to raw electronics data processing (section 4.2), particle physics Monte Carlo event processing (section 4.3) or physical track identification and pattern recognition routines (section 4.4). The processes responsible for such library inter-connectivity are hosted on an independent library, the connectors library (see section 4.5).

4.2. The raw library

The raw library [33] implements a *raw signal event* type that is suited to describe the time evolution of physical quantities that have been acquired with a fixed sampling rate. Inside this event type one may find an arbitrary number of *raw signals* that, in the case of TPC technology, are identified with the induced currents in the electronic channels. Each *raw signal* inside the event definition contains usually the same number of samples, a value which is fixed during the *raw signal* initialization. The data depth of the physical quantity described inside the *raw signal* is 16-bits precision, which is enough to fit the typical values of electronic acquisition systems.

This library includes processes related to signal conditioning, such as signal shaping, de-convolution, pulse fitting, de-noising, Fast Fourier Transform operations, common noise reduction and other signal manipulation routines in the time domain (see Fig. 11).

In addition, the raw library includes processes, belonging to the external process type, that allow to import into the frame-

work the binary data generated by different electronics acquisition cards used in our field, such as AGET [34] and AFTER [35] chips, or DREAM [36] electronics, among others.

4.3. The geant4 library

The geant4[9] library [37] defines a *geant4 event* type that registers the energy deposits, or hits, resulting from a Geant4 simulation. A Geant4 simulation performs the physics particle tracking including the interaction probability with the materials defined for a given detector geometry. The energy deposits are similar to those found at a *detector hits event*, although the *geant4 event* hits contain additional information, like the physical interaction process, the geometrical volume where the interaction took place or the remaining available kinetic energy of the particle that produced the energy deposit. The energy deposits are encapsulated into *geant4 tracks* that describe properties common to a particular group of hits, such as the particle name producing the energy deposits, the position where the particle was originated, the track and parent ids, and in general, any relevant information directly extracted from the tracks produced by the Geant4 simulation package.

It is important to mention that this library is not directly linked to the official Geant4 libraries. Its purpose is to store the event information generated by a Geant4 simulation, but once a simulation package has registered the information inside the *geant4 event* data holder, the connection to Geant4 libraries is not required anymore. Therefore, a user would be able to access a Monte Carlo database of previously Geant4-generated files in REST format without the need to perform a system Geant4 installation.

Inside the REST-for-Physics ecosystem we have developed an independent package, *restG4* [38], which is a particular Geant4 code implementation taking advantage of the *geant4 event* type and all the definitions available at the library to describe the simulation conditions. For example, the *geant4 metadata* class defines the number of primaries to be generated, together with their energy and angular distributions, or the generator type, in order to determine how the primaries will be launched or initialized. There are many other options that allow to produce datasets in different experimental conditions and apply specific storage instructions. The library implements another relevant metadata object, the *geant4 physics list*, in which the particle physics processes to be considered in the simulation can be customized. *restG4* will register those metadata structures and the *geant4 event* tree, together with a *run*

---

[9] We will use the lowercase version of the *geant4* word when we refer to our own REST-for-Physics code implementation, while we will use the upper-case version, Geant4, to refer to the official CERN software package [21]. As a reminder, highlighted words provide a connection with the code objects, as *geant4 event* being linked to the object *TRestGeant4Event*.
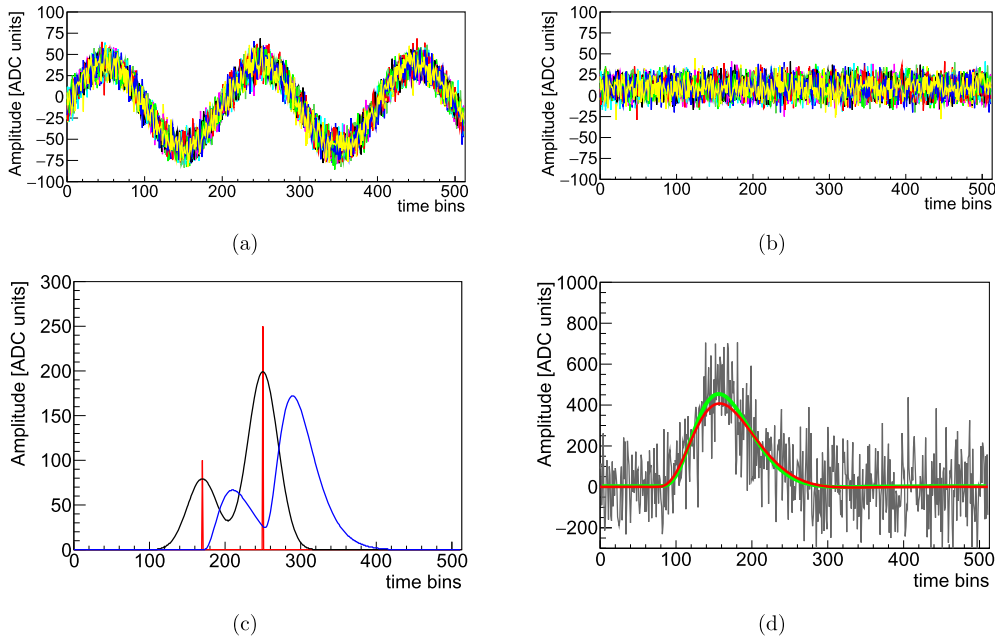
(a)

