ATLAS in-file metadata and multi-threaded processing

Frank Berghaus[1,*], *Attila* Krasznahorkay[2], *Tim* Martin[3], *Tadej* Novak[4], *Marcin* Nowak[5], *A.C.* Schaffer[6], *Vakho* Tsulaia[7], and *Peter* van Gemmeren[1,**,***]

[1]Argonne, Illinois, USA
[2]CERN, Geneva, Switzerland
[3]University of Warwick, Coventry, UK
[4]DESY, Hamburg, Germany
[5]BNL, New York, USA
[6]IJCLab, Université Paris-Saclay, CNRS/IN2P3, 91405, Orsay; France
[7]LBNL, Berkeley, USA

Abstract. Processing and scientific analysis of the data taken by the ATLAS experiment requires reliable information describing the event data recorded by the detector or generated in software. ATLAS event processing applications store such descriptive metadata information in the output data files along with the event information.
To better leverage the available computing resources during LHC Run3 the ATLAS experiment has migrated its data processing and analysis software to a multi-threaded framework: AthenaMT. Therefore in-file metadata must support concurrent event processing, especially around input file boundaries. The in-file metadata handling software was originally designed for serial event processing. It grew into a rather complex system over the many years of ATLAS operation. To migrate this system to the multi-threaded environment it was necessary to adopt several pragmatic solutions, mainly because of the shortage of available person-power to work on this project in early phases of the AthenaMT development.
In order to simplify the migration, first the redundant parts of the code were cleaned up wherever possible. Next the infrastructure was improved by removing reliance on constructs that are problematic during multi-threaded processing. Finally, the remaining software infrastructure was redesigned for thread safety.

ATL-SOFT-PROC-2021-005
03 June 2021

1 Introduction

Modern particle physics experiments produce a variety of data required to perform physical measurements. The elementary measurement of the ATLAS detector [1] is an event: the detector response to a collision at its center. The event may be the physical response of the detector to a collision delivered by the Large Hadron Collider or the simulated detector

*e-mail: fberghaus@anl.gov

***Argonne National Laboratory's work was supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357.

Table 1. The ATLAS in-file metadata may be separated into ten categories. Each category uses one shared object to contain information. Multiple instances of such objects may and often do exist. Each category is managed by a dedicated component.

Category	Summary description
Byte stream	Detector and data acquisition configuration
Event bookkeeping	Ledger of event selection
Event format	Relation of C++ classes with branch names in the file
Event stream	Event type summary and C++ classes with branch names
File	Summary of event information and file provenance
Interval of Validity	Information with a lifetime other than event or file
Luminosity block	Events in a period of constant experimental conditions
Trigger	Trigger configuration used to select collisions to record
Truth	Event generator and detector simulation information

response to collision generated by software. To process and analyze these events a wealth of information describing the data is required. Examples of such descriptive information are the object content available for the event recorded, the nature of the event, the state of the detector, and so on. Some of this information is stored in central databases and some as additional content in files of event data [2]. This contribution concerns that additional *in-file metadata*. Section 2 provides and overview of the information stored as in-file metadata.

The content and management of in-file metadata in the ATLAS experiment was developed organically over many years. That is, in response to the needs of the ATLAS event data model, software framework, and physics analyses. As the effort of the ATLAS collaboration to produce thread safe production software, *AthenaMT* [3, 4], resulted in mature code it became clear that the in-file metadata infrastructure required improvement. A pragmatic approach was chosen to support thread safe operation in a timely manner. The process is described in Section 3.

As the reader will note in Section 2, the same information often exists in many different in-file metadata objects. While this provides convenient access, it makes the in-file metadata system difficult to maintain. The interval of validity (*IOV*) metadata described in Section 2.1 duplicates the functionality provided by external databases. This parallel structure increases the maintenance burden. The complex interface and lack of documentation lead to frequent misuse in client code. The authors are involved in an effort to redesign the ATLAS metadata system. That redesign is however beyond the scope of this contribution.

2 Organization and content

The in-file metadata can be organized into categories by the objects containing the information and the tools and services managing those objects. The categories are summarized in Table 1. Details on each category are provided in the sections below.

