Flexible, robust and minimal-overhead Event Data Model for track reconstruction in ACTS

Paul Gessinger[1,*]

[1]CERN

Abstract. ACTS (A Common Tracking Software) is an experiment independent toolkit for track reconstruction, which is designed from the ground up for thread-safety and high performance. It is built to accommodate different experiment deployment scenarios, and also serves as community platform for research and development of new approaches and algorithms.
The Event Data Model (EDM) is a critical piece of the tracking library that is visible to clients. Until this point, ACTS was mostly focused on an internal EDM, targeting data interchange various components in the toolkit. This contribution reports on a new and improved client EDM for ACTS. For an experiment-agnostic toolkit like ACTS, this requires strong abstractions of potentially experiment-specific details, including event context data like sensor alignments, and tracking inputs like measurements. By applying similar abstraction strategies, the presented EDM can be an expressive, low-overhead abstraction over experiment-specific backends, and seamlessly integrates into an experiment framework and IO model.
The presented EDM includes the ACTS track class, the main data type which tracking clients interact with. It is designed to be interfaced with different IO backends, and also flexible enough to support dynamic information required by various track fitters. At the same time, careful design ensures it can seamlessly serve as a key data object in experiment reconstruction data flows.
In this contribution, the interaction of this centerpiece of the example workflows in ACTS with the standalone ROOT IO, as well as the integration with the EDM4hep package will be shown, and key performance characteristics discussed.

1 Track reconstruction

Track reconstruction is the process of converting raw signals from charged particle detectors into higher-level measurements of the physical particles that traversed the detector. Depending on the experiment setup, detector technologies and geometric configuration, different stages of pattern recognition algorithms are deployed to perform this process.

In many silicon-based tracking detectors, and in particular in the ATLAS[1] experiment and its reconstruction software [2], reconstruction starts by grouping measurements into combinations loosely compatible with a track coming from the interaction region. Starting from

*e-mail: paul.gessinger@cern.ch

clustered detector signals, measurements are first converted into three-dimensional space-points, and then grouped into triplets (*seeding*), allowing for a crude estimation of the curvature and thus momentum.

After a sequence of filtering steps, a *track finding* stage explores the recorded event looking for additional sensor measurements that are compatible with the initial track hypothesis. The resulting track candidates can then be refitted to yield the highest-precision estimate of the associated particle properties. In case of ambiguities between competing tracks in an event, an ambiguity resolution stage can identify and reduce duplications.

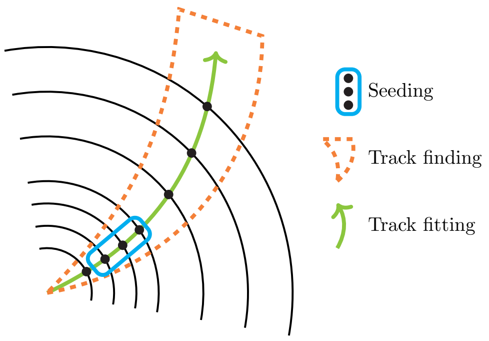

Figure 1. Illustration of a track reconstruction chain in ACTS. Seeding from space-points yields track candidates, that are completed by track finding. A final track fit yields high-precision properties.

2 The ACTS toolkit

ACTS [3] is an experiment independent software toolkit for charged particle reconstruction. Originally based off of the ATLAS tracking code, it has since significant development and improvement. The core concepts and approaches, however, remain the same.

ACTS consists of a number of components that serve various purposes. A geometry modelling component is used to describe both sensitive and passive parts of the detector. The passive material is treated in an approximated fashion, allowing significant speedup compared to fully detailed geometry descriptions, with negligible impact on precision. A pattern recognition component implements a triplet seeding algorithm based on the ATLAS pattern recognition. It is designed to be flexible and allow experiment-specific tuning of ranking and filtering criteria. Track finding is achieved with a track finding component, which contains a combinatorial Kalman Filter [4] implementation, that can iteratively attach measurements to a track candidate. Multiple fitters are included in the track fitter component. A production-ready Kalman Filter is the backbone of this component, with a dedicated fitter for particles with non-gaussian material interactions and an experimental global fitter based on χ^2 minimization are available. A robust vertexing component completes the toolkit, with multiple vertex finding and fitting algorithms available for primary vertex reconstruction.

2.1 Event Data Model

The components mentioned in section 2 can be assembled into a complete track reconstruction chain. In fact, ACTS ships with a fully-integrated example reconstruction chain using the OpenDataDetector [5]. Between these components, data needs to be exchanged in a well-defined way. This is achieved through the Event Data Model (EDM), which is a set of data types and interfaces representing the content of an event. Until very recently, ACTS has focused mainly on an *internal* EDM, which is really focused on efficient interchange between components inside the toolkit, sometimes at the cost of usability for clients. With the main reconstruction chain becomes more and more mature, however, the focus has shifted to a more client-oriented EDM encapsulating the outputs of tracking.

