A C++ Template Library for Data-Centric Type Modeling for DDS

Sumant Tambe, Ph.D.
Real-Time Innovations, Inc.
(sumant@rti.com)

EXECUTIVE SUMMARY

This paper describes a powerful C++ template library to allow users to describe their types in plain C++ and use those types directly for data-centric communication over DDS. The library transforms C++ types into equivalent run-time TypeObject representation as specified in the DDS-XTypes [4] standard. The library obviates the need to describe application-level data-types in external representations, such as IDL, XSD, and XML. The users of the library can use the full expressive power of native C++ to encapsulate the application-level data-types and use the same data-types for data distribution over DDS. The types may include all the standard template library containers (e.g., vector, list, map, unordered containers, etc.), raw pointers, smart pointers, and many more. The restrictions imposed by popular serialization/deserialization tools are eliminated. The application-level data are written directly using DDS. At the receiver side, the library populates application-level data structures *in-place*. No copies are necessary.

Table of Contents

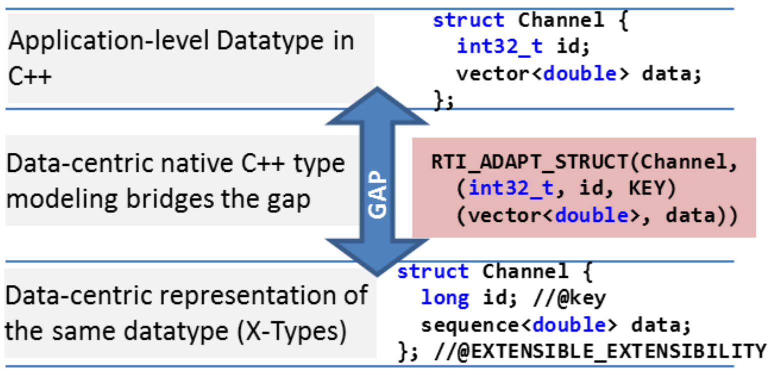
1 Background

Over the period of last two decades, the C++ users' community in the scientific domain has learned to cope with the restrictions imposed by popular tools (e.g., ROOT [1]). C++, however, has evolved significantly since and the C++11 standard [2] is now available with a number of improvements for efficiency, safety, succinctness, productivity, and simplicity. Users must be able to make use of the new language capabilities and libraries to simplify programs and improve productivity and robustness. Existing tools, however, limit what they can use from the language in describing application-level data-types. The scientific workflow frameworks often need to rely on file-based data sharing as a consequence. Therefore, the data distribution infrastructure must substantially alleviate, if not eliminate the application-level restrictions on data-types.

Data analysis applications in scientific computing are often written by teams of scientists collaborating on a scientific endeavor. To test scientific hypotheses, scientific groups often develop new data types, which need to evolve quickly as scientists refine the algorithms and data structures for data processing modules. Typically, multiple data processing modules are assembled into a workflow forming a directed graph. The data that flows through the graph is often both large in volume and complex in nature. The individual components are often built in C++ programming language for native efficiency. For example, ROOT Toolkit [1] is widely used in the scientific domain to share and store complex C++ data structures in files. The scientists strongly prefer to write their types in C++ as opposed to alternative type description languages (e.g., IDL, XSD).

Current scientific systems exchange data through in-memory object models or through files. Both models have limitations because the prior requires communicating elements of the workflow to exist in the same process memory space whereas the later requires that the data have a persistent representation. Two independently developed components running in the same process must use the exact same data structure, which tightly couples the components, hinders quick evolution of types, and delays scientific discoveries. Although some support for schema evolution is available in the existing tools, the file-based approach incurs expensive disk I/O due to use of files. Furthermore, ROOT toolkit enforces several restrictions on the data types as it supports a narrow set of features available in the C++ type system.