For offline computing, ATLAS uses the ROOT [5] file format with ROOT-generated C++ class dictionaries describing the experiment specific objects stored in the files. The in-file metadata is stored in a dedicated `TTree` within a single entry. This in-file metadata is managed by an Athena service: the `MetaDataSvc`. This paper will refer to it as the *metadata service*. When a file is opened the metadata service fills a read-only transient object store [6], called "InputMetaDataStore", with the content of this tree. Each metadata category listed

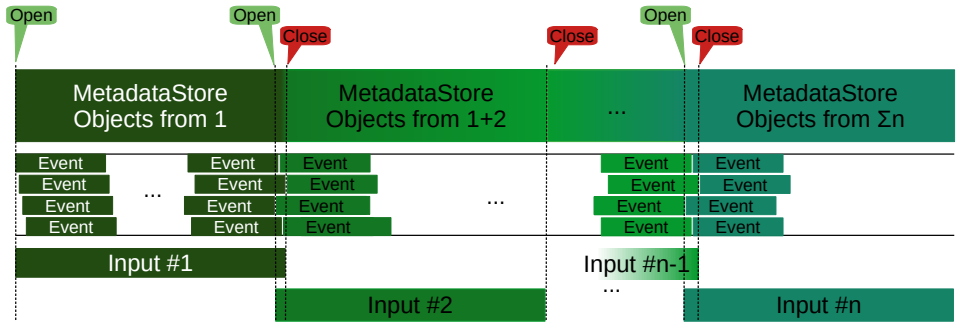

Figure 1. A multi-threaded job processes a number of events concurrently. The job reads events from many input files and writes results to one output file. The second input file is opened when no further events from the first input file remain to be processed, and so on. An input file is closed after the job finishes processing the last event from that file because ATLAS reads persistent data on demand. That means, the job processes the initial events from the new input file concurrently with events from the previous input file. Each event should access in-file metadata appropriate to it. Providing clients with the in-file metadata appropriate to the event they are processing as the job switches input files presents the central challenge for in-file metadata in AthenaMT.

in Table 1 is managed by a dedicated component[1] which implements the `IMetaDataTool` abstract interface. After the input metadata store is filled the metadata service calls the `beginInputFile` method of all metadata tools registered with the service. When all events from the first input file are processed the metadata service calls the `endInputFile` method of the registered tools before the file is closed. Then the content in the input metadata store is removed and the next input file is opened. The process is illustrated in Figure 1.

A second metadata store is required to allow metadata objects to persist throughout the job because the input metadata store is cleared on each input file transition. This second metadata store it is called the "MetaDataStore". After the input metadata store has been filled its content is either copied to the metadata store or merged with objects already present in the metadata store. This procedure is handled by a separate tool for each of the metadata categories in Table 1.

In the online environment, ATLAS uses its custom byte stream format [7]. This format does not provide a place to store in-file metadata. A well defined header provides the run parameters. In offline jobs processing this format the tools called by the metadata service create new objects from information available.

2.1 Interval of Validity

Most of the in-file metadata categories listed in Table 1 represent metadata with fixed values that are valid for a given file. The interval of validity category represents metadata with values that may change with time and are not constrained by file boundaries [8]. A time period when a given metadata object has known and unchanging attribute values defines a single IOV for this object. An object may have any number of such intervals.

IOV metadata format is used in ATLAS primarily to describe detector conditions stored in relational databases. This data is retrieved by jobs over database connections. The in-file

[1] An `AlgTool`

implementation of the IOV format permits Athena to store IOV metadata also in files. This allows workflows without access to the external database to access these objects from the file instead of the database.

The Athena framework provides transparent access to IOV metadata by maintaining a snapshot of attribute values valid in the context of a currently processed event in a service (`IOVDbSvc`). The validity of a snapshot is reevaluated for every new event by another dedicated service: (`IOVSvc`). This approach is well suited for serial event processing, but becomes problematic in the presence of multiple events processed concurrently.

2.2 Run Parameters

Data acquired from the ATLAS detector uses a custom byte stream format. This format includes a well defined set of bytes as header. This header contains information about the accelerator, detector, and data acquisition process such as: the beam energy, the run number, active sub-detectors, and so on. This provenance information is referred to as the *run parameters* and made available during offline processing in the `ByteStreamMetaData` object. Multiple byte stream metadata objects may be collected in a `ByteStreamMetaDataContainer`.

2.3 Luminosity Block

In the ATLAS experiment, a luminosity block (*LB*) is a time interval of data recording over which the experimental conditions are assumed to be constant. In particular, it is assumed that the instantaneous luminosity is constant over the duration of the luminosity block. Luminosity blocks are set by the ATLAS central trigger processor. For the event processing it is important to keep track of the number of events within a given luminosity block. This information is recorded at the end of the event processing job into the metadata of the output file as three collections of objects: *Complete LB* all events from this luminosity block have been processed; *Incomplete LB* not all events from this luminosity block have yet been processed; *Suspect LB* for these luminosity blocks more events than expected have been processed. The expected number of events per luminosity block is read by the event processing application from the ATLAS Conditions Database.