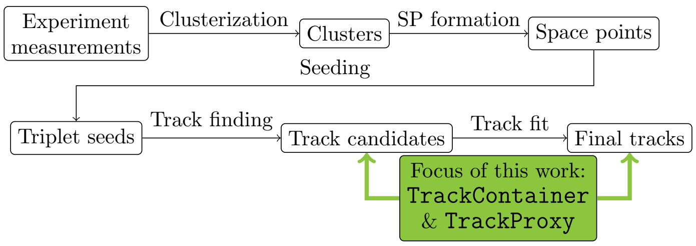

Figure 2. Diagram showing the stages and data flow of a track reconstruction chain. The boxes show EDM objects that are passed between the stages. `TrackContainer` and `TrackProxy` are highlighted as the focus of this work.

Figure 2 shows an overview of the EDM data types and how they form a data-flow sequence. Measurements coming from the experiment software are the main inputs of the chain. This data-type is abstracted in ACTS in a way that allows the details of these measurements to be fully experiment-specific, for example allowing careful treatment of hardware details. ACTS ships with a clusterization algorithm, which can turn segmented raw measurements into clusters, the second EDM object in the chain, representing particle intersections with the sensors. For the creation of track seeds, clusters need to be converted into three-dimensional space-points, combining information from multiple clusters where needed, e.g. for one-dimensional silicon strip sensors. The space-points and seeds are part of the EDM as well, since they are handed over to the track finding component. This component is responsible for the creation of completed tracks, which can then optionally be refitted with a precision track fitter.

Both the track finding and the precision track fit produce track objects, which are the primary output of the tracking chain.

3 High-level Track Event Data Model

Track information in ACTS can be divided into two parts: track-level information and track state-level information.

Track-level information are properties that relate to the full track. This includes the fitted track parameters with respect to some reference point, often the origin of the detector or the beamspot. It can also include summary information from the track finding stage, like the overall number of clusters that were used in the creation of the track, or the fit quality from the track fit.

Tracks are built-up from track states, where each track state corresponds to a discrete state determining the track properties. This mainly includes measurements states, expected intersections with sensors where no measurement was found (*holes*), and intersections with known passive material. The EDM allows building up a track from these track states iteratively. For example, the Kalman Filter will append track states to the sequence whenever it encounters a sensitive detector layer. The content of the track states is defined such that the fitter can store all relevant information, with as little need for extra information as possible.

It is also designed to be flexible enough to support different fitters, which might require different information to be stored, as well as the combinatorial Kalman Filter, which produces a tree of track states, instead of a fully linear sequence.

Ultimately, each output track is associated with a well-defined sequence of track states, allowing downstream consumers of the EDM to access the fully detailed information produced during track reconstruction.

3.1 Architecture

Several considerations have been taken into account during the design of the EDM architecture. A main goal was to store different properties as individual vectors of information, rather than having a single vector of a structure containing all information. At the same time, the access should still behave in an object-oriented way, where an object corresponds to a track or track state. Figure 3 shows this object-oriented access model for the example of the track container and track proxy object. The track container holds vectors of the various pieces of information, and has methods to add a track, and to allow iteration over all tracks. This iteration, or index based access, yields a track proxy object, which exposes the properties as methods returning references, while internally only holding a pointer to and an index into the track container. The types are carefully built to preserve const-correctness, i.e. even though a track proxy is a value type which can be copied, it will not allow modification of the underlying track container if it is immutable.

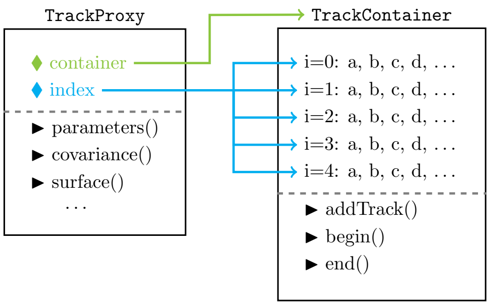

Figure 3. Illustration of the proxy pattern used in the track EDM. The track proxy logically represents a single track, and points to the data stored in the track container.

Another important goal was to make the track EDM fully agnostic to the concrete persistency framework of an experiment. This is crucial, since the need for conversion between different representations of the same information, just to be able to use the experiments input-output system can be a significant overhead.

3.2 Implementation

To make the EDM implementation independent of an experiment persistency framework, it is separated into a *frontend layer* and a *backend layer*. The frontend layer contains user-facing getters and setters, as well as any convenience methods that might be helpful. These methods are located either in the proxy objects or in the containers, depending on whether they operate on a single element or the entire container.

