Pyrate v2: a software system for data transformation, event reconstruction and analysis at the SABRE experiment

Federico Scutti

Swinburne University of Technology, John St, Hawthorn VIC 3122, Melbourne, AU

E-mail: `fscutti@swin.edu.au`

Abstract. The `Pyrate` system has been previously developed and recently updated by the SABRE experiment for direct dark matter detection. This framework, developed in Python, provides a dynamic, versatile, and memory-efficient approach to data format transformations, object reconstruction and data analysis in particle physics. `Pyrate` relies on a blackboard design pattern where object dependencies are dynamically evaluated throughout a run where a central control unit manages root nodes. The framework intends to improve the user experience, portability and scalability of offline software systems currently available in the particle physics community, with particular attention to medium to small-scale experiments.

1. Introduction

The `Pyrate` software system [1, 2] has been developed and is being used by the SABRE experiment for dark matter direct detection [3]. This software framework enables the transformation of data formats, reconstruction of events and data analysis at SABRE and has been developed to suit the need of small-scale experiments. The system is designed to make it easy to maintain, modular, and stable against many workflows for a small team of developers.

`Pyrate` is written entirely in Python [4], and provides all the necessary functionalities typically implemented in particle physics workflows by other larger and more complex systems [5, 6]. The software is currently hosted on the local `Bitbucket` repository of the SABRE collaboration. Experiment-specific algorithms have been implemented to transform the custom binary files of the SABRE DAQ system output into `ROOT` ntuples [7], to augment `ROOT` ntuples with higher-level event-reconstruction variables, and to achieve data analysis and plotting. The first version of the system `Pyrate v1` has been previously presented [1, 2]. This document outlines the second version of the software, `Pyrate v2`, including updates to the core class structure. The following two sections will be dedicated to the structure and essential components of the system, and then illustrate the behaviour of the software at runtime.

2. The structure of `Pyrate v2`

`Pyrate v2`, as its previous version, implements a *blackboard design pattern* [8]. Different algorithms cooperate toward the computation of an object and share data using a `Store`. The control element in the pattern, the `Run`, creates dependencies between objects at runtime which are arranged in a directed acyclic graph. The main elements of the system and their relationships are illustrated in the diagram in Figure 1. In the following, a summary of their essential functions is given.

Configuration : `Pyrate` jobs are configured using `YAML` [9] files. A primary configuration file is used to specify input objects (`Readers`), output ones (`Writers`) and intermediate objects to be handled by a run instance and how many run instances will be managed in a single job. In the configuration files, each object O_i references other objects $O_j, ..., O_n$ as their inputs and declares its output variables, which other objects can use as inputs in turn.

`Job` : The `Job` class creates and launches each run. In parsing configuration files, this class retrieves all input objects and creates run instances for each input separately. The `Job` class contains facilities for handling parallel tasks on generic batch systems, where different runs are executed in parallel.

`Run` : The `Run` class is the control element in the blackboard design pattern. It is responsible for creating an instance of the blackboard, the `Store`, and creating one or more directed acyclic graphs where each head node corresponds to an output object declared in the configuration files. In the graph, each node corresponds to an object, and edges point to their dependencies as found in the configuration. The execution order of the algorithms computing the objects does not need to be explicitly declared in the configuration files. The `Run` resolves the computation by starting from the head node of the graph and recursively calling further dependencies until all objects necessary for the computation of the depending object are found in the `Store`. Circular dependencies are naturally prevented by using a directed acyclic graph which is implemented with the `NetworkX` python package [10]. All objects are re-evaluated by the `Run` each time new events are loaded from input files by a `Reader` object corresponding to the lowest nodes in the dependency graph.

`Store` : The `Store` is the blackboard of the system. It is instantiated by the `Run` and implements the `get(object_name)` and `put(object_name)` methods to retrieve and store information. `Pyrate` algorithms call these functions during execution using a private `Store` instance. The `Store` is partitioned into two elements called the $Permanent$ and $Transient$ stores, where objects are never cleared or cleared after each event, respectively, during the execution of a program.

