Yadage and Packtivity – analysis preservation using parametrized workflows

Kyle Cranmer[1] **and Lukas Heinrich**[1]

[1] Department of Physics, New York University, New York, USA

E-mail: lukas.heinrich@cern.ch

Abstract. Preserving data analyses produced by the collaborations at LHC in a parametrized fashion is crucial in order to maintain reproducibility and re-usability. We argue for a declarative description in terms of individual processing steps – "packtivities" – linked through a dynamic directed acyclic graph (DAG) and present an initial set of JSON schemas for such a description and an implementation – "yadage" – capable of executing workflows of analysis preserved via Linux containers.

1. Introduction

Data analyses of LHC data consist of workflows that utilize a diverse set of software tools to produce physics results. The tools range from large software frameworks like Gaudi[1] to single-purpose scripts written by individual analyzers or analysis teams. The analysis steps that lead to a particular physics result are often not reproducible without significant assistance from the original authors. This severely limits the capability to re-execute the original analysis or to re-use its analysis procedures in new contexts. An important application for such re-use is the systematic re-interpretation of a given analysis with respect to alternative models of new physics[2]. Therefore, it is desirable to have a system to archive analysis code as well as the analysis procedure in a manner, that enables both re-execution and re-use. This document presents work on workflow capture that addresses these issues in a platform and language-agnostic manner.

1.1. Short anatomy of analysis workflows

The driving paradigm of LHC analyses is the selection of events within the experiments' dataset and, typically, comparing those events to expectations derived using both data-driven techniques and Monte-Carlo simulations. Since every collision event (whether real or simulated) is independent of the others, the data analysis problem becomes *embarrassingly parallel*. Consequently, the most common task in a LHC analysis is the parallel processing of events by algorithms that transform the event data into higher-level representations (e.g. from raw detector data to reconstructed 'analysis objects') or perform event selection or otherwise reduce the dataset size, for example by selectively storing only partial event information ('thinning').

The main reconstruction transformations are often handled either on a collaboration-wide or physics working group level and use centrally managed and documented code with fixed release schedules and procedures. Transform configurations, such as the used executable and

its command line options, are managed centrally as well (e.g. with databases such as AMI[3]). Therefore, these operations are comparatively easy to preserve and reproduce.

On the other hand, custom code developed by the end-user analysis team is often much harder to reproduce due to the diversity of tools, workflows and computing environments that are used by an individual analysis team. In the case of the ATLAS experiment, a very wide spectrum of analysis frameworks have been used during Run-1 to analyze events in the "post-AOD" stage, i.e. after central reconstruction. This ranged from large and complex frameworks such as `SFrame` or `Athena`, handling not only the main event-loop, but also managing calibration tool instantiation and data-handling, to pure ROOT-based programs such as `TTree::MakeClass`- and `TTree::MakeSelector`-based codes. Since Run-2, ATLAS has seen a increasing level of homogenization in analysis codes, where many groups use one of two high-level analysis frameworks within which they develop the custom routines needed for the analysis at hand.

Once all data (real and simulated) is sufficiently reduced, usually a statistical analysis is performed, in which the observed data is compared to the expectations given by the physics model under study. Here, a range of statistics packages such as HistFitter and HistFactory or loosely-structured scripts, that utilize RooFit/RooStats directly, is used to extract the relevant physics results such as interval estimates on model parameters. This can include precision measurements of Standard Model observables or exclusion limits on parameters of models of physics beyond the Standard Model.

1.2. Analysis preservation for re-use

In the context of analysis preservation, the entire analysis can viewed as an abstract function that maps data and the model hypothesis to the analysis results:

$$\text{result} = f_{\text{analysis}}(\text{data}, \text{model}) \tag{1}$$

Ideally, one would like to preserve this map in a completely parametrized form, $f_{\text{analysis}}(\cdot, \cdot)$ independent of the specific data and model on which it has been applied to obtain the result at hand. Realistically, however, the analysis is tightly coupled to the recorded data it was developed against, due to various reasons such as file formats and re-processing versions. The model-dependence, on the other hand, can often times be factored out more easily, especially for analyses that search for physics phenomena beyond the Standard Model, where the Beyond the Standard Model (BSM) contribution is estimated separately from the Standard Model backgrounds. An analysis preservation approach that is designed to be model-independent would thus enable both re-interpretation and statistical combinations of multiple analyses after the initial publication.