Overall, there are four main classes that make up the frontend layer: `TrackProxy`, `TrackContainer`, `TrackStateProxy` and `MultiTrajectory`. The latter serves as the

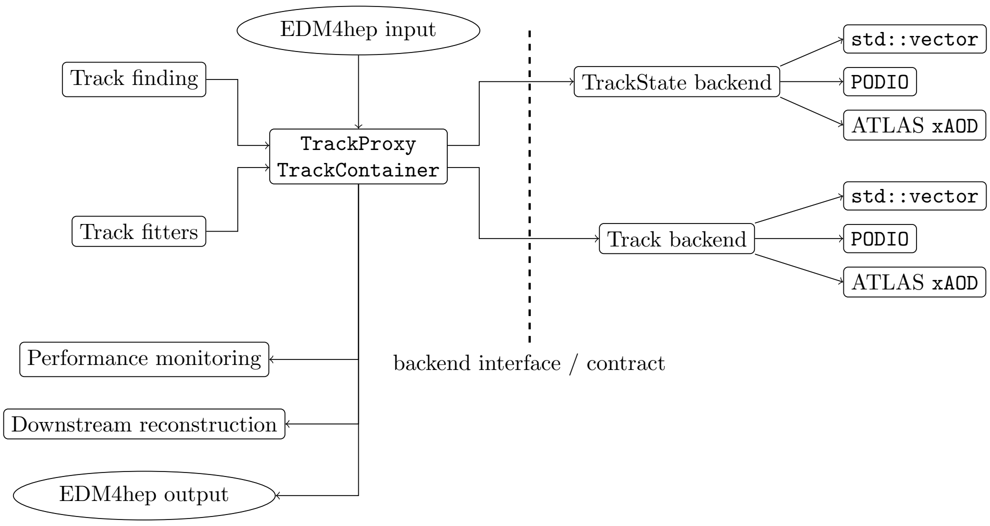

Figure 4. Diagram of the EDM architecture. The frontend layer is used by other ACTS components, and downstream clients. It is separated from the backend layer by an interface. Conversion to and from EDM4hep is possible. Examples of direct backend implementations are shown.

track state container, where the name indicates that it is able to handle a branching tree structure of track states. `TrackProxy` and `TrackStateProxy` expose methods to get the local track parameters and covariance, corresponding reference surface, and also includes global statistics like the total number of measurements, outliers or holes in case of `TrackProxy`. `TrackProxy` also has a method to conveniently iterate over the associated track states from the outside inwards, yielding `TrackStateProxy` objects from the track state container.

In case of `TrackStateProxy`, functionality is exposed in the frontend layer to allocate optional components, with the goal of reduced memory footprint. There are two main uses of this: track parameters and measurements. The track-state EDM supports storing up to three sets of local track parameters and covariance matrices, modeled after the information the Kalman Filter formalism needs to store: predicted, filtered and smoothed parameters and covariances. In case of combinatorial track finding, track hypothesis can start out with a common sequence of track states, and then branch out when multiple compatible measurements are encountered. The track state EDM allows allocating only the track parameters that are needed, and even allows sharing the same track parameters between multiple track states, so that branching track states can share for example the same predicted parameters. How this is achieved is left to the backend layer. Measurements are handled in a similar way, where the track finding decides how much storage is needed based on the number of dimensions of an incoming measurement. It then instructs the EDM through the frontend layer to ensure enough memory is available, where the specifics are again left up to the backend layer.

The backend layer exposes an interface that is used by the frontend layer to store and retrieve information. It uses dedicated methods where needed, such as for storing reference surfaces or source-link objects, which are lightweight container objects for experiment-specific measurements. For the majority of components, the frontend communicates with the backend through a single method to obtain references to the underlying data. Components are accessed via hashes of the component name, where the hashes are calculated at compile-time wherever possible. The backend can then use the hashed component name to retrieve the relevant memory. To allow directly manipulating the backing memory, the frontend expects the backend to return references into the backing storage.

`TrackProxy` provides a method to copy a track between different track containers, and only uses the frontend layer to accomplish this. This means that copying tracks between different backend implementations is trivial.

Figure 4 shows a diagram of the EDM architecture. At the center are the `TrackProxy` and `TrackContainer`. These classes are produced by the track finding and track fitting components, and are the main interface point with the clients of tracking. In ACTS itself, all of the performance monitoring and downstream reconstruction is either directly built on top of these objects, or converts them into an internal EDM depending on the use case. Behind the backend interface, the track container coordinates with both a track state and a track backend, where a few examples are shown, and will be discussed below.