In the Athena framework there are two components which manage Luminosity Block metadata. One of these components is an *Algorithm* which reads the expected number of events per luminosity block from the conditions database at initialization. During event processing this algorithm simply counts events within lumi-blocks and, at the end of the job, writes the collected information into the output file metadata.

Another Athena component, an `AlgTool`, is simply propagating luminosity block metadata from input into the output file. If a job runs over multiple input files, then their corresponding metadata information is merged by the `AlgTool` prior to writing to the output metadata.

2.4 Trigger menu

The trigger system is an essential component of data acquisition in modern high energy physics experiments. The trigger searches for signatures that are of interest to physics groups in the collaboration and selects matches to be recorded for thorough processing and analysis. The collection of event signatures the trigger searches for is referred to as the trigger menu. In addition, for specific signatures, the rate at which a trigger accepts that signature may be reduced by randomly sampling. The probability of accepting an event in this random sampling is referred to as the *prescale*. The ATLAS trigger system is implements in two tiers [9]:

- The Level 1 trigger physically resides on the detector. It makes fast and rough decisions to reduce the event rate to something the High Level Trigger can process.
- The High Level Trigger is a data center at the surface above the detector. It selects events at a rate which can be recorded for processing and analysis and stored indefinitely.

There are many physics groups each interested in a diverse set of event signatures. In addition event signatures are added or removed as insights about where interesting physics may be found change. The selection process must be well understood as it biases the events available for analysis. The exact configuration, event signatures, and prescales of the trigger is stored in the trigger menu metadata.

The trigger menu metadata are used in processing and analysis, for example, to check if a given trigger passed the event, or perform matching between physics objects reconstructed by the trigger and in later processing.

2.5 Event stream information

The event stream is a concept of grouping incoming or outgoing events into a pipeline. Athena implements this concept using a special algorithm called `AthenaOutputStream`. The output stream sends event data and metadata to a specific output location, usually a single ROOT file. Events themselves are described by the event level metadata object `EventInfo`. The event information uniquely identifies each event, provides the event type (e.g. data, simulation, etc.), information on the trigger and detectors states for the event, the measured and expected number of collisions, and sets of flags to identify detector and error states[2]. The `EventStreamInfo` object summarizes the information of all events written to a file. In addition the event stream information contains a list of the object's classes and corresponding keys in the output file.

2.6 Event format

The event format is a list of all objects the event stream writes to an output file along with the key in string form that identifies where the object is found in the file. This list is later used when reading the file as input to construct the objects from the file content. The event format object is built to be read in ROOT analyses without the bulk of the ATLAS production software.

2.7 File information

The `FileMetaData` stores information as a set of key value pairs. Keys are strings and values may be integers, strings or floating point numbers. The `FileMetaData` object stores a summary of properties shared by all events in the in the file, for example: the beam energy, detector geometry used in reconstruction, or the software release used to produce the file. The file metadata is stored in a format allowing access using ROOT, without the bulk of the ATLAS production software. The summary information is build by using the event information, taking the first set of values encountered in a job. Changes in a value throughout the job are ignored. This greatly simplifies the object. The simplification is justified by the observation that in the ATLAS production workflows all the values stored in the file metadata object are constant throughout a job. That means, the file metadata presents a reasonable simplification with respect to the event stream information.

[2]This information is made available for analyzers in the root-accessible `xAOD::EventInfo` object

2.8 Generator truth information

During the processing of generated Monte Carlo events, ATLAS applications write a set of *event weights* associated with each individual event into the output file. This information is written out as part of the event data. Each weight in the set corresponds to different generator configuration[3]. The generator truth information store machine-readable strings encoding these configurations. These strings are referred to as *weight names* in the *truth metadata* object.

In the Athena framework there are two components which manage truth metadata. One of these components is a *re-entrant algorithm* which adds new weight names to the transient metadata object during event processing and, at the end of the job, writes out this object into the output file metadata.

Another Athena component, an *AlgTool*, propagates truth metadata from the input into the output file, and also augments this information with additional weight names, if necessary.