(b)

(c)

(d)

Fig. 11. (a) An artificially-generated noise-*raw signal event* with a common sinusoidal pattern. (b) The result after applying the *common noise reduction* process to the event shown in (a), where only randomly added noise remains. (c) An idealized *raw signal* composed of two point-like deposits (in red) is conditioned by the *shaping* process using a Gaussian (in black) and an AGET electronics response (in blue) convolutions. (d) An artificially-generated noise-*raw signal* (in black), together with the original pulse used to generate it (in red), and the recovered *raw signal* after applying the *fitting* process (in green).
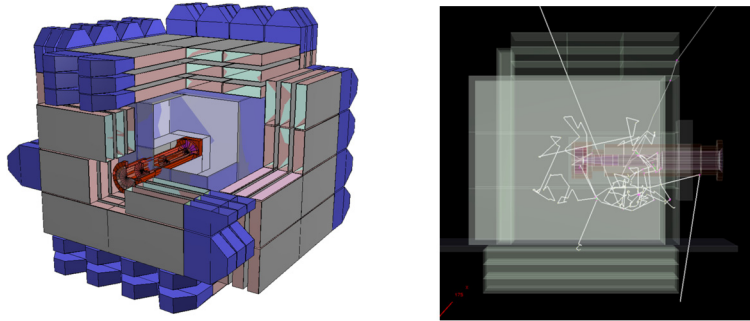

Fig. 12. *Left*, a visualization of the GDML geometry for the Baby-IAXO detector [40]. *Right*, a simulated cosmic neutron event in the same geometry visualized using the ROOT TEve viewer libraries.

metadata object complying with the REST data format conventions so that the resulting data are ready to be further processed with this or other libraries available in REST. A simulation with *restG4* requires as input the description of those three objects, the *run*, the *geant4 metadata* and the *geant4 physics list*, through an *rml* file, and a description of the geometry through a GDML [39] file (see Fig. 12).

Once a first Monte Carlo dataset has been generated using *restG4* it can be processed using the existing routines in this library. These routines, or processes, can be used to extract the Monte Carlo truth at an early processing stage. One example is the *blob analysis* process, aiming to extract the real electron track-ends in a 0νββ event; another is the *neutron tagging* process allowing to produce elaborated observables (e.g. the mean position of energy deposits found at a particular volume in the geometry) to perform a detailed analysis of the interaction of neutrons with an active cosmic veto system. Therefore, some processes at this library introduce sophisticated physics models producing results that will be exported to the *analysis tree* in the form of observables, to be accessed at a later stage of data treatment. The main idea, or philosophy, is that *restG4* is simply used to generate a first dataset, while the geant4 library will be used to introduce models that need to know about the nature of the particles or the interactions

that produced the energy deposits inside the detector geometry. Once all the relevant information has been extracted and placed in the form of observables in the *analysis tree* it can be migrated to other REST libraries (see section 4.5) in order to include a detailed detector response, condition the data to mimic raw detector data, and perform the same data processing and analysis applied to experimental data.

