ASAM OSI Flatbuffers

Application Report

Clemens Linnhoff, Philipp Rosenberger

March 2022

ASAM OSI Flatbuffers

Application Report

Abstract

At the current stage of ASAM Open Simulation Interface (OSI), a possibly backward compatibility breaking new release 4.0 is discussed within the OSI change control board (CCB). As there have to be good arguments for such a step in a standard by providing new opportunities in the other hand, OSI CCB has commissioned Persival GmbH to evaluate application and performance of Google Flatbuffers instead of the currently used Protobuf for serialization and deserialization of the data for its exchange.

This first report is discussing Flatbuffers application for OSI during the pioneering implementation of working examples. It is accompanied by a detailed performance analysis in a second report. The project goal is to support the decision of ASAM OSI to change the standard's serialization method in future releases. The project period has been from Jan 18 - Mar 31, 2022.

After a short introduction to Flatbuffers and fbs files in general, the differences in the header files between proto and fbs files are shown. Then the concept of Builders in Flatbuffers is explained.

Afterwards, appearing difficulties during conversion from proto to fbs are presented. It starts with the already reported bug in Flatbuffers regarding type enums in OSI when converting them. Second, the differences between tables and structs are provided and their usages in OSI are proposed.

Next, some general implementation considerations are given. They start with considerations on the common practice of copying SensorView to SensorData, presents substitutes for Protobuf Functions and shortly explains the experiences with mutable fields and the buffer format.

Additionally, detailed application experiences are provided for the generation of a SensorView with the DummySource example from ASAM OSI Sensor Model Packaging (OSMP). Besides, the usage of the object API that comes as an alternative for “pure” Flatbuffers implementation is explained. Furthermore, the lessons learned from the exemplary conversion of an open-source reflection based lidar detection model are explained. Finally, a short conclusion on Flatbuffers application experience is provided.

Contents

1 Flatbuffers – A Short Introduction

1.1 What is Flatbuffers?

FlatBuffers is an efficient cross platform serialization library for C++, C#, C, Go, Java, JavaScript, PHP, Python, etc. It was originally created at Google for game development and other performance-critical applications and is now open-Source on GitHub under the Apache license, v2[1].

Access to serialized data is provided without parsing / unpacking due to hierarchically structured data in the flat binary buffer. The only memory needed for that data access is that of the buffer as it requires zero additional allocations in C++. (Other languages may vary.) The hereby performance is close to the speed of raw structs access. It supports Optional fields for forwards and backwards compatibility

Due to its homepage[1], it is said that a lot of choice in what data to write and what not, and how to design data structures is provided. It has a tiny code footprint by small amounts of generated code and just a single small header as the minimum dependency. It is strongly typed and errors happen at compile time, so no manually writing repetitive and error prone run-time checks are necessary.

1.2 Generate FBS Files and Headers from Proto Files

Flatbuffers data structures are defined within fbs schema files using the interface definition language. So, the fbs files are the counterpart to the proto files in Protobuf. Code Snippet 1 shows the definition of a SensorView message in a proto file.

```
message SensorView
{
  optional InterfaceVersion version = 1;
  optional Timestamp timestamp = 2;
  optional Identifier sensor_id = 3;
  optional MountingPosition mounting_position = 4;
  optional MountingPosition mounting_position_rmse = 5;
  optional HostVehicleData host_vehicle_data = 6;
  optional GroundTruth global_ground_truth = 7;
  optional Identifier host_vehicle_id = 8;
  repeated GenericSensorView generic_sensor_view = 1000;
  repeated RadarSensorView radar_sensor_view = 1001;
  repeated LidarSensorView lidar_sensor_view = 1002;
  repeated CameraSensorView camera_sensor_view = 1003;
  repeated UltrasonicSensorView ultrasonic_sensor_view = 1004;
}

message GenericSensorView
{
  optional GenericSensorViewConfiguration view_configuration = 1;
}
```

Code Snippet 1: Example of a message definition in a proto file

The counterpart of messages are tables in Flatbuffers. The definition of a SensorView table in an FBS file is shown in Code Snippet 2.