2.9 Event Bookkeeping

At any stage of data processing event-level selection can be applied and only fraction of the initial events is stored in the output file. Event bookkeeping is used to track the impact of each of the selection criteria. Each criterion also called *cut* has a unique name and a cycle number corresponding to the job it is run in. All cuts in a job have the same cycle number which is increased each time a new job is run. If available a friendly name of the cycle taken from event stream information is also added.

When an event passes a cut, a counter is increased and the weight of the event is added to the sum for that cut. For each job there is also one cut representing all executed events. This is especially important for simulated events as the sum of weights represents an effective luminosity of the sample. As metadata for all cycles is propagated onward this retains the information of the total simulated effective luminosity of the sample even after selection is applied.

Any file can contain an arbitrary number of bookkeeping containers, each storing information about specific cuts. In case a file is not fully processed incoming bookkeeping information is marked as incomplete and moved in a separate container to clearly separate such cases. Furthermore systematic variations of Monte Carlo weights can also be separated in multiple containers.

3 Development

An in-depth review and rewrite of the ATLAS metadata content and infrastructure is well motivated but beyond the scope of this contribution. The development described here is limited to pragmatic solutions because person-power to take responsibility of the metadata system was found late in the AthenaMT development cycle.

Figure 1 is a diagram of a multi-threaded job processing multiple input files. It highlights the challenge for the in-file metadata infrastructure: ensure that events from multiple input files, which are processed concurrently, have access to appropriate in-file metadata. To meet this requirement we first attempt to reduce the problem by cleaning up obsolete code and removing as many clients as reasonable. After the clean-up we reduce the dependence on thread-unsafe constructs to a necessary minimum. The remaining components are redesigned for thread safety. Metadata objects are merged on the file boundary whenever possible, otherwise objects are resolved as appropriate for event and corresponding input file.

[3] Such as a combination of parton distribution function, factorization and renomalization scales used to investigate the effect of systematic variations in generator parameters

3.1 Thread-unsafe constructs

The most prominent construct that is dangerous in multi-threaded processing are incidents. Incidents are events managed by a service which receives notifications from an emitting client. The service then broadcasts the emitted incident to all registered listeners. Both emitters and listeners may run on multiple concurrent threads, so incidents must be handled with care.

We reduced our reliance on incidents to a necessary minimum: the opening of a new file is signaled by the *BeginInputFile* incident, the end of an input file be the *EndInputFile* incident. The writing out of the metadata to the output stream is initiated on the *MetaDataStop* incident. In the current design the ATLAS software these incidents may be assumed to occur sequentially, as indicated by Figure 1.

In AthenaMT, Athena components within individual events as well as multiple events are processed concurrently. To avoid the introduction of thread locks, the use of a *BeginEvent* incident was removed from use in the metadata management as this would cross thread boundaries.

3.2 Event Service

Event Service in ATLAS is special way of processing events on opportunistic computing resources, where jobs may be suddenly stopped and lose unfinished output streams they were producing when interrupted. To minimize such losses, event service jobs process events in small numbers - in groups called *Event Ranges* - that can be as short as a single event. Once a Range is processed, it is written to a separate output file, which is immediately finalized and not lost if the job interrupted later. These fragmentary output files are merged together when all Ranges are finished.

During Run 2, Event Service operated on the base of multi-process AthenaMP, with each worker process handling sequentially the Ranges assigned to it and keeping only one output stream open at a time. With the move to AthenaMT the situation became more complicated, as multiple Ranges, each with different output file, are now processed concurrently. The Range output files belong, logically, to the same output stream, but Athena actually handles metadata on a per-stream basis (the file being just a storage-specific implementation of one). Therefore a solution to provide consistent in-file metadata for each range output file was needed.

A new mechanism for handling in-file metadata for event service was implemented in AthenaMT. It is based on a templated metadata container type (`MetaCont<T>`) that can manage metadata objects. The container keeps a different copy of such object for each Event Range being processed. Access to the container is made thread-safe with the use of an internal mutex. Algorithms executing for a given event can access and update metadata objects that are relevant to the correct event range.

The existence of the metadata container is hidden by the metadata service API, which is identical both for event service jobs and standard Athena (MP/MT) jobs. In standard jobs the container keeps a single instance of the metadata object. The metadata container is also not visible in any way in the output file metadata. The `AthenaOutputStream` that writes metadata from the metadata store to the output stream is configured to recognize metadata containers and extract the metadata object relevant to the correct range. That approach ensures that files produced by event service jobs have identical format as standard output files.