4.4. The track library

The track library [41] implements a *track event* type that defines inheritance relations between a set of *tracks* stored inside the event. A *track* itself contains a group of hits (or cluster) that define a discrete energy distribution in a 3-dimensional coordinate space. In order to produce or initialize a first *track event*, a process in the connectors library (section 4.5) makes use of the *detector hits event* as input to identify groups of hits, or energy deposits, that have a proximity relation, in order to create *tracks*. It is important to remark that the *track event* is an abstract object[10] that allows to

---

[10] Not to be confused with an abstract C++ class (it would have been highlighted otherwise). We want to emphasize that it is an object that does not have a strict or fixed scope.
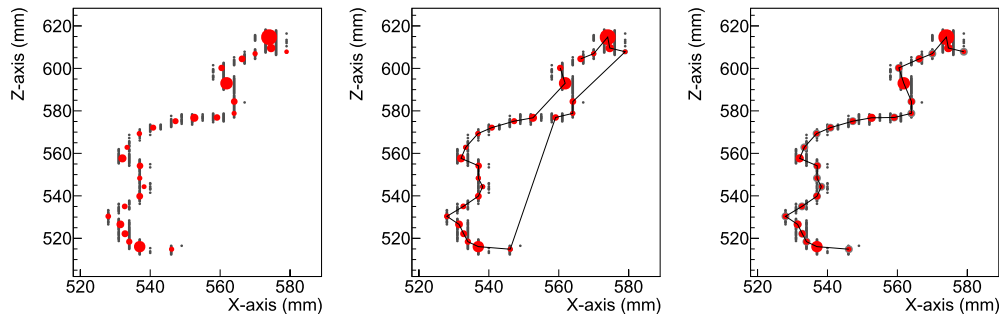
Fig. 13. A *track event* representation of a simulated $0\nu\beta\beta$ decay after the treatment with different *track* processes used for physical track identification. *Left*, an image of the hit reduction produced by the *track reduction* process. The red circles represent the final position of reduced hits, whose size is weighted with their energy value. The small grey circles on the background represent the hits of the *parent track* used as input. *Middle*, a polyline is added to this representation to visualize the hits inter-connectivity after the *track path minimization* process. If path minimization works on the whole, it produces at times obviously unphysical connections, as our example illustrates. *Right*, the unphysical connections are corrected using a *track reconnection* process. Figure extracted from reference [15].

define groups of hits, clusters, with an inheritance relation, i.e. one may develop *track* levels by generating new daughter *tracks* from the original ones. This could be exploited in different contexts: it could serve to describe isolated clusters (or group of hits) in a single physical volume, or it could serve to describe correlated *tracks* from independent physical volumes by creating a new *track* that incorporates all those mother tracks into one.

This library contains, on one hand, graph theory algorithms helping to identify and reconstruct physical tracks by finding the shortest path that interconnects energy deposits within a *track*, and on the other, processes that allow to extract topological information from a *track event*. Since graph theory algorithms are computationally expensive when dealing with a large number of nodes, a *reduction* process can be used to decrease the effective number of *hits*, so that Traveling Sales Problem (TSP) algorithms can be applied in an acceptable computation time [24,25]. TSP methods help to find a reasonable solution for the physical track identification, although further *reconnection* algorithms may be needed to improve the result (see Fig. 13). An important application of these algorithms is the identification of neutrinoless double beta decays, as it has been shown in the context of the PandaX-III experiment [15].

4.5. The connectors library

The connectors library [42] contains class definitions that need to combine the features from classes residing in different REST-for-Physics libraries. This includes processes that transform the *event* type from one library specific *event* type into another library *event* type, or it includes complex *metadata* object descriptions that require combining specific metadata descriptions from different libraries. The main mission of this library is to keep inter-library dependencies isolated or encapsulated in a single entity. In this way the fundamental libraries described in previous sections will be operative in a stand-alone mode philosophy (see Fig. 14). The REST-for-Physics building system will compile only those connectors library classes related to libraries that were marked for compilation: in the extraordinary case that only a single library was marked, then the connectors library will not be compiled at all. This library differentiation helps the coherent development of independent libraries. Using this design any library may be enabled or disabled at will, avoiding unnecessary dependencies on dedicated systems.

The main functionality of this library is to allow moving from one fundamental library domain into another, e.g. transforming a *raw signal event* into a *detector signal event* by using data reduction techniques, or grouping hits inside a *detector hits event* to produce a *track event*. However, the connectors library must not be understood as a simple *event* data type transformation, since the *specific*
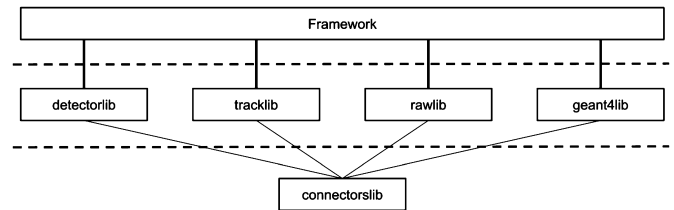

Fig. 14. REST-for-Physics libraries hierarchy and connectivity to the framework. The connectors library depends on the other fundamental libraries, providing class definitions that help inter-library communication. On the other hand, fundamental libraries with a direct connection to the framework are capable to operate in a stand-alone mode, without any other REST-for-Physics libraries requirements.

event data usually requires sophisticated routines that include the detector physics involved for the event reconstruction, data reduction inside signal processing algorithms or graph theory for the clustering of hits. This library will play a crucial role to define how different library domains inter-connect.

