Pyrate: a novel system for data transformations, reconstruction and analysis

Federico Scutti

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

E-mail: fscutti@swin.edu.au

Abstract. The `Pyrate` framework provides a dynamic, versatile, and memory-efficient approach to data format transformations, object reconstruction and data analysis in particle physics. The system is implemented with the Python programming language, allowing easy access to the scientific Python ecosystem and commodity big data technologies. Developed within the context of the SABRE South experiment for dark matter direct detection, `Pyrate` relies on a blackboard design pattern where algorithmic trees are dynamically generated throughout a run where root nodes are managed by a central control unit. The system guarantees an economical usage of memory allocated by algorithms where individual algorithmic instances can be reused for multiple objects. The framework is intended to improve upon 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 [1] has been developed within the SABRE South experiment for dark matter direct detection [2]. The main aim of this system is to enable the transformation of data formats, event reconstruction, and data analysis for small-scale experiments. The system is designed with particular attention to making it easy to maintain, modular, and stable against a wide variety of workflows. These features make it particularly useful for relatively small experimental collaborations, like SABRE South, where just a handful of researchers devote most of their time to software development.

`Pyrate` is written entirely in Python [3] and provides all the necessary functionalities typically implemented in usual particle physics workflows by other more complex software systems [4, 5]. The software is currently hosted on the local `Bitbucket` repository of the SABRE South collaboration. Experiment-specific algorithms are being implemented to transform the custom binary files of the SABRE South DAQ system output into `ROOT` ntuples [6], to augment `ROOT` ntuples with higher-level event-reconstruction variables, and to achieve data analysis and plotting. The following two sections will be dedicated to discussing the system's structure and its essential components and then illustrating their dynamic and interplay at runtime.

2. Structure of the system

`Pyrate` is implemented using a so-called *blackboard design pattern* [7], where different algorithms cooperate toward the computation of an object called a `Target`. They share data using a `Store` instance and create dependencies dynamically at runtime via calls to the `Store`.

Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

The main elements of the system and their relationships are illustrated in the UML diagram in Figure 1. In the following, a summary of their essential functions is given.

Configuration : `Pyrate` jobs are configured using `YAML` [8] files. A primary configuration file is used to specify the tasks or targets to be completed and to define the input, the output, and reference the location of secondary configuration files containing the definition of the objects computed by each algorithm in order to finalise a `Target` T_i. In these secondary configuration files, each object O_i declares other objects $O_j, ..., O_n$ as its immediate dependency.

`Job` : The `Job` class is responsible for checking the consistency of the configuration, e.g. the existence of the required input and algorithms, and augmenting the object dependency list to guarantee a consistent execution of algorithms across all three system states, `initialise`, `execute` and `finalise`, to be discussed later. Finally, the `Job` is responsible for instantiating and launching the `Run`.

`Run` : The `Run` class is the control element in the jargon of a blackboard design pattern. It is responsible for creating an instance of the blackboard and implementing a well-defined strategy for evaluating the `Target` objects. This strategy involves launching consecutive loops where a small number of algorithms associated with `Target` objects are called in a predefined order specified by the user. Each new loop is launched after updating the reading state of the input to read a new event or new information in general.

`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, respectively. `Pyrate` algorithms are responsible for calling these functions during execution using a private instance of the `Store`. 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 `Targets`, inherit from this base class. The user is responsible for implementing three methods called `initialise`, `execute` and `finalise`. After loading the input, the `initialise` method allows the algorithm to access general input information, e.g. metadata. The `execute` method is launched after updating the current event information in the input and allows the algorithm to access event-based data. The `finalise` method is launched where the algorithm can access the same type of input information available at initialise.

`Reader` : Each input file in `Pyrate` is handled by a separate reader. Various readers read different file formats, all inheriting from a base `Reader` class. A `Reader` holds an event index referring to the current event number in the reading file. It also contains an instance of the `Store`. `Readers` implement a `read(object_name)` function responsible for retrieving information from the file and saving it in the `Store`. `Readers` only read data from the file and put it on the `Store` when prompted by some algorithm call. No reading action is ever performed when not explicitly required by an algorithm.

`Input` : The `Input` class is responsible for handling multiple input files representing a single input type. The `Input` inherits from the `Reader` base class and implements a *factory method* [9] to instantiate multiple file `Readers` for the appropriate formats. The `Input` is responsible for guaranteeing consistent handling of the reading status of the input intended as a collection of files, for example, maintaining a global event index. In the `Run` implementation of the event loop, `Run` handles multiple `Input` instances to push forward the global event indexes of the inputs declared by the user.

`Writer` : A `Writer` is a base class analogous to the `Reader`. It implements the `write(object_name)` method dedicated to retrieving the finalised targets from the `Store`

and writing them to the output. Different writers can be defined for various file formats, inheriting from this base class. Each writer is responsible for writing to one output file.

`Output` : This class inherits from the `Writer` base class and is analogous to the `Input`. It can handle multiple output files, and eventually in different formats.