In order for such to achieve such a parametrized preservation, two separate types of information need to be captured:

(i) a descriptions of the individual parametrized analysis steps such as event selection steps or the subsequent statistical analysis

(ii) a description of the workflow that logically links these individual steps in order to arrive at the analysis result data

In this document we introduce schemas to capture this information in flat JSON data as well as a framework to read back that information and re-execute such a preserved analysis.

2. Capturing parametrized activities

An appropriate model to capture the different steps of an analysis is the data model employed by the W3C PROV standard[4], in which the basic ingredients are *entities* and *activities* to track data provenance. Activities act on existing entities and generate new ones. In the context of an

HEP analysis, an entity is often a set of files (such as a dataset) or a data product derived from them, while an activity is most often the execution of a piece of software that takes entities (i.e. data) as input and generates new entities as outputs, e.g. by writing a new set of files. These operations can be parametrized by a few variables such that the activity appears as a function of the parameters $p_1, p_2, \cdots, p_n$ and some notion of an input state σ (this could for example be a filesystem directory), which may be modified as a side-effect of the function.

$$\text{output} = f_{\text{step}}(p_1, p_2, \cdots, p_n, \sigma), \tag{2}$$

It is useful to partition the return value of this function into a tuple of an output state after processing σ' and a separate record of human- and machine-readable *result data*, $\mathbf{r}$, that provides additional machine-readable data, possibly describing the side effects such as filesystem paths of files generated during this step. This separation allows for a convenient definition of workflows later on, as each step identifies and publishes the relevant data fragments (i.e. entities) it produces.

$$(\mathbf{r}, \sigma') = f_{\text{step}}(\mathbf{p}, \sigma), \tag{3}$$

An activity is thus an abstract interface that transforms parameters into result data while modifying an external state. As an interchange format for both the input parameters $\mathbf{p}$ and result data $\mathbf{r}$ JSON is a suitable choice.

The information required to fully capture such parametrized activities may be partitioned into three basic pieces:

process a parametrized description with which one can produce a fully-defined activity description – the ("job") – based on concrete parameters (such as a templated command line)

environment a description of the environment in which this job is to be executed. This may for example include a description of the necessary software to run the above process

publisher a description of how to extract the relevant information or data fragments subsequent to the execution of the job

.

For each of these pieces, multiple concrete implementations are possible. Irrespective of the implementation, the basic procedure for execution (given some execution backend) is shown in algorithm 1.

Input: $\mathbf{p}$, σ
Output: $\mathbf{r}$, σ'
begin
  job ← `Process(`$\mathbf{p}$`)`;
  σ' ← `Backend(`σ'*,job,environment*`)`;
  $\mathbf{r}$ ← `Publisher(`*job*,σ'`)`;
  return $\mathbf{r}$,σ'
end

Algorithm 1: Activity($\mathbf{p}$,σ)

2.1. Packtivity

To capture such 'packaged activities' – or 'packtivities' – a extensible set of JSON schemas have been developed to describe the three interfaces – process, environment, publisher – identified in the previous section. The choice is motivated by the simplicity and ubiquity of the JSON format, which makes it suitable for long-term and implementation-independent archival. Sub-schemas will generally be identified by a interface-wide 'property' but implementation specific 'property

value‘. An examples of full packtivity definitions are provided in listing 1. A number of JSON schemas are collected under the `yadage-schemas` package available at GitHub and via PyPI[5]

```
process:
  process_type: 'string-interpolated-cmd'
  cmd: 'DelphesHepMC  {delphes_card} {outputroot} {inputhepmc} && root2lhco {outputroot} {outputlhco}'
publisher:
  publisher_type: 'frompar-pub'
  outputmap:
    lhcofile: outputlhco
    rootfile: outputroot
environment:
  environment_type: 'docker-encapsulated'
  image: lukasheinrich/root-delphes
```

Listing 1: An example packtivity manifest

2.1.1. Process descriptions A form of capturing parametrized process information, that is accessible and convenient for analysis teams to produce, are templated string of multi-line scripts or single-line command lines from which concrete job manifests are formed by interpolating these strings based on input parameters provided as JSON documents. In the example shown in listing 1, the process has five replacement fields that need to be provided by the input JSON document, for example by having a top-level object with appropriately named properties.

2.1.2. Environment descriptions Describing the software environment or run-time that the job formed by the ‘process’ requires is a trade-off between completeness and convenience and could range from specifying merely a specific software release number to a full virtual machine image that includes both hardware and software virtualization. In practice, using Linux container technologies such as Docker have proven to be a useful middle-ground.