5. Summary

In this work we have given a broad overview of the REST-for-Physics framework and components. Our aim was to provide the reader with a general idea of the philosophy, structure and organization of the software project. And, without entering into great detail, provide an overview of the present use and functionality of REST-for-Physics.

The REST-for-Physics framework and libraries are a natural extension of ROOT, since the most basic elements inherit directly from TObject. The ROOT I/O serialization is exploited to manage the data storage while focusing on the development of physical processes that provide to REST its functionality. The motivation for this choice is the experience acquired with the ROOT framework, and the benefit of using the analysis tools it provides. ROOT was born already more than 25 years ago and it is still strongly supported and actively maintained by the CERN community which counts thousands of users. ROOT is exhaustively used in particle physics today, and its continuity in the long term seems to be guaranteed by CERN.

The REST-for-Physics framework fully exploits the schema evolution from ROOT in order to minimize the impact on data member changes in *specific event* or *metadata* objects, thus making files written with REST to be backward- and forward-compatible. One of the key aspects of the REST-for-Physics code, crucial for the storage and processing of experimental data, is its versioning strategy that it was carefully described in this paper. Such versioning strategy provides a unique relation between the code and the reg-

istered data, ensuring data and code traceability, leading to reproducible results.

One of the main motivations of the development of REST-for-Physics is to collect and centralize the software efforts and progress on detector physics for the construction of low-background detection technologies. As such, REST-for-Physics aims to serve as a platform to support future contributions in the field, consolidating common processing routines on event reconstruction, signal conditioning or pattern recognition. REST has been widely tested using gaseous TPCs, although its routines share many aspects with other detector technologies: some of the routines could be directly exploited by other technologies, while others would require minor changes to be useful for other detection setups.

CRediT authorship contribution statement

Konrad Altenmüller: Formal analysis, Investigation, Software. **Susana Cebrián:** Investigation, Resources. **Theopisti Dafni:** Investigation, Resources, Writing – review & editing. **David Díez-Ibáñez:** Formal analysis, Investigation, Software. **Javier Galán:** Formal analysis, Methodology, Software, Writing – original draft. **Javier Galindo:** Investigation, Resources. **Juan Antonio García:** Formal analysis, Methodology, Software. **Igor G. Irastorza:** Conceptualization, Funding acquisition, Writing – review & editing. **Gloria Luzón:** Funding acquisition, Methodology, Software. **Cristina Margalejo:** Formal analysis, Investigation, Software, Writing – review & editing. **Hector Mirallas:** Investigation, Project administration, Resources. **Luis Obis:** Formal analysis, Investigation, Software. **Oscar Pérez:** Formal analysis, Investigation. **Ke Han:** Supervision. **Kaixiang Ni:** Methodology, Software, Writing – original draft. **Yann Bedfer:** Supervision. **Barbara Biasuzzi:** Formal analysis, Investigation, Validation. **Esther Ferrer-Ribas:** Funding acquisition, Supervision. **Damien Neyret:** Funding acquisition, Software. **Thomas Papaevangelou:** Investigation, Supervision. **Cristian Cogollos:** Formal analysis, Investigation, Validation. **Eduardo Picatoste:** Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

References

[1] J. Galan, K. Ni, L. Obis, G. Luzon, J.A.G. Pascual, D. Diez, C. Margalejo, K. Altenmueller, I. Irastorza, K. Han, REST-for-physics/framework: https://doi.org/10.5281/zenodo.5092550, Jul. 2021.

[2] I.G. Irastorza, et al., J. Cosmol. Astropart. Phys. 1601 (2016) 033, https://doi.org/10.1088/1475-7516/2016/01/033, arXiv:1512.07926.

[3] I.G. Irastorza, et al., J. Cosmol. Astropart. Phys. 01 (2016) 034, arXiv:1512.06294, Erratum: J. Cosmol. Astropart. Phys. 05 (2016) E01, https://doi.org/10.1088/1475-7516/2016/05/E01.