[1] Source: https://google.github.io/flatbuffers/

```
table SensorView
{
version:osi3.InterfaceVersion;
timestamp:osi3.Timestamp;
sensor_id:osi3.Identifier;
mounting_position:osi3.MountingPosition;
mounting_position_rmse:osi3.MountingPosition;
host_vehicle_data:osi3.HostVehicleData;
global_ground_truth:osi3.GroundTruth;
host_vehicle_id:osi3.Identifier;
generic_sensor_view:[osi3.GenericSensorView];
radar_sensor_view:[osi3.RadarSensorView];
lidar_sensor_view:[osi3.LidarSensorView];
camera_sensor_view:[osi3.CameraSensorView];
ultrasonic_sensor_view:[osi3.UltrasonicSensorView];
}

table GenericSensorView
{
  view_configuration:osi3.GenericSensorViewConfiguration;
}
```

Code Snippet 2: Example of a message definition in an fbs file

Flatbuffers comes with a conversion tool to automatically generate fbs files from existing proto files: The flatc compiler[2]. Flatc not only converts proto files to fbs, it also created the C++ header files needed for using Flatbuffers in a C++ model.

1.3 Using Builders

To start using Flatbuffers, an instance of FlatBufferBuilder needs to be initialized with

```
flatbuffers::FlatBufferBuilder builder(1024);
```

Afterwards, there are two ways to create a table. The first option is to use the Create functions provided by the Flatbuffers header files. These functions fil all fields of the data structure at once, for example when setting a position with x, y and z coordinates:

```
flatbuffers::Offset<osi3::Vector3d> pos = osi3::CreateVector3d(builder, 50, 2, 0.0);
```

Instead of setting all the fields at once, a builder object can be used:

```
osi3::BaseMovingBuilder base_moving_builder(builder);
base_moving_builder.add_position(pos);
flatbuffers::Offset<osi3::BaseMoving> base_moving = base_moving_builder.Finish();
```

In the end, the FlatBufferBuilder has to be finished, which closes the buffer:

```
builder.Finish(sensor_view);
```

[2] https://google.github.io/flatbuffers/flatbuffers_guide_using_schema_compiler.html

2 Flatbuffers in OSI

With OSI Version 3.4.0 Flatbuffers was included in an OSI release for the first time. Flatbuffers is a submodule in the OSI repository and CMake commands were added to generate fbs files and C++ headers from the proto files with the flatc compiler. Flatbuffers is activated with the CMake flag BUILD_FLATBUFFER. By building the target open_simulation_interface_fbs_build the OSI proto files are converted to fbs and C++ headers are generated. The fbs files are located in *build/open-simulation-interface* and the OSI C++ headers are stored in *build/open-simulation-interface/include*.

2.1 Considerations for Conversion from Proto to FBS

The conversion from proto files to fbs unfortunately is not completely plug and play, as it is supposed to be. The following considerations have to be taken into account currently.

2.1.1 Types

In OSI there are several enums called "Type" in different messages. There is the type of a MovingObject, the type of a VehicleClassification within a MovingObject and there is also the type of the Classification of a StationaryObject. By converting the proto files from OSI 3.4.0 with the flatc compiler, these types are assigned to the wrong tables in the fbs files, as marked in red in Code Snippet 3.

This issue seems to be a bug in the official flatc converter and was brought to the developers' attention on the Flatbuffers GitHub repository with issue #7109 [3]. There is however a fix that can be used until the flatc bug is addressed. The order of defining the enums and the message utilizing it in the proto file matters. Currently, enums are defined after the messages, as can be seen in Code Snippet 4.