3.3 Metadata with Intervals of Validity

In-file metadata with intervals of validity is managed by a dedicated Athena tool called `IOVDbMetaDataTool`. After all in-file metadata is loaded by the framework into the "InputMetaDataStore" on a file-open incident, this tool takes care of copying the IOV metadata to the "MetaDataStore", merging data and adjusting validity intervals if necessary. Clients, notably the `IOVDbSvc`, would access the data directly from the "MetaDataStore".

In order to introduce protection again concurrent access to the IOV metadata in the "MetaDataStore", we modified the clients to always request a shared lock from the Tool for the duration of the access. This concerns mainly attribute value queries, but also protects metadata consistency during larger operations, like writing all metadata to the output stream before it is closed. `IOVDbMetaDataTool` uses the same lock in exclusive mode to perform any IOV metadata updates. Access locking is implemented for metadata on a per-tool basis and not for all of the metadata to avoid unnecessary serialization of longer operations like writing.

During event processing, in-file IOV metadata is still read by the `IOVDbSvc` - although now all interactions between `IOVDbSvc` and the "MetaDataStore" go through the new, thread-safe API of `IOVDbMetaDataTool`. During the migration to AthenaMT, the role of the `IOVSvc`, controlling the validity of conditions data used in the event processing, was to a large extent taken over by the new condition handles and algorithms [10]. We foresee an integration of the in-file IOV metadata handling with the new Conditions infrastructure in the near future.

3.4 Run Parameters

A survey of all clients creating or consuming bytestream metadata found that none do so during the event loop. That means it is not necessary to resolve, on the event level, which is the appropriate object. To ensure all relevant run parameters are available in later processing and analysis steps all the `ByteStreamMetaDataTool` aggregates the data into a `ByteStreamMetaDataContainer`. Should the run parameters of the new input file indicate the same unique identifier, the rest of the parameters are assumed to be equal and the new object is not appended to the collection to avoid duplication.

3.5 Luminosity Block

During the review of the ATLAS metadata handling software components, it was determined that the current implementation of the Luminosity Block metadata handling mechanism can be considered *AthenaMT*-compatible. Both the *Algorithm* and the *AlgTool* use internal caches for counting events within luminosity blocks. Each Athena job can have at most one instance of either the *Algorithm* or the *AlgTool*. None of them offer any public API for accessing their private cache either. Hence, at this point no immediate changes are required in the code of these two components. For the future, though, we can implement some optimizations (e.g. turning the *Algorithm* into an *ReentrantAlgorithm*), and also work on making these two components thread-safe.

3.6 Trigger menu

Trigger development focused on migrating from a limited in-file metadata specific trigger menu format to a general JSON format. This unifies access to the trigger configuration in both online and offline environments. Additional modifications regarding per-slot configurations backed by the in-memory cache were added for MT-safety.

From 2020, the data summarizing the configuration of the trigger and *AthenaMT* processes running the High Level Trigger (HLT) are stored in a series of configuration files in JSON format [11]. Each run is configured from a Super Master Key (SMK) which is tied to a specific software release. The SMK references three configuration files: one for the trigger menu , one for the HLT menu, and one which fully configures the HLT *AthenaMT* processes. In addition the trigger prescale, HLT prescale, and bunch-group configuration are stored in JSON formatted files. This second set have their own unique integer keys because they may change during a run. All these files are stored in an online database in serialized form and referenced by an integer key.

When processing bytestream data the configuration files, except the one configuring the AthenaMT process of the HLT, are downloaded from the database during the first step of offline processing. The JSON content of the downloaded files is stored as in-file metadata in the output files. The in-file metadata objects are five ROOT stream-able `xAOD::TriggerMenuJsonAuxContainer` collections: one for each of the trigger menu, HLT menu, trigger prescale, HLT prescale, and bunchgroup information. Each collection contains information from the set of JSON files required to cover the configuration of every event in the file. The payload of each container is stored as a JSON formatted string. ROOT compression is used to reduce the size of the container on disk.

When a reconstructed file is subsequently opened, the contained payloads are added to an in-memory cache of available configurations. The string objects are decoded into `boost::ptree` data structures. The trigger configuration service's in-memory cache is updated on the *BeginInputFile* incident after having obtained an exclusive lock over the mutable cache.

Upon the *BeginEvent* incident, a shared lock over the immutable cache is used to verify that the configuration loaded into the event's processing slot is correct or to update it from the service's cache if not. This check is performed by reading the new event's SMK, trigger prescale key, HLT prescale key, and bunchgroup key from the per-event data collection. The obtained keys are compared to those currently loaded in the event's slot. With this design, the trigger configuration service is able to supply the correct configuration to multiple simultaneous clients which are making requests from multiple simultaneous events, and which may happen to cross trigger configuration or file boundaries.