[4] Iguaz Gutiérrez, Francisco José, Development of a time projection chamber prototype with micromegas technology for the search of the double beta decay of ^{136}Xe, Ph.D. thesis, Zaragoza U., 2010, https://zaguan.unizar.es/record/5731?ln=en.

[5] A. Tomas, Development of time projection chambers with MICROMEGAS for rare event searches, Ph.D. thesis, Zaragoza U., 2013, https://zaguan.unizar.es/record/12540?ln=en.

[6] L. Seguí Iglesia, Pattern Recognition in a High Pressure Time Projection Chamber prototype with a Micromegas readout for the ^{136}Xe double beta decay, Ph.D. thesis, Zaragoza U., 2013, https://zaguan.unizar.es/record/12528?ln=en.

[7] D.C.H. Muñoz, Development of a micromegas time projection chamber in Xe-based Penning mixtures for rare event searches, Ph.D. thesis, Zaragoza U., 2014, https://jinst.sissa.it/jinst/theses/2015_JINST_TH_002.jsp.

[8] J. Gracia Garza, Micromegas for the search of solar axions in CAST and low-mass WIMPs in TREX-DM, Ph.D. thesis, Zaragoza U., 2015, https://zaguan.unizar.es/record/47876?ln=en.

[9] J.A. García Pascual, Solar Axion search with Micromegas Detectors in the CAST Experiment with ^{3}He as buffer gas, Ph.D. thesis, Zaragoza U., 2015, https://zaguan.unizar.es/record/31618?ln=en.

[10] E.R. Chóliz, Ultra-low background microbulk micromegas x-ray detectors for axion searches in IAXO and babyIAXO, Ph.D. thesis, Zaragoza U., 2019, https://zaguan.unizar.es/record/87032?ln=en.

[11] V. Anastassopoulos, et al., Nat. Phys. 13 (2017) 584–590, https://doi.org/10.1038/nphys4109, arXiv:1705.02290.

[12] J. Castel, S. Cebrián, I. Coarasa, T. Dafni, J. Galán, F.J. Iguaz, I.G. Irastorza, G. Luzón, H. Mirallas, A. Ortiz de Solórzano, E. Ruiz-Chóliz, Eur. Phys. J. C 79 (9) (2019) 782, https://doi.org/10.1140/epjc/s10052-019-7282-6.

[13] X. Chen, C. Fu, J. Galan, K. Giboni, F. Giuliani, L. Gu, K. Han, X. Ji, H. Lin, J. Liu, K. Ni, H. Kusano, X. Ren, S. Wang, Y. Yang, D. Zhang, T. Zhang, L. Zhao, X. Sun, S. Hu, S. Jian, X. Li, X. Li, H. Liang, H. Zhang, M. Zhao, J. Zhou, Y. Mao, H. Qiao, S. Wang, Y. Yuan, M. Wang, A.N. Khan, N. Raper, J. Tang, W. Wang, J. Dong, C. Feng, C. Li, J. Liu, S. Liu, X. Wang, D. Zhu, J.F. Castel, S. Cebrián, T. Dafni, J.G. Garza, I.G. Irastorza, F.J. Iguaz, G. Luzón, H. Mirallas, S. Aune, E. Berthoumieux, Y. Bedfer, D. Calvet, N. d'Hose, A. Delbart, M. Diakaki, E. Ferrer-Ribas, A. Ferrero, F. Kunne, D. Neyret, T. Papaevangelou, F. Sabatié, M. Vanderbroucke, A. Tan, W. Haxton, Y. Mei, C. Kobdaj, Y.-P. Yan, Sci. China, Phys. Mech. Astron. 60 (6) (2017) 061011, https://doi.org/10.1007/s11433-017-9028-0.

[14] H. Lin, et al., J. Instrum. 13 (06) (2018) P06012, https://doi.org/10.1088/1748-0221/13/06/P06012, arXiv:1804.02863.

[15] J. Galan, et al., J. Phys. G 47 (4) (2020) 045108, https://doi.org/10.1088/1361-6471/ab4dbe, arXiv:1903.03979.

[16] E. Armengaud, et al., J. Cosmol. Astropart. Phys. 06 (2019) 047, https://doi.org/10.1088/1475-7516/2019/06/047, arXiv:1904.09155.

[17] REST-for-physics git repository, https://github.com/rest-for-physics/.

[18] REST-for-physics. API documentation, https://sultan.unizar.es/rest.

[19] REST-for-physics. A comprehensive guide to REST-for-physics, https://rest-for-physics.github.io.