But if the enum is defined prior to its use, as demonstrated in Code Snippet 5, flatc assigns it correctly in the fbs files. This change might have an impact on backwards compatibility of the proto files. However, this was not evaluated during this report and needs to be addressed when applying in a future OSI release. As Types are not needed for the performance study, for now this will simply be commented out e.g. in the DummySource example in OSMP, where MovingObjects are created.

```
table MovingObject {
    id:osi3.Identifier;
    base:osi3.BaseMoving;
    type:osi3.StationaryObject_.Classification_.Type;
   […]
}
[…]
namespace osi3.MovingObject_;
table VehicleClassification {
    type:osi3.MovingObject_.Type;
    light_state:osi3.MovingObject_.VehicleClassification_.LightState;
    has_trailer:bool;
    trailer_id:osi3.Identifier;
}
```

Code Snippet 3: Excerpt from osi_object.fbs with wrong type assignment

[3] https://github.com/google/flatbuffers/issues/7109

```
message MovingObject
{
  optional Identifier id = 1;
  optional BaseMoving base = 2;
  optional Type type = 3;
  repeated Identifier assigned_lane_id = 4;
  …
  enum Type
  {
    TYPE_UNKNOWN = 0;
    TYPE_OTHER = 1;
    TYPE_VEHICLE = 2;
    …
  }
}
```
Code Snippet 4: Definition of the MovingObject message in OSI 3.4.0 osi_object.proto

```
message MovingObject
{
  enum Type
  {
    TYPE_UNKNOWN = 0;
    TYPE_OTHER = 1;
    TYPE_VEHICLE = 2;
    …
  }
  optional Identifier id = 1;
  optional BaseMoving base = 2;
  optional Type type = 3;
  repeated Identifier assigned_lane_id = 4;
  …
}
```
Code Snippet 5: Inversed order of defining enums and messages in proto files.

2.1.2 Tables vs. Structs

While tables are the main way for defining data structures in Flatbuffers, there are also structs[4]. Flatc automatically converts all Protobuf messages to tables, but manually changing some tables to structs might be considered. There are two main benefits to using structs over tables for certain messages. First the code handling is easier with structs. There is no need for a builder to generate a struct, it can be implemented as usual with structs in C++ and then be added to an overlying table. The second benefit is performance. Structs use less memory and are faster to access. For a detailed analysis, please refer to the separate performance report.

But there is one major downside to using structs over tables. Structs cannot be extended with further fields during a minor release while keeping backwards compatibility. Therefore, structs can only be used for fields that are not expected to change in the future. For primitive fields like Vector3D, Timestamp, Identifier etc. this will be a valid option.

[4] https://google.github.io/flatbuffers/md__schemas.html

2.2 Considerations when Using OSI with Flatbuffers

There are not just some oddities with fbs and header generation. There are also certain things to consider when using Flatbuffers.

2.2.1 Copying SensorView to SensorData

Flutbuffers does not come with a deep copy function, so there is no easy way to copy table or entire buffers to a different buffer. With OSI it is currently common practice to copy the SensorView input of a sensor model to the SensorData output. As there is no standard function to do this with Flatbuffers, one has to be custom written. However, for efficiency reasons it has to be discussed if this practice is necessary in the first place. It is a lot more efficient not to copy data structure from on buffer to the other, but to just access the first buffer directly, instead.

2.2.2 Substitutes for Protobuf Functions

Most developers of OSI 3.4.0 and prior versions are used to Protobuf standard functions like *empty()*, *has_*()* and **_size()*, as applied in Code Snippet 6.

```
int num_moving_obj = sensor_view.global_ground_truth().moving_object_size();
bool has_moving_obj = !sensor_view.global_ground_truth().moving_object().empty();
bool has_base = sensor_view.global_ground_truth().moving_object(0).has_base();
```

Code Snippet 6: Protobuf functions applied with OSI 3.4.0

These functions do not exist in Flatbuffers, but there are substitutes, as can be seen in Code Snippet 7.

```
size_t num_moving_obj = sensor_view->global_ground_truth()->moving_object()->size();
bool has_moving_obj = sensor_view->global_ground_truth()->moving_object();
bool has_base = sensor_view.global_ground_truth().moving_object()->Get(0)->base();
```

Code Snippet 7: Substitutes for Protobuf functions in Flatbuffers

2.2.3 Mutable Fields

There is the possibility of using mutable fields in Flatbuffers, but this is slower and not recommended.[5]

This means, that once a table is finished, the data is not changeable anymore. It exists only as an offset value after finishing and is not even accessible anymore until the whole buffer is finished.