3.3 Integration with storage technologies

EDM4hep

EDM4hep [6] is a common EDM implementation that is meant to cleanly integrate and connect different pieces of a reconstruction chain, mainly geared towards collider studies. It is based on the PODIO [7] framework, which uses a high-level declarative format to define an EDM which can then be used to generate C++ code to manipulate the data objects. The tracking related parts EDM4hep use the LCIO parametrization [8], which defines which information is stored to define the properties of a track:

$$d_0, z_0, \phi, \tan\lambda, \Omega. \tag{1}$$

Here, d_0 and z_0 are the transverse and longitudinal impact parameters, relative to some reference point. ϕ is the azimuth angle of the track, while $\tan\lambda$ is directly related to the polar angle θ. Ω is the signed inverse of the curvature radius of the track, and is therefore a measure of the momentum of a track, in the presence of a magnetic field.

ACTS itself natively uses a parametrization adopted from the ATLAS experiment:

$$l_0, l_1, \phi, \theta, q/p, t. \tag{2}$$

Here, l_0 and l_1 are the general local two-dimensional coordinates, whose interpretation is defined by a reference surface. ϕ and θ are again the azimuth and polar angles, while q/p is the signed inverse momentum. The time t is also included by default.

Note that ACTS requires a reference surfaces and stores a local position on it. This means that during the conversion from ACTS to EDM4hep, the local position on the sensor is lost, as the perigee reference point has to be set to the global track position to be correct. As a consequence, when converting from EDM4hep to ACTS, the local position information cannot be recovered.

Additionally, it is not feasible to implement EDM4hep as a backend for the abstracted track EDM infrastructure described in the previous sections, as it does not fulfill the required contract needed by the interface mechanism.

For this reason, the EDM4hep integration in ACTS is limited to conversion to and from the high-level track EDM, as is indicated in Figure 4. The conversion includes track states and is technically complete, except for the local position on surface. When converting EDM4hep tracks to the high-level track EDM, ad-hoc perigee surfaces are created to model the information stored in EDM4hep.

PODIO

As mentioned before, PODIO can be used to define an EDM using a high-level declarative approach. A PODIO backend was implemented that can be used with the high-level track EDM discussed before.

The PODIO EDM consists of the following primary *datatypes*:

- `ActsPodioEdm::Track`
- `ActsPodioEdm::TrackState`
- `ActsPodioEdm::BoundParameters`
- `ActsPodioEdm::Jacobian`

These datatypes do not contain the full information directly, but instead delegate this to the following *components*:

- `ActsPodioEdm::Vector3f`
- `ActsPodioEdm::Surface`
- `ActsPodioEdm::TrackInfo`
- `ActsPodioEdm::TrackStateInfo`
- `ActsPodioEdm::BoundParametersInfo`
- `ActsPodioEdm::JacobianInfo`

Using these components, it is possible for the backend implementation to supply constant and mutable references to the underlying information, which is required to implement the interface contract expected by the frontend layer.

The functionality of the PODIO backend is tested using a common test suite that is shared with the default transient-only backend. In addition, a roundtrip test is implemented that ensures that information can be written and read in, reproducing the same information. The backend has also been tested to work correctly with the Kalman Filter, demonstrating that the abstraction mechanism is successful.

4 Conclusion

This document presents work on a high-level track EDM in the ACTS toolkit. In this context, a two-part EDM architecture was presented, consisting of a frontend layer that other components of the tracking as well as clients of tracking interact with, and a backend layer that is responsible for the actual storage. A transient-only backend is used by default.

The possibility to convert to and from the EDM4hep track EDM was discussed, and is included in the ACTS library. Additionally, a full backend implementation using PODIO was discussed, which allows components like the Kalman Filter in ACTS to transparently write outputs to PODIO files, enabling them to be used by downstream consumers.

5 Acknowledgements

The work has been supported by the CERN Strategic Programme on Technologies for Future Experiments. https://ep-rnd.web.cern.ch/

References

[1] G. Aad et al. (ATLAS), JINST **3**, S08003 (2008)
[2] *Athena*, https://doi.org/10.5281/zenodo.2641996 (2019)
[3] X. Ai, C. Allaire, N. Calace, A. Czirkos, M. Elsing, I. Ene, R. Farkas, L.G. Gagnon, R. Garg, P. Gessinger et al., Computing and Software for Big Science **6** (2022)
[4] R. Frühwirth, Nucl. Instrum. Meth. **A262**, 444 (1987)
[5] C. Allaire, P. Gessinger, J. Hdrinka, M. Kiehn, F. Kimpel, J. Niermann, A. Salzburger, S. Sevova, *OpenDataDetector*, https://doi.org/10.5281/zenodo.4674401 (2021)
[6] *EDM4hep GitHub repository*, https://github.com/key4hep/EDM4hep
[7] *podio GitHub repository*, https://github.com/AIDASoft/podio
[8] T. Kraemer, DESY, *Track Parameters in LCIO* (2006), https://bib-pubdb1.desy.de/record/81214