2 A Generic Data-Centric Type Modeling Library in C++


Application-level Datatype in C++
Data-centric native C++ type modeling bridges the gap
GAP
Data-centric representation of the same datatype (X-Types)

```
struct Channel {
  int32_t id;
  vector<double> data;
};
```

```
RTI_ADAPT_STRUCT(Channel,
  (int32_t, id, KEY)
  (vector<double>, data))
```

```
struct Channel {
  long id; //@key
  sequence<double> data;
}; //@EXTENSIBLE_EXTENSIBILITY
```

To address these challenges, we developed an innovative approach that allows scientist to use a wide range of features from the C++11 type system to define datatypes with little or no restrictions. We use the same type definition and translate it into an equivalent TypeObject representation as defined by the DDS-XTypes [4] specification. The translation is done at program compilation time using advanced C++ template programming techniques and does not require separate code generation step. The generated TypeObject is automatically registered and can be later used by DDS endpoints to communicate with each other. An example is in order.

2.1 Example from Scientific Computing

Listing 2 shows an example type definition in C++ borrowed from the Fermilab's darkart project. The example has been adapted for brevity. Left hand side in Listing 2 defines a top-level EventInfo datatype with members of other structured types. One of the nested types, `VetoTDCHits`, is an alias for C++ standard library `vector<T>` instantiated with `VetoTDCHit`. EventInfo also includes a member of enumeration type: `STATUS_FLAGS`.

The right hand side of Listing 2 shows the equivalent TypeObject synthesized at runtime based on the XTypes type system standardized by the Object Management Group (OMG). Structures and enumerations are mapped to "extensible" structures and enumerations respectively. Extensible structures and enumerations allow addition of new members retroactively and enables interoperability with old versions of the type. The "key" member `event_id` is described in the next subsection.

C++ Type Definition
(adapted from Fermilab's darkart repository)

```
struct VetoTDCHit  {
  int32_t  hit_index;
  int32_t  pmt_index;
  float    pmt_theta;
  double     pmt_phi;
};
typedef std::vector<VetoTDCHit>
VetoTDCHits;
struct VetoTruth   {
  int32_t       sim_event;
  VetoTDCHits        hits;
};
enum STATUS_FLAGS { NORMAL=0,
                    ID_MISMATCH=1,
                    BAD_TIMESTAMP=2
};
struct EventInfo  {
  VetoTruth        truth;
  int32_t       event_id;
  STATUS_FLAGS    status;
};
```

Equivalent TypeObject Synthesized at Run-time
(IDL syntax)

```
enum STATUS_FLAGS {
     NORMAL = 0,
     ID_MISMATCH = 1,
     BAD_TIMESTAMP = 2,
}; //@Extensibility EXTENSIBLE_EXTENSIBILITY

struct VetoTDCHit {
    long hit_index;
    long pmt_index;
    float pmt_theta;
    double  pmt_phi;
}; //@Extensibility EXTENSIBLE_EXTENSIBILITY

struct VetoTruth {
    long sim_event;
    sequence<VetoTDCHit, 256> hits;
}; //@Extensibility EXTENSIBLE_EXTENSIBILITY

struct EventInfo {
    VetoTruth truth;
    long event_id; //@key
    STATUS_FLAGS    status;
}; //@Extensibility EXTENSIBLE_EXTENSIBILITY
```

Listing 2: Left: original C++ type definition. Right: Equivalent TypeObject synthesized at run-time

The `std::vector<VetoTDCHit>` maps to a `sequence<VetoTDCHit, 256>`. Note that any C++ data structure with support for STL-like iterators can be mapped to a sequence. Such data structures include `std::list`, `std::map`, etc. In fact, any custom data structures are supported as long as STL-compatible iterators are available.

The TypeObject synthesis library bounds the sequence to 256 because the current implementation of the RTI DDS sequences require a statically know limit. C++ `std::vector` has no way to specify the bound statically. The TypeObject synthesis library uses 256 as the default static limit. It can be configured to be arbitrarily high (actually INT_MAX). We plan to remove the limitation of RTI DDS sequences and support truly unbounded sequences in near future.