`Algorithm` : All objects in `Pyrate`, including input `Readers` and output `Writers`, inherit from this class. The user is responsible for implementing three methods: `initialise`, `execute` and `finalise`. The `initialise` method is used to initialise basic algorithm data structures. The `execute` method is launched after updating the current event information, allowing the algorithm to access event-based data. The `finalise` method finalises the computation of the object after the event loop is over.

`Reader` : A `Reader` is an object dedicated to reading a stream of files. If a `Reader` is used to provide event-based information, it inherits from both the `Input` and the `Algorithm` base classes, otherwise only from the latter. Different `Readers` are defined depending on different input file stream data formats. Each `Reader` inheriting from `Input` implements a `get_event()` function from its parent virtual class, which is dedicated to retrieving event information from the input file and putting it on the store. Event-based readers maintain an `event_id` identifier for the read event. The `get_event()` function is called by the `Run` for every event-loop iteration and before launching calls for evaluating objects in the directed acyclic graph. If the event can already be considered built into the input, then only one type of `Reader` is used. Otherwise, a special type of `Reader` is defined, called the `EventBuilder`, which is composed of several `Readers` managing different data streams.

`Input` : The `Input` class is a virtual one, declaring methods to manage the retrieval of event-based information from an input file. These methods are to be implemented by the derived `Reader` classes.

`Writer` : A `Writer` is an object inheriting from an `Algorithm` and dedicated to writing its output to a file. It holds an instance of the `OutputFileHandler` class managing the state of output files. Since `Writer` objects are dedicated to the computation of the highest-level output type, they are the head node of the object dependency graph.

`OutputFileHandler` : This class is dedicated to managing the state of all output files in a `Run`. It is instantiated by the `Run` and passed to all `Writers` which retrieve files via its `get_file()` method.

3. Runtime behaviour

A run collection is configured using a job configuration file, referencing object configurations. Based on the job file, a `Job` instance associates a single input to a `Run` instance which is first set up and then executed. The following two sections outline these two steps in detail.

3.1. Run setup

During the setup stage of the `Run`, instances are created for all objects declared in the configuration passed by the `Job`. In addition to this, an instance is created of the `Store`, the `OutputFileHandler` and the directed acyclic graph representing dependencies, using the `NetworkX` package [10]. Each graph node is then filled with object instances and connected to other nodes based on the input/output dependencies declared in the configuration. The head node of a graph always corresponds to `Writer` objects, while `Readers` occupy the end of the dependency chain. In addition to this, to each node, a boolean flag is added (`was_called`), indicating the execution status of an object within the current iteration, which will be updated when the three algorithmic states run and when events are read.

3.2. Run execution

A `Run` execution, qualitatively sketched in Figure 2, is performed in three stages, called *initialise*, *execute* and *finalise*, corresponding to the three methods implemented by all objects. The `Run` class also implements methods corresponding to these three states and calls them in order within the `launch()` function. Each method is responsible for evaluating the entire object dependency chain in multiple iterations where multiple evaluations are necessary during the *execute* method, the so-called `event_loop()`. Each iteration of the `event_loop()` proceeds by first calling the `get_event()` method of the `Reader`, expected to put event-based information on the `Store`, and then by evaluating the object dependency chain.

The `loop()` function evaluates the object dependency chain where head node objects are called in turn using the `call(object_name)` method. This recursive function checks the `was_called` status of the node for a given iteration. A loop over the node dependencies is performed if `was_called==False`. Dependencies are evaluated with `call(dependency_name)` executions in turn. This recursive strategy reaches the lowest level in the dependency chain, *i.e.* the `Reader`. At this point, fundamental event-based information has previously been computed by the `get_event()` function, which makes its output available on the `Store`. This allows, in turn, execution of the `Reader` dependent objects, and so forth, up to the highest head node level. After each iteration in this evaluation strategy, the `was_called` flag is reset, and a new evaluation is ready to proceed.