2.2.4 Buffer Format

Just a minor difference is the format of the pointer to the buffer. While with Protobuf it is implemented as a string* in OSMP, with Flatbuffers it is a string. This is implemented in the OSMPDummy Examples in the new Flatbuffers branch of the OSMP GitHub Repository provided with this report[6].

[5] https://google.github.io/flatbuffers/flatbuffers_guide_tutorial.html
[6] https://github.com/OpenSimulationInterface/osi-sensor-model-packaging/blob/feature/flatbuffers_examples/examples/OSMPDummySensor_flat/OSMPDummySensor.h#L211

3 Flatbuffers OSI in Sensor Model Application

3.1 Generate a SensorView

As a reference implementation on how to generate a SensorView table with Flatbuffers, the OSMPDummySource example was fully converted from its original Protobuf implementation to Flatbuffers[7].

Code Snippet 8 shows a simple example on how to generate and serialize a SensorView message with one MovingObject using Protobuf. First, a SensorView object is initialized. Then mutable fields are filled using setter functions provided by Protobuf. In this case a sensor ID and a global ground truth with one moving object are added. The object has an ID an an x-y position. In the end, the SensorView message is serialized to string into the buffer.

```
osi3::SensorView sensor_view;
sensor_view.mutable_sensor_id()->set_value(1);

osi3::GroundTruth *currentGT = sensor_view.mutable_global_ground_truth();

osi3::MovingObject *veh = currentGT->add_moving_object();
veh->mutable_id()->set_value(10);
veh->mutable_base()->mutable_position()->set_x(50);
veh->mutable_base()->mutable_position()->set_y(2);

sensor_view.SerializeToString(currentBuffer);
encode_pointer_to_integer(currentBuffer->data(),integer_vars[BASEHI_IDX],...);
```

Code Snippet 8: Generate SensorView with one MovingObject in Protobuf OSI

Code Snippet 9 shows the exact same functionality written with Flatbuffers. As described in section 1.3, a builder needs to be initialized. Because only one builder can be used at a time, the order of setting the individual fields is reversed, compared to Protobuf. Instead of creating the SensorView object first, it starts with the innermost field of the nested data structure, in this case the position of the moving object. To create the position vector, the CreateVector3d function, provided by Flatbuffers, is used.

Next, the position has to be added to the base. This is done with a base_moving_builder. The base is then added to the moving object with a moving_object_builder and the finished offset is pushed to a standard vector. This whole process of setting a position, base, and moving object can be done in a loop in case of multiple objects. They would all be added to the vector. When all objects are processed, the offset of the vector is created with the CreateVector function. Then the global ground truth, sensor ID and finally the SensorView are created the same way. In the end, the builder is finished. At that point, everything is already serialized. Now the pointer to the buffer and its size are available with the provided functions. The buffer is casted to a string and the integer values for OSMP are set.

```
flatbuffers::FlatBufferBuilder builder(1024);

std::vector<flatbuffers::Offset<osi3::MovingObject>> moving_object_vector;

flatbuffers::Offset<osi3::Vector3d> position = osi3::CreateVector3d(builder, 50, 2,
0.0);
```