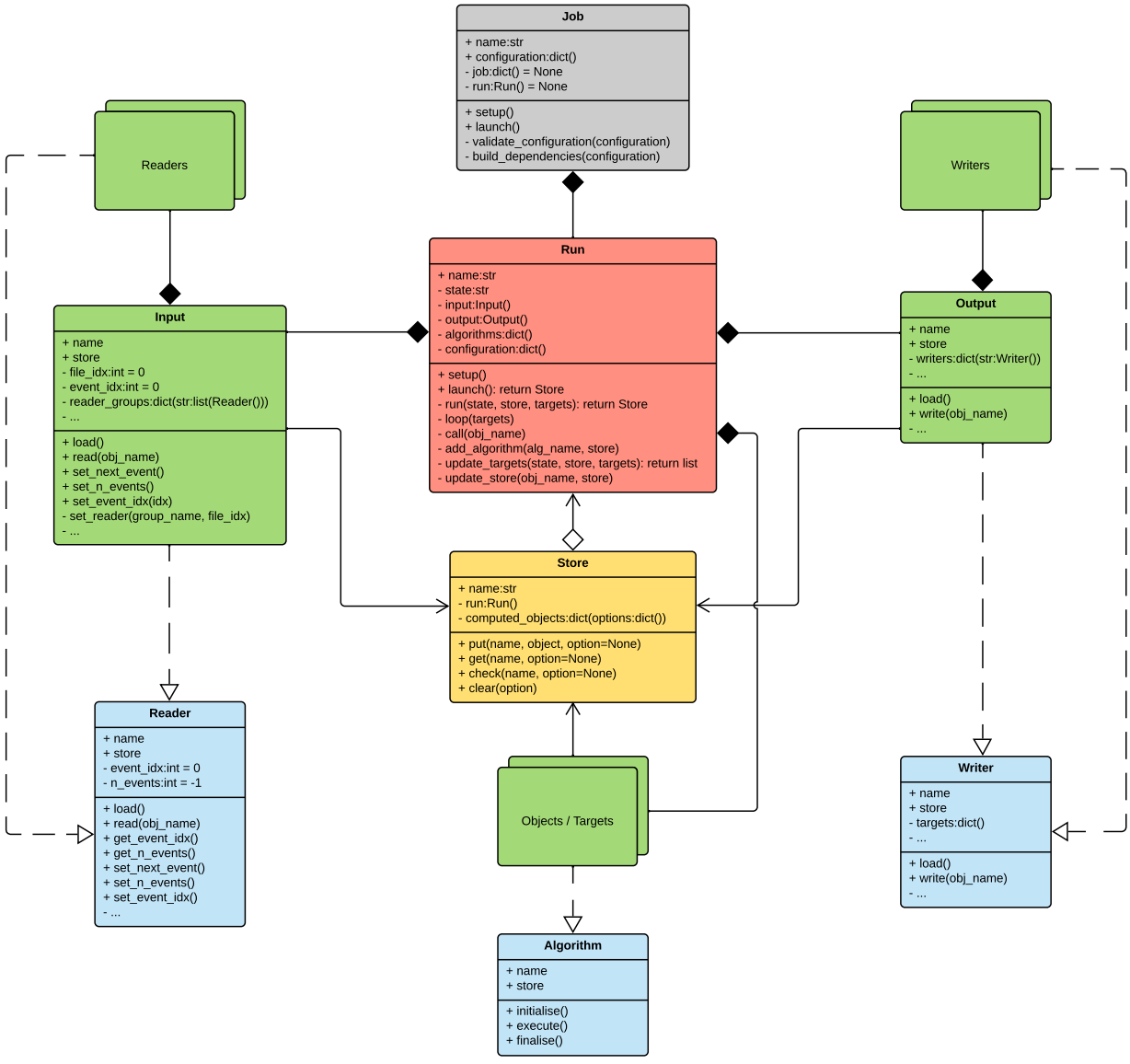

Figure 1. UML diagram structure of the core `Pyrate` codebase.

3. Execution dynamics

The dynamics of a `Pyrate` program execution is characterised by a lazy-evaluation strategy for the objects required by the targets and other objects in turn. While configuration files are required to specify only an object's direct input, the evaluation order of the complete dependency chain is dynamically determined by the system at runtime. This feature is possible by using the following strategy to evaluate objects:

(i) A `Store.get(object_name)` call is issued within an algorithm.
(ii) If the object exists, it is simply retrieved from the `Store`.
(iii) If the object is not present in the `Store`, the `Run._update_store(object_name)` function is called, which in turn applies the following strategy to evaluate the object:
  (a) Try to retrieve the object from the input by calling the `Reader.read(object_name)` function.
  (b) If the object is not found, call the appropriate algorithm dedicated to its evaluation.
  (c) If the algorithm is not already instantiated, call the `Run.add(algorithm_name)` to add the algorithm in the list of available ones and then go to the previous step.