The recursive nature of the evaluation strategy described above allows for defining a mechanism where only a subset of dependency objects has to be evaluated under certain event conditions. For data analysis workflows, this allows one to handle event selection efficiently, where one does not need to evaluate all selection criteria downstream of a list if previous criteria fail. For those workflows not implementing a selection of events, an option is provided

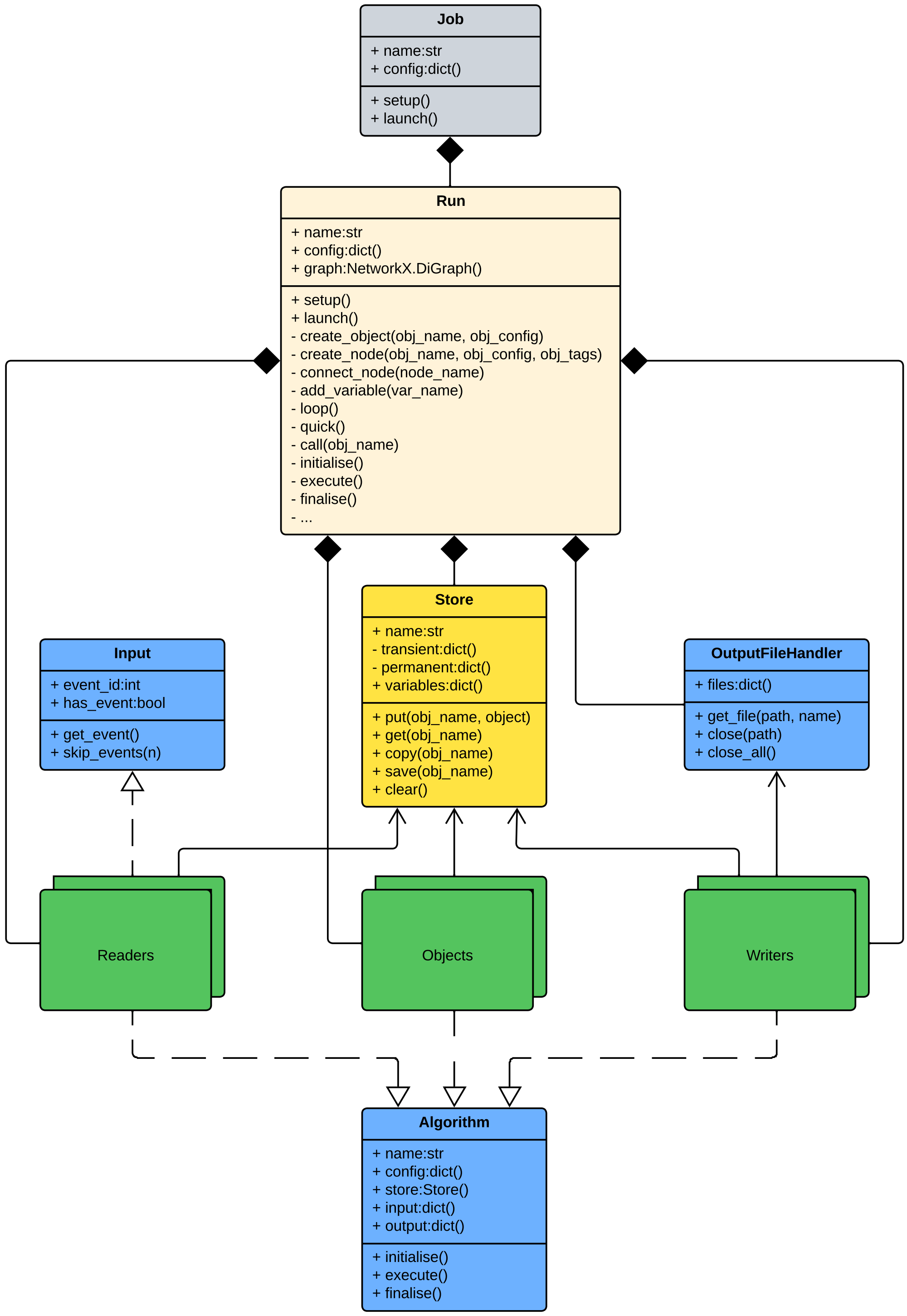

Figure 1. Class structure of the `Pyrate` core.

for instructing the `Run` to arrange objects in an ordered list and execute `call(object_name)` according to that order.