3.7 Event stream information

As described in Section 2.5 the event stream information is an object that accumulates event information. The action of accumulating information is a natural fit for merging two event stream information objects. A `MetaDataTool` merges event stream information from new input files with the object in the "MetaDataStore" on `BeginInputFile` incidents. This object is not written to the output file. To support the event service workflow the event stream information is handled using the metadata service.

A separate tool belonging to the output stream creates and fills a separate event stream information object from the events processed by the output stream. This object, summarizing the content of the output file, is written into the metadata content of the output file.

3.8 Event format

Recall that event format objects are lists of object classes and corresponding keys. These lists can naturally be merged. The event format metadata is managed using the same process as for the event stream information.

3.9 File information

Recall from Section 2.7 that the file information only has the first instance of any encountered piece of information. That means the `FileMetaData` objects cannot be merged. The previous implementation handled this by adopting the first values encountered and warning the user about differences. The thread safe implementation adopts the same practice: using the first object provided and warning about differences when a new input file is opened. Aside from assuring thread safety the new implementation for the file information management follows the same workflow as the event stream information.

3.10 Generator truth information

The *Reentrant Algorithm*, which creates new transient Truth metadata objects and fills them in with the event weight names, protects all write operations to these objects with a mutex. The `AlgTool`, which propagates metadata information from the input to the output, manages all transient objects in a private cache, and does not offer any public API to potential clients. Hence, after a detailed review of the implementation of the aforementioned components, it was determined that the current implementation of the Truth metadata handling mechanism can be considered AthenaMT-compatible.

3.11 Event Bookkeeping

Event bookkeeping has a very broad and diverse usage in the ATLAS code-base. In the past bookkeeping was done during the whole reconstruction chain but often not making any selection. This has now been disabled to avoid redundant information and potential issues with incomplete bookkeepers. For all other clients a flexible utility class has been developed to communicate with the bookkeeping service and can be plugged to any kind of algorithm. It ensures selection is communicated in a thread-safe way only once per algorithm execution on a specific event. Furthermore, all Monte Carlo systematic variations are now processed for every cut so full bookkeeping is available.

4 Conclusion

This paper provides and overview of the ATLAS in-file metadata contents and the software infrastructure supporting this descriptive information. In 2020 the in-file metadata system of the Athena framework was redesigned to support concurrent event processing in multi-threaded simulation and reconstruction workflows. We demonstrated how the code migration challenges presented by each category of the ATLAS in-file metadata were met with pragmatic solutions. The new system can now provide in-file metadata clients with the appropriate descriptive information in multi-threaded concurrent event processing. The migration to the multi-threaded framework is a necessary first step for the ATLAS experiment to be able to take advantage of heterogeneous hardware architectures, such as graphical processing units. The work on redesigning the in-file metadata system for operation in the multi-threaded environment highlighted a number of areas where the ATLAS metadata infrastructure must be improved to take advantage of novel computing hardware and processing techniques.

References

[1] ATLAS Collaboration (ATLAS), JINST **3**, S08003 (2008)

[2] D. Malon, P. van Gemmeren, R. Hawkings, A. Schaffer, J. Phys. Conf. Ser. **119**, 042022 (2008)
[3] ATLAS Collaboration, *Athena* (2020), https://doi.org/10.5281/zenodo.2641996
[4] ATLAS Collaboration, J. Phys. Conf. Ser. **898**, 042009 (2017)
[5] G. Amadio et al. (ROOT Team) (2020), 2004.07675
[6] P. Calafiura, C.G. Leggett, D.R. Quarrie, H. Ma, S. Rajagopalan, eConf **C0303241**, MOJT008 (2003)
[7] ATLAS Collaboration, *eformat* (2020), https://gitlab.cern.ch/atlas-tdaq-software/eformat/
[8] C. Leggett, in *14th International Conference on Computing in High-Energy and Nuclear Physics* (2005), pp. 528–530
[9] ATLAS Collaboration, JINST **15**, P10004 (2020), 2007.12539
[10] ATLAS Collaboration, PoS **ICHEP2016**, 188 (2016)
[11] *The JavaScript Object Notation (JSON) Data Interchange Format*, https://tools.ietf.org/pdf/std90.pdf (2017), accessed: 2020-01-31