[7] https://github.com/OpenSimulationInterface/osi-sensor-model-packaging/tree/feature/flatbuffers_examples/examples/OSMPDummySource_flat

```
osi3::BaseMovingBuilder base_moving_builder(builder);
base_moving_builder.add_position(position);
flatbuffers::Offset<osi3::BaseMoving> base_moving = base_moving_builder.Finish();

osi3::MovingObjectBuilder moving_object_builder(builder);
moving_object_builder.add_base(base_moving);
moving_object_vector.push_back(moving_object_builder.Finish());

auto moving_object_flatvector = builder.CreateVector(moving_object_vector);

osi3::GroundTruthBuilder ground_truth_builder(builder);
ground_truth_builder.add_moving_object(moving_object_flatvector);
auto ground_truth = ground_truth_builder.Finish();

auto sensor_id = osi3::CreateIdentifier(builder, 1);

osi3::SensorViewBuilder sensor_view_builder(builder);
sensor_view_builder.add_sensor_id(sensor_id);
sensor_view_builder.add_global_ground_truth(ground_truth);
auto sensor_view = sensor_view_builder.Finish();

builder.Finish(sensor_view);

auto uint8_buffer = builder.GetBufferPointer();
auto size = builder.GetSize();
std::string tmp_buffer(reinterpret_cast<char const*>(uint8_buffer), size);
currentBuffer = tmp_buffer;
encode_pointer_to_integer(currentBuffer->data(),integer_vars[BASEHI_IDX],...);
```
Code Snippet 9: Generate SensorView with one MovingObject in Flatbuffers OSI

All in all, it becomes apparent, that the Flatbuffers implementation needs quite a bit more code and is more complicated than Protobuf. There is however an alternative to make coding easier: The Object API.

3.2 Object API

The Flatbuffers Object API enables an implementation with closer resemblance to Protobuf. Code Snippet 10 shows the same example as in the previous section implemented with the Object API.

```
flatbuffers::FlatBufferBuilder builder(1024);

osi3::SensorViewT currentOut;
auto sensor_id = std::unique_ptr<osi3::IdentifierT>(new osi3::IdentifierT());
sensor_id->value = 1000;
currentOut.sensor_id = std::move(sensor_id);

auto currentGT = std::unique_ptr<osi3::GroundTruthT>(new osi3::GroundTruthT());

auto veh = std::unique_ptr<osi3::MovingObjectT>(new osi3::MovingObjectT());
auto veh_id = std::unique_ptr<osi3::IdentifierT>(new osi3::IdentifierT());
veh_id->value = 10;
veh->id = std::move(veh_id);

auto base_position = std::unique_ptr<osi3::Vector3dT>(new osi3::Vector3dT());
base_position->x = 50;
base_position->y = 2;
base->position = std::move(base_position);

currentOut.global_ground_truth = std::move(currentGT);

builder.Finish(osi3::SensorView::Pack(builder, &currentOut));
```

```
auto uint8_buffer = builder.GetBufferPointer();
auto size = builder.GetSize();
std::string tmp_buffer(reinterpret_cast<char const*>(uint8_buffer), size);
currentBuffer = tmp_buffer;
encode_pointer_to_integer(currentBuffer->data(), integer_vars[BASEHI_IDX], ...);
```
Code Snippet 10: Generate SensorView with one MovingObject with the Flatbuffers Object API

The implementation is more intuitive and the order of setting the fields is more similar to Protobuf. There is however a catch: The Object API has a lower performance than the conventional Flatbuffers implementation. So, for performance critical implementations it is not recommended to be used. More information about the performance differences can be found in the performance report.

3.3 Convert Protobuf Model to Flatbuffers Implementation

To assess the expenses of converting an existing model utilizing OSI with Protobuf to a Flatbuffers implementation, a reflection-based lidar model[8] was converted. This model receives reflections obtained via ray tracing by a source FMU or a simulation tool. These reflections are sorted into beams and a thresholding and peak detection is performed. The output of the model is lidar detection data. The converted model for this report is available as a branch in the original repository[9].

The lidar model is based on the FZD OSMP Framework[10] which allows for the model to be modularized in so called strategies. The framework reads in the received SensorView and copies it to a SensorData object. The SensorData is passed as a reference sequentially to the individual strategies. This approach does not work with Flatbuffers, because fields are not mutable as described in section 2.2.3. Instead, the pointer to the SensorView buffer is directly passed to the strategy.