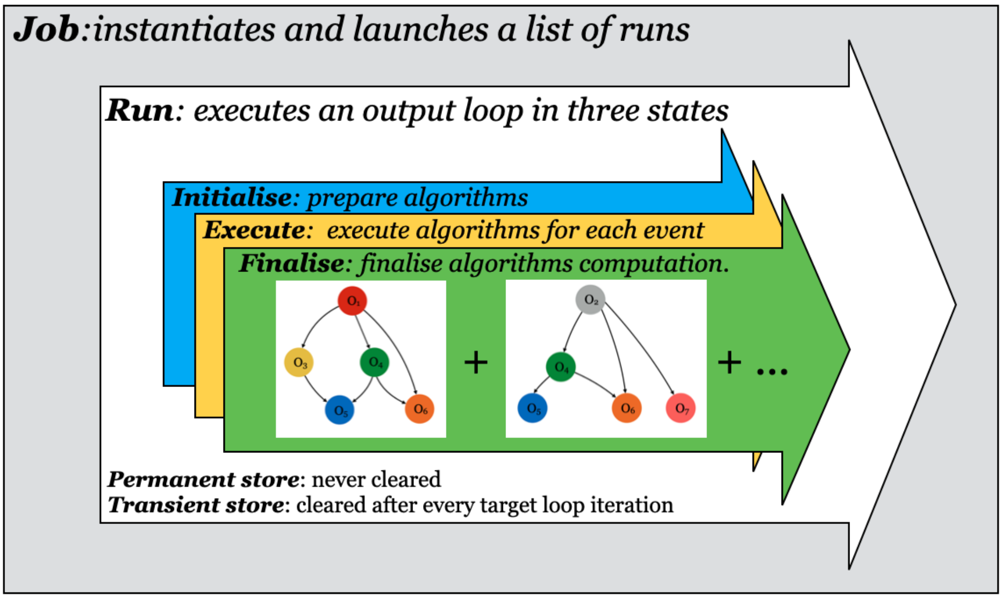

Figure 2. Qualitative representation of the different stages of the execution of `Pyrate`. Objects are arranged in a directed acyclic graph and are evaluated in the three stages: *initialise*, *execute* and *finalise*. A `Pyrate` output corresponds to the graph head node.

4. Conclusions

The `Pyrate v2` system has been described as corresponding to the second version of the software system developed and used by the SABRE collaboration for data transformations, event reconstruction and data analysis in particle physics. The `Pyrate` system currently supports custom binary input files and `ROOT` input and outputs. In the future, the addition of an extended variety of formats is foreseen, including `HDF5` [12] and `Parquet` [13].

References

[1] Scutti F 2023 *Pyrate: a novel system for data transformations, reconstruction and analysis J. Phys.: Conf. Ser.* **2438** 012061.

[2] Scutti F 2022 Pyrate *DOI:*10.5281/zenodo.6257646.

[3] Bignell L *et al* 2020 *SABRE and the Stawell Underground Physics Laboratory Dark Matter Research at the Australian National University EPJ Web Conf.* vol 232 (EDP Sciences) p 6.

[4] Van Rossum G and Drake F 2009 *Python 3 Reference Manual* (CreateSpace).

[5] Barrand G *et al* 2001 GAUDI — A software architecture and framework for building HEP data processing applications *Comp. Phys. Comm.* **140** 45-55.

[6] Zou J *et al* 2015 SNiPER: an offline software framework for noncollider physics experiments *J. Phys.: Conf. Series* **664** p 072053.

[7] Brun R and Rademakers F 1997 ROOT - An Object Oriented Data Analysis Framework *Nucl. Inst. & Meth. in Phys. Res. A* **389** 81-86.

[8] Corkill D 1991 Blackboard Systems *AI Expert* **9** 40–47.

[9] Ingerson B, Evans C and Ben-Kiki O 2001 *Yet Another Markup Language (YAML) 1.0.*

[10] Aric A *et al* Exploring network structure, dynamics, and function using NetworkX *Proceedings of the 7th Python in Science Conference (SciPy2008)* 2008 pp 11-15

[11] Gamma E, Helm R, Johnson R and Vlissides J 1994 *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley) pp 107.

[12] Koranne S 2011 *Hierarchical data format 5: HDF5* Handbook of Open Source Tools (Springer) pp 191-200.

[13] Vohra D 2016 *Apache Parquet* Practical Hadoop Ecosystem (Apress) pp 325-335.