In HEP contexts, a large amount of software is installed centrally in the global, read-only filesystem CVMFS, thus it may not be feasible for all application to provide a standalone CVMFS-independent installation. However, as CVMFS is a versioned filesystem, in principle it is possible to mount it at the version that the original analysis executed against. Similarly, some applications may require additional run-time data such as secrets used for VO authentication. Generally, the number of such external dependencies should be kept at a minimum for preservation purposes.

2.1.3. Publisher descriptions The execution of the process in a given environment typically results in the modification of an external state which may, for example, be provided by mounting an external shared filesystem such as CephFS into the containers. However in order to describe multi-step analysis workflows, it is necessary to have a semantic description of what the relevant data fragments of this activity are. As laid out in the previous section, JSON is a suitable format. The processes themselves do not necessarily produce JSON data, so that it is helpful to include the notion of an external ‘publishing manifest’ into the packtivity definition, that a implementation can use in order to to derive a JSON object.

Often, the result JSON data can be derived by simple inspection of the input parameters such as in listing 1. In this case it suffices to provide a mapping from input parameters to output keys. In other scenarios the result data may only be fully formed after the process execution,

for example if the process generates a dynamic number of files that all need to be published. In this case other publishers may be defined, such as using glob patterns or regular expressions.

2.2. Reference implementation

The python-based `packtivity`[6] package provides an implementation of algorithm 1 for a number of different backends. Both synchronous and asynchronous backends such as Celery or IPython Clusters are provided for the currently defined process-types and environments. Both python language bindings as well as command line executables are provided to run packtivities.

3. Parametrized workflow model

The packaging of the individual processing activities captures a lot of the information needed to re-produce or re-use a given analysis. The command line interface or the language bindings can be used in code to execute pre-packaged activities in a suitable order. It is, however, desirable to capture this workflow logic in a similarly declarative fashion as the packtivities themselves, such that its execution may be automated.

A suitable data model for the description of workflows is the directed acyclic graph (DAG). In such a graph, nodes represent individual activities, while directed edges denote dependency relations of activities. This allows to capture non-linear workflows and enables the distribution and orchestration of analysis workflows across distributed systems.

As noted, for re-use applications, it is important that the workflows are parametrized. This may in turn introduce some parameter-dependency on not only the parameters for individual packtivities, but also the topology of the workflow graph. A number of DAG-based workflow systems such as DAGMan or the Common Workflow Language [7, 8] exist, however those tools introduce limitations that hinder the definition of workflow in those parameter-dependent scenarios. Therefore, a number an extensible workflow definition system has been developed that grants first-class status to parametrized DAGs.

3.1. Workflow stages

The central tenet of parametrized and dynamic workflows is that, instead of archiving DAG descriptions that fully fix the topology, one should rather store sufficient information into a *workflow template* T, from which it is possible construct these DAGs – *workflow instances* W – during run-time once sufficient information (such as parameter values) is available. As such, the workflow template T is made up from a set of *workflow stages* s_i and $T = \{s_1, s_2 \dots s_n\}$. A stage represents a piece logic that add nodes and edges to the instance DAG.

The time at which this operation may be applied can be dependent on the state of the instance. For example it may require that some node within the instance graph has already been processed by a backend and its result data is known. Therefore a stage is defined by two pieces of information

- a definition of dependencies which the workflow instance W must fulfill for the stage to be applicable. This is conveniently expressed as a DAG-valued *predicate function* $d : W \to \{\text{true}, \text{false}\}$
- a DAG-valued function that, given a workflow instance, returns an updated workflow instance with additional nodes and edges or newly defined stages. This may be expressed as a DAG-valued *scheduler function* $f : W \to W'$

In order to process a parametrized workflow defined by such a template, a simple algorithm, shown in algorithm 2 may be followed in order to continuously monitor a workflow instance. The algorithm applies the stages' scheduler functions as soon as its predicate is fulfilled and

submit individual nodes to a packtivity backend.

Input: Workflow Template, Initialization JSON
Output: Complete set of Workflow data entities
initialize new workflow instance (empty DAG) wflow;
while *there exist unapplied stages* **do**
  foreach *stage in unapplied stages* **do**
    if *stage applicable (predicate returns True)* **then**
      expand DAG with stage as per scheduler function
    end
  end
  foreach *node in DAG* **do**
    if *node has not been submitted to backend and all incoming edges succeeded* **then**
      submit node to backend
    end
  end
end
wait until all nodes processed by backend