[20] REST-for-physics forum, https://rest-forum.unizar.es.

[21] S. Agostinelli, et al., Nucl. Instrum. Methods A 506 (2003) 250–303, https://doi.org/10.1016/S0168-9002(03)01368-8.

[22] H. Schindler, R. Veenhof, Garfield++ — simulation of ionisation based tracking detectors, http://garfieldpp.web.cern.ch/garfieldpp, 2018.

[23] L. Fousse, G. Hanrot, V. Lefèvre, P. Pélissier, P. Zimmermann, ACM Trans. Math. Softw. 33 (2) (2007) 13-es, https://doi.org/10.1145/1236463.1236468.

[24] D.L. Applegate, R.E. Bixby, V. Chvátal, W.J. Cook, The Traveling Salesman Problem: A Computational Study, Princeton Series in Applied Mathematics, Princeton University Press, Princeton, NJ, USA, 2007.

[25] Concorde TSP library, http://www.math.uwaterloo.ca/tsp/concorde/index.html.

[26] R. Brun, F. Rademakers, Nucl. Instrum. Methods Phys. Res., Sect. A, Accel. Spectrom. Detect. Assoc. Equip. 389 (1) (1997) 81–86, https://doi.org/10.1016/S0168-9002(97)00048-X.

[27] R. Brun, P. Canal, F. Rademakers, PoS ACAT 2010 (2011) 002, https://doi.org/10.22323/1.093.0002.

[28] I. Antcheva, M. Ballintijn, B. Bellenot, M. Biskup, R. Brun, N. Buncic, P. Canal, D. Casadei, O. Couet, V. Fine, L. Franco, G. Ganis, A. Gheata, D. Maline, M. Goto, J. Iwaszkiewicz, A. Kreshuk, D. Segura, R. Maunder, M. Tadel, Comput. Phys. Commun. 180 (2011) 2499–2512, https://doi.org/10.1016/j.cpc.2009.08.005.

[29] I. Taligent, Taligent's Guide to Designing Programs: Well-Mannered Object-Oriented Design in C++., Taligent Reference Library, Taligent Press, 1994.

[30] REST-for-physics. Main GitHub framework repository, https://github.com/rest-for-physics/framework.

[31] REST-for-physics. GitHub repository of the detector library, https://github.com/rest-for-physics/detectorlib.

[32] REST-for-physics. GitHub repository including basic readouts definitions, https://github.com/rest-for-physics/basic-readouts.

[33] REST-for-physics. GitHub repository of the raw library, https://github.com/rest-for-physics/rawlib.

[34] S. Anvar, P. Baron, B. Blank, J. Chavas, E. Delagnes, F. Druillole, P. Hellmuth, L. Nalpas, J. Pedroza, J. Pibernat, E. Pollacco, A. Rebii, N. Usher, in: 2011 IEEE Nuclear Science Symposium Conference Record, 2011, pp. 745–749.

[35] P. Baron, et al., in: Topical Workshop on Electronics for Particle Physics, 2009.

[36] A. Acker, D. Attié, S. Aune, J. Ball, P. Baron, Q. Bertrand, D. Besin, T. Bey, F. Bossù, R. Boudouin, M. Boyer, G. Christiaens, P. Contrepois, M. Defurne, E. Delagnes, M. Garçon, F. Georges, J. Giraud, R. Granelli, N. Grouas, C. Lahonde-Hamdoun, T. Lerch, I. Mandjavidze, O. Meunier, Y. Moudden, S. Procureur, M. Riallot, F. Sabatié, M. Vandenbroucke, E. Virique, Nucl. Instrum. Methods A 957 (2020) 163423, https://doi.org/10.1016/j.nima.2020.163423.

[37] REST-for-physics. GitHub repository of the geant4 library, https://github.com/rest-for-physics/geant4lib.

[38] REST-for-physics. GitHub repository of the restg4 package, https://github.com/rest-for-physics/restG4.

[39] R. Chytracek, J. McCormick, W. Pokorski, G. Santin, IEEE Trans. Nucl. Sci. 53 (06) (2006) 2892, https://doi.org/10.1109/TNS.2006.881062.

[40] A. Abeln, et al., J. High Energy Phys. 137 (10 2021), https://doi.org/10.1007/JHEP05(2021)137, arXiv:2010.12076.

[41] REST-for-physics. GitHub repository of the track library, https://github.com/rest-for-physics/tracklib.

[42] REST-for-physics. GitHub repository of the connectors library, https://github.com/rest-for-physics/connectorslib.