There are multiple differences between Protobuf and Flatbuffers when accessing data. One difference in accessing the SensorView input in the model is, that it is now represented as a pointer. Fields are accessed with the arrow notation instead of the dot notation. In addition to the missing functions described in section 2.2.2, accessing vector elements is also different. Instead of using braces like in `lidar_sensor_view->reflection(reflection_idx)`, a getter function has to be used: `lidar_sensor_view->reflection()->Get(reflection_idx)`.
Most of these differences can be converted rapidly by using find and replace function of the IDE. However, a controlled step-wise replacement is highly recommended to not replace something else unintentionally.

Writing the output of the model into FeatureData mainly consists of generating lidar detection data. This is done analogously to the examples given in section 3.1.

How time-consuming the overall conversion is highly depending on the internal data structure. In general, the fewer OSI objects are used internally, the easier the adaption to Flatbuffers. In case of the reflection-based lidar model, most changes happened in the DetectionSensing strategy[11], apart from minor changes to the framework. However, if a model relies on OSI objects as an internal data structure, the conversion will be a lot more complex.

[8] https://gitlab.com/tuda-fzd/perception-sensor-modeling/reflection-based-lidar-object-model
[9] https://gitlab.com/tuda-fzd/perception-sensor-modeling/reflection-based-lidar-object-model/-/tree/flatbuffers
[10] https://gitlab.com/tuda-fzd/perception-sensor-modeling/modular-osmp-framework
[11] https://gitlab.com/tuda-fzd/perception-sensor-modeling/reflection-based-lidar-object-model/-/merge_requests/7/diffs#diff-content-4ac61f5a3d8b115e3297e6026466c6afcb6bba30

In the Protobuf implementation it is quite tempting to use OSI objects for the internal data representation. Especially more primitive objects like osi3::Vector3d or osi3::Dimension3d are tempting to use. These data structures need to be completely recreated, when switching to Flatbuffers, creating quite a big overhead and time expense.

4 Conclusion

This report provides necessary application experience when utilizing Flatbuffers for OSI. After a short theoretical introduction to Flatbuffers, the different header files in fbs instead of proto regarding tables instead of structs are shown. This would mean to change all proto files completely and would already break backward compatibility. As a different approach, instead of switching directly from proto to fbs, one could instead install a field description format for OSI as an independent layer to separate the structuring of OSI fields from its serialization. This would allow to choose Protobuf or Flatbuffers afterwards. Nevertheless, as explained in section 2.1.2, some primitive fields like Vector3D would make sense to be structs, while others must be tables in Flatbuffers what brings more coding effort and a complete turnaround of the process in mind during coding.

A bug report regarding wrong type ordering for OSI fields is described here and started at the Flatbuffers source on GitHub, while proposing a workaround in the meantime.

To support the decision between both serialization libraries, some general implementation considerations are given. For example, the common practice to copy the whole SensorView input into SensorData output is questioned independently of its complicated implementation in Flatbuffers. Additionally, the limited support of mutable fields is shortly discussed and the changing buffer format string in Flatbuffers instead of string* in OSMP is mentioned.

However, detailed application experiences are provided for the generation of a SensorView using Flatbuffers with the DummySource example from ASAM OSI Sensor Model Packaging (OSMP). Special focus here is put on the new concept of using the FlatBufferBuilder starting from inside-out (Base to SensorView) instead of starting with the creation of the Sensorview and adding objects and then their base.

Besides, the usage of the object API that comes as an alternative for “pure” Flatbuffers implementation is explained while pointing out that while keeping the familiar coding style with this API comes with the cost of lower performance compared to using the FlatBufferBuilder.

In conclusion, the findings from the exemplary implementation of an open-source reflection-based lidar detection model can be summarized with the sentence "the fewer OSI objects are used internally, the easier it is to adapt to Flatbuffers", which shows that each model and framework needs to be investigated independently. Therefore, further investigation with other sensor models (e.g., camera/radar) and other model types (e.g., road users) is strongly recommended and must be done by all interested partners before moving to Flatbuffers. It is definitely not the simple implementation solution to all performance problems that it was promised to be.