Algorithm 2: Basic Yadage Workflow Engine

3.2. Yadage

3.2.1. Workflow Definition As for packtivities, the `yadage-schemas` package includes JSON schemas to define workflow stages. From experience complex workflows may be defined using a small number of stage-types. Currently stages that schedule one or more nodes of a singled packtivity (each with different parameters) are defined. In the case of multi-node stages, a number of scattering patterns maybe used. An example stage definition is shown in listing 2. The dependencies are listed by naming other stages. The current stage is then considered applicable if all nodes by the dependent stage are successfully processed. The scheduler function f, is defined under the `scheduler` property and includes instructions on how to access the result data of dependent nodes added to the Graph by dependent stages in order to define the parameters of the packtivity to be scheduled. As seen in the listing, JSON References are used in order to reference packtivity definitions via URIs.

```
name: delphes
dependencies: [pythia]
scheduler:
  scheduler_type: 'singlestep-stage'
  step: {$ref: 'delphes.yml'}
  parameters:
    outputroot: '{workdir}/output.root'
    outputlhco: '{workdir}/output.lhco'
    delphes_card: 'delphes/cards/delphes_card_ATLAS.tcl'
    inputhepmc: {stages: pythia, output: hepmcfile}
```

Listing 2: An Example Yadage Stage Manifest

3.3. Workflow composition

An important feature of workflows defined via the yadage schemas is composability that is not dependent on coordination between workflow authors. HEP workflows can involve many different stages in order to transform real or simulated events from detector or even particle-level data all the way to a final analysis result. Parts of these workflows may be primarily defined by different groups. For example, the workflow to describe the generation of Monte Carlo

events based on a certain model may be defined primarily by physics working groups, while the reconstruction chain to transform generated events into fully reconstructed events ('AOD' data) usually is the responsibility of a central reconstruction group. Finally the downstream analysis of those reconstructed events is done on the analysis-team level. Each of these macro-parts of the workflow may be a multi-stage workflow themselves, such that the ability to compose them without modification into a larger workflow is desirable. In the case of the currently defined yadage stages, this is easily achieved by modifying the stage schedule 'workflows' instead of individual packtivities. In this case, the scheduling function does not add nodes or edges, but rather adds newly defined stages corresponding to the sub-workflows to the instances list of stages, that subsequently will be applied in a scoped fashion to exclude the possibility of e.g. naming collisions/ambiguities.

3.3.1. Reference implementation Workflows declared using the above schemas may be executed using the `yadage` package[9] that implements the basic scheduling and submission algorithm outlined above and transparently is able to use any packtivity backend implementation. Further, it implements a number of convenience features such as caching/memoization of individual packtivity results and visualization capabilities.

4. Applications

4.1. Run-I reinterpretation

Workflows defined via the schemas outlined above have been used in a number of different contexts. ATLAS has published multiple re-interpretations of analyses prepared in Run-1. In these campaigns, a number of analyses designed to investigate particular supersymmetric scenarios have been re-interpreted to derive a more comprehensive assessment of the ATLAS experiments' sensitivity to supersymmetry such as under the 19-dimensional 'phenomenological MSSM' (pMSSM) or a more restricted five-dimensional scan targeting electroweak sparticle production[10, 11, 12].

4.2. CERN Analysis Preservation Portal

The CERN Analysis Preservation Portal (CAP)[13] aims to preserve analysis information in the form of a digital library. Besides assembling metadata of analyses, such as the involved researchers and institutes, it also seeks to archive more technical information, such as code repositories and software environments. The workflow and activity schemas described here have been deeply integrated into the CAP system. Thanks to the choice of JSON schemas, the can be treated as native data in the context of the Invenio Digitial Library framework[14] facilitating, for example, discoverability and composability of workflow pieces.

4.3. RECAST

As noted, LHC experiments already engage in re-interpretation campaigns. However, the current approach requires a high level of coordination between analysis groups for each re-interpretation. RECAST is a framework in which the re-interpretation is streamlined by utilizing workflows that are archived in a re-usable manner. Originally proposed in 2010, a prototype backend implementation has been recently been developed and deployed at CERN. RECAST allows interested parties outside of the LHC collaboration to suggest for re-interpretation, by providing model information such as parameter cards in the SLHA format to the experiments using a web-based interface or, alternatively, REST and Python APIs. Upon review by the experiments, the experiment may decide to re-run an archived analysis based on these new model inputs and provide a response in the form of likelihood information (e.g. CL_s values)

The prototype backend consists of a 'control-center' web-service that is accessible using VO-filtered CERN Single-Sign-On. This web-service displays incoming requests and allows operators

to launch a re-execution of an analysis based on the RECAST request. The actual workflow execution is then handed off to a distributed system. Here, yadage and packtivity are heavily used to both define workflows and drive their execution. In the course of this development, a packtivity backend has been implemented that allows the scheduling of HEP container workloads on a Kubernetes Cluster deployed on the CERN OpenStack infrastructure using OpenStack Magnum. The integration with cloud-native tools allows for convenient scaling characteristics and a highly distributed workflow execution that may be monitored in real-time using the web-interface of the 'control-center'.