2.2 The Design of the Generic Data-Centric Type Modeling Library

The TypeObject synthesis library is declarative in nature because no procedural code is required to enable translation of a C++ type to its equivalent XTypes TypeObject. However, the programmer must provide compile-time meta-information about the user-defined type because C++ has no built-in support for reflection at compile-time*. The meta-information is used by the TypeObject synthesis library to obtain struct name, member names, enumeration element names, enumeration ordinal values, and whether the member is a key or not. The specification of meta-information is greatly simplified using RTI_ADAPT_STRUCT macro as shown below.

C++ Type Definition (same as before)	**Meta Information Specification at Compile-time using the RTI_ADAPT_STRUCT Macro**
`struct VetoTDCHit {`	`RTI_ADAPT_STRUCT(`
`  int32_t   hit_index;`	`  VetoTDCHit,`
`  int32_t   pmt_index;`	`  (int32_t, hit_index)`
`  float     pmt_theta;`	`  (int32_t, pmt_index)`
`  double      pmt_phi;`	`  (float,   pmt_theta)`
`};`	`  (double,   pmt_phi))`
`typedef std::vector<VetoTDCHit> VetoTDCHits;`	
	`RTI_ADAPT_STRUCT(`
`struct VetoTruth   {`	`  VetoTruth,`
`  int32_t       sim_event;`	`  (int32_t,     sim_event)`
`  VetoTDCHits        hits;`	`  (VetoTDCHits,     hits))`
`};`	
`enum STATUS_FLAGS { NORMAL=0,`	`RTI_ADAPT_ENUM(`
`                    ID_MISMATCH=1,`	`  STATUS_FLAGS,`
`                    BAD_TIMESTAMP=2`	`  (NORMAL,        0)`
`};`	`  (ID_MISMATCH,   1)`
`struct EventInfo  {`	`  (BAD_TIMESTAMP, 2))`
`  VetoTruth        truth;`	
`  int32_t       event_id;`	
`  STATUS_FLAGS    status;`	`RTI_ADAPT_STRUCT(`

`};`	`EventInfo,`
	`(VetoTruth,           truth)`
	`(int32_t,       event_id, KEY)`
	`(STATUS_FLAGS,        status))`

Listing 3: Left: Original C++ type definition. Right: Specification of meta-information for RTI_ADAPT_STRUCT macro

The RTI_ADAPT_STRUCT macro expands the parameters such that the TypeObject synthesis library can make use of the meta-data. That is, the RTI_ADAPT_STRUCT macro is a substitute for the lack of compile-time reflection capability in C++.

We now describe the architecture of the declarative TypeObject synthesis library. The figure below shows the layered architecture.

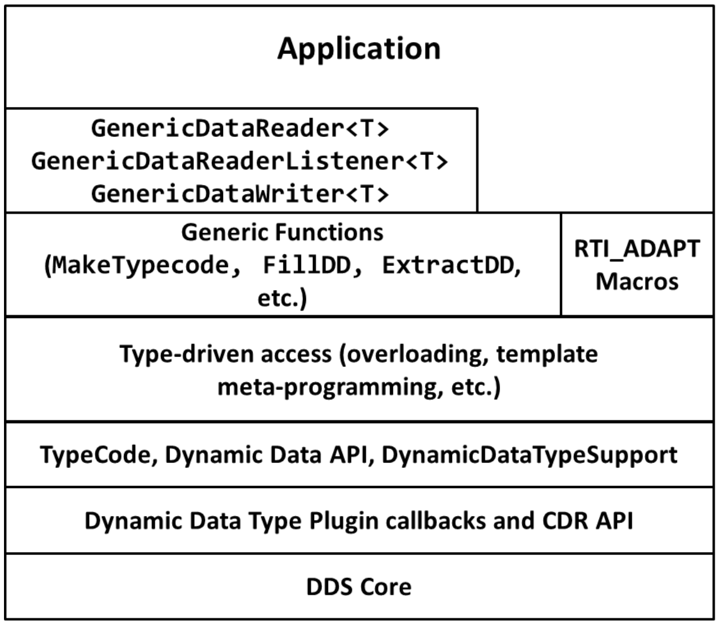

Application
GenericDataReader<T>
GenericDataReaderListener<T>
GenericDataWriter<T>
Generic Functions
(MakeTypecode, FillDD, ExtractDD, etc.)
RTI_ADAPT Macros
Type-driven access (overloading, template meta-programming, etc.)
TypeCode, Dynamic Data API, DynamicDataTypeSupport
Dynamic Data Type Plugin callbacks and CDR API
DDS Core