The described dynamic allows an optimal usage of computational resources, where variables are only evaluated when needed, and at the same time, guarantees system stability, as algorithms work independently in a modular structure. The resulting dependency chain built at runtime can be visualised as a tree with the `Target` object on top, as illustrated in Figure 2. The picture also qualitatively shows how the `Run` operates toward finalising the targets. The `Run` strategy can be summarised as follows:

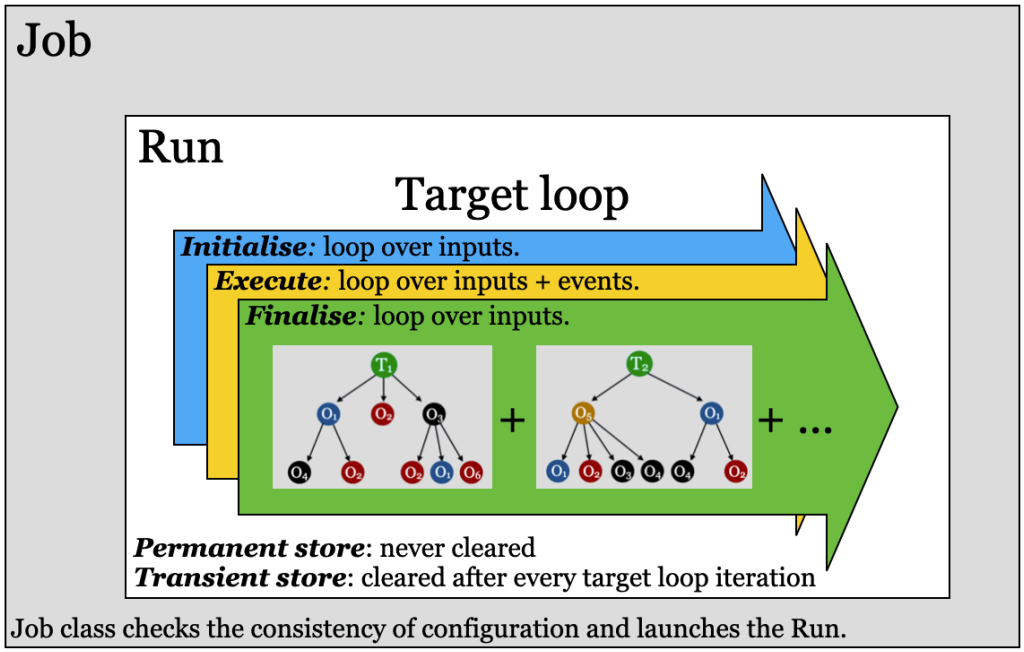

Figure 2. Qualitative representation of the different stages of a typical `Pyrate` program execution.

(i) Instantiate the `Store` object.
(ii) Load all relevant `Input` objects, where different `Readers` are allocated to a list of files and initialised with the total number of events.
(iii) Instantiate `Target` objects and put them in a list.
(iv) Launch successive evaluation calls of the `Target` objects, i.e. the so-called *target loop*, on all items in the target list in three consecutive states as follows:

Initialise : In this state, the target loop is nested within an input loop running on all relevant inputs defined by the user. This step is used to prepare relevant variables and data structures useful for successive iterations. At this stage, no event-based information is available to be retrieved by algorithms.

Execute : In this state, the target loop is nested within an event loop which is itself nested within an input loop. At this stage, algorithms can access input information at the event level. This stage performs the main computations scheduled in the program.

Finalise : This final state is used to finalise the computation of the object, which might depend on integrated information computed during the event loop. The target loop is nested within an input loop, similar to the initialising stage.

(v) Finally, write finalised objects to the output.

4. Conclusions

The novel `Pyrate` system has been described for data transformations, event reconstruction and data analysis in particle physics. The high modularity of the system and its entire Python-based structure make it a tool easy to maintain and further develop, especially for small-scale experimental collaborations. Developed within the context of the SABRE South experiment, `Pyrate` 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` [10] and `Parquet` [11], as well as supporting monitoring facilities for online data acquisition.

References

[1] Scutti F 2022 Pyrate *DOI:*10.5281/zenodo.6257646.
[2] 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.
[3] Van Rossum G and Drake F 2009 *Python 3 Reference Manual* (CreateSpace).
[4] Barrand G *et al* 2001 GAUDI — A software architecture and framework for building HEP data processing applications *Comp. Phys. Comm.* **140** 45-55.
[5] Zou J *et al* 2015 SNiPER: an offline software framework for noncollider physics experiments *J. Phys.: Conf. Series* **664** p 072053.
[6] Brun R and Rademakers F 1997 ROOT - An Object Oriented Data Analysis Framework *Nucl. Inst. & Meth. in Phys. Res. A* **389** 81-86.
[7] Corkill D 1991 Blackboard Systems *AI Expert* **9** 40–47.
[8] Ingerson B, Evans C and Ben-Kiki O 2001 *Yet Another Markup Language (YAML) 1.0.*
[9] Gamma E, Helm R, Johnson R and Vlissides J 1994 *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison Wesley) pp 107.
[10] Koranne S 2011 *Hierarchical data format 5: HDF5* Handbook of Open Source Tools (Springer) pp 191-200.
[11] Vohra D 2016 *Apache Parquet* Practical Hadoop Ecosystem (Apress) pp 325-335.