5. Summary

We have presented a framework to define parametrized workflows in order to preserve high-energy physics analyses in a format that allows collaboration members to re-execute the original analysis in the context of new physics models. The framework defines a set of portable JSON schemas that describe both the individual processing steps and and workflow logic to orchestrate multiple steps. We also presented language-agnostic algorithms to re-execute analyses based on these descriptions. Furthermore, reference implementations of these algorithms distributed as python-based packages (`packtivity` and `yadage`) were presented. Workflows definitions and the reference implementation have been used in the past for re-interpretation campaigns within the ATLAS collaboration and are deeply integrated in the CERN Analysis Preservation Portal and RECAST projects.

6. Acknowledgements

Cranmer and Heinrich are both supported through NSF ACI-1450310, additionally Cranmer is supported by PHY-1505463.

References

[1] Barrand G, Belyaev I, Binko P, Cattaneo M, Chytracek R, Corti G, Frank M, Gracia G, Harvey J, Herwijnen E, Maley P, Mato P, Probst S and Ranjard F 2001 *Computer Physics Communications* **140** 45 – 55 ISSN 0010-4655 URL `http://www.sciencedirect.com/science/article/pii/S0010465501002545`

[2] Cranmer K and Yavin I 2011 *Journal of High Energy Physics* **2011** 38 ISSN 1029-8479 URL `http://dx.doi.org/10.1007/JHEP04(2011)038`

[3] Albrand S, Fulachier J and Lambert F 2010 *Journal of Physics: Conference Series* **219** 042030 URL `http://stacks.iop.org/1742-6596/219/i=4/a=042030`

[4] Moreau L, Missier P, Belhajjame K, B'Far R, Cheney J, Coppens S, Cresswell S, Gil Y, Groth P, Klyne G, Lebo T, McCusker J, Miles S, Myers J, Sahoo S and Tilmes C Prov-dm: The prov data model Tech. rep. URL `http://www.w3.org/TR/prov-dm/`

[5] Heinrich L and Cranmer K 2017 diana-hep/yadage-schemas: Initial zenodo release

[6] Heinrich L and Cranmer K 2017 diana-hep/packtivity: Initial zenodo release

[7] Thain D, Tannenbaum T and Livny M 2005 *Concurrency and Computation: Practice and Experience* **17** 323–356 ISSN 1532-0634 URL `http://dx.doi.org/10.1002/cpe.938`

[8] Amstutz P, Crusoe M R, Tijani N, Chapman B, Chilton J, Heuer M, Kartashov A, Leehr D, Mnager H, Nedeljkovich M, Scales M, Soiland-Reyes S and Stojanovic L 2016 URL `https://figshare.com/articles/Common_Workflow_Language_draft_3/3115156`

[9] Heinrich L and Cranmer K 2017 diana-hep/yadage: Initial zenodo release

[10] Aad G *et al.* (ATLAS) 2015 *JHEP* **10** 134 (*Preprint* `1508.06608`)

[11] Aaboud M *et al.* (ATLAS) 2016 *JHEP* **09** 175 (*Preprint* `1608.00872`)

[12] 2016 A re-interpretation of $\sqrt{s} = 8$ TeV ATLAS results on electroweak supersymmetry production to explore general gauge mediated models Tech. Rep. ATLAS-CONF-2016-033 CERN Geneva URL `https://cds.cern.ch/record/2198316`

[13] Chen X, Dallmeier-Tiessen S, Dani A, Dasler R, Fernández J D, Fokianos P, Herterich P and Šimko T 2016 *CERN Analysis Preservation: A Novel Digital Library Service to Enable Reusable and Reproducible Research* (Cham: Springer International Publishing) pp 347–356 ISBN 978-3-319-43997-6 URL `http://dx.doi.org/10.1007/978-3-319-43997-6_27`

[14] Kuncar J, Nielsen L H and Simko T 2014 Invenio v2.0: A Pythonic Framework for Large-Scale Digital Libraries Tech. Rep. ATLAS-CONF-2016-033 CERN Geneva URL http://urn.fi/URN:NBN:fi-fe2014070432294