Figure: The layered architecture of the declarative TypeObject synthesis library

The applications use the top-level `GenericDataReader<T>` and `GenericDataWriter<T>` entities. The generic DDS entities are a thin wrapper on top of the underlying `DynamicDataReader` and `DynamicDataWriter` entities respectively. The dynamic DDS entities are designed to publish and subscribe to DDS topics of arbitrarily complex types that are not known at compile-time. When using these entities, the TypeObject and the `DynamicData` instances must be created by the programmer, which is extremely cumbersome and error-prone.

The declarative TypeObject synthesis library uses advanced C++ template programming (a.k.a. compile-time template meta-programming) that synthesizes the TypeObject and `DynamicData` objects from the user-defined C++ types and instances, respectively. `MakeTypeCode()`, `FillDD()`, and `ExtractDD()` functions provide the top-level API for this functionality. These functions use the programmer-specified meta-information (i.e., RTI_ADAPT_STRUCT) to synthesize the TypeObject and the runtime instances of the DynamicData. TypeObject and DynamicData API are part of the DDS XTypes standard.

The library supports a broad range of C++ type system features and its corresponding XTypes type system equivalent. Specifically, we support the following:

Supported features in the C++ point of view	Supported features in the XTypes type system point of view
• Fundamental types, arrays, enumerations • Struct (with public members) • Classes (with setter/getters) • Nested struct/classses • Standard Template Library (STL) `string, vector, list, set, map, array, tuple, pair`, iterators, etc. • All combinations of the above • Smart pointers • User-defined/custom containers • Lazy container adapters (e.g., boost.Iterators)	• Basic types/enumerations/strings • Arrays • Sequences of strings/basic types/enumerations • Bounded sequences/strings • Structures • Unions (including cases with defaults, multiple discriminators, and enumerations) • Optional members (sparse types) • Sequences of sequences (of sequences… and so on…) • Sequences of structures • Multidimensional arrays of strings, structures, sequences,… • Nested structures, unions

More information about the library is available at [5].

3 Embedding vs Code Generation

An alternative approach is to generate type descriptions from the existing C++ source code. Specifically, a C++ parser tool that would generate (1) type descriptions in IDL, (2) serialized representation of types using the XTypes standard, and (3) C++ code for serialization and deserialization of native C++ objects.

Our approach, however, is substantially more powerful, easier to use, easier to integrate incrementally, and also easier to maintain. Development of a C++ parser, although simplified substantially due to the Clang/LLVM tool chain, is an extremely complex endeavor. Furthermore, integrating user-written code with tool-generated code is a challenge.

Our new approach is completely library-based. Library-based approach allows programmers to use their existing source code and other third party libraries (if any) with the new declarative TypeObject synthesis library. Library based approach is substantially more extensible and maintainable compared to code generation approach.

4 Bibliography

[1] ROOT—A Data Analysis Framework. http://root.cern.ch/drupal/
[2] The C++ Programming Language; http://isocpp.org/
[3] The Data Distribution Service specification, v1.2, http://www.omg.org/spec/DDS/1.2
[4] Extensible And Dynamic Topic Types (XTypes) Specification v1.0 FTF 2, http://www.omg.org/spec/DDS-XTypes/1.0/Beta2/ Retrieved October 4, 2012
[5] Sumant Tambe; Overloading in Overdrive: A Generic Data-Centric Messaging Library for DDS; http://www.slideshare.net/SumantTambe/overloading-in-overdrive-a-generic-datacentric-library-for