Creating an HLA 1516 Data Encoding Library using C++ Template Metaprogramming Techniques

Jay Graham
Digital Fusion, Inc.
5030 Bradford Drive
Building 1, Suite 210
Huntsville, AL 35805
256-327-8211
jgraham@digitalfusion.com

Keywords:
Interoperability, HLA, RTI, HLA Evolved, Data Encoding, Encoding Helpers, C++, Template Metaprogramming

ABSTRACT: *This paper presents an HLA 1516 encoding/decoding application programming interface (API) called the Data Type Encoding Language or DTEL (pronounced 'detail'). DTEL is a domain specific embedded language written using C++ template metaprogramming techniques. DTEL is an alternative API to the proposed HLA Evolved API called Encoding Helpers.*

Like the Encoding Helpers API, DTEL is a utility API that makes it easier for federate developers to conform to the standard encoding rules defined in HLA 1516. Both APIs aim to decrease integration costs by making it easier to eliminate encoding errors that cause federate and federation crashes, hangs, and incorrect results.

In this paper, we assert that the design of the Encoding Helpers API leads to an overly general solution that fails to take advantage of host language type systems to catch encoding errors early in the development cycle. The DTEL API, by contrast, trades generality for a solution that fully leverages the C++ type system to catch many encoding errors at compile time, making these errors less costly to find and fix. Some other features of the DTEL API include:

- *Support for a declarative programming style*
- *Time and space efficiency comparable to hand-crafted code*
- *Complete type safety i.e., no error prone down-casting operations required*

This paper begins with a brief overview of the problem Encoding Helpers and DTEL are designed to help solve. It discusses the potential adverse consequences of the Encoding Helpers design, and then discusses the DTEL approach.

1. Introduction

Correct data serialization in accordance with federation agreements is critical to successful federation interoperability. Möller, Karlsson, and Löftstrand [1] identify some common problems that lead to serialization mistakes, which include

- Coding errors.
- Incorrect interpretation of the data serialization specification.
- A lack of understanding of the effects of the technical environment (e.g., data type systems, compiler options, and processor architecture) on serialization.

They note that these serialization errors lead to system crashes and hangs, invalid or misleading results, and loss of data. They point out that one of the achievements of the HLA 1516-2000 standard for the Object Model Template (OMT) specification is a set of encoding rules for all data types that are expressible within the standard [2]. In principle, these standard encodings remove all ambiguity from the question of how to properly serialize and de-serialize data for publication and subscription and should result in the elimination of many of the errors attributable to incorrect data exchange. Nevertheless, they observe that serialization errors occur more frequently than might be expected and that serialization errors increase integration risk and cost.

HLA Evolved, the next generation HLA, includes a proposal for an Application Programming Interface (API) called Encoding Helpers [1], [3]. The Encoding Helpers API is designed to address some of the problems identified by Möller, Karlsson, and Löftstrand by making it easier to construct correct 1516 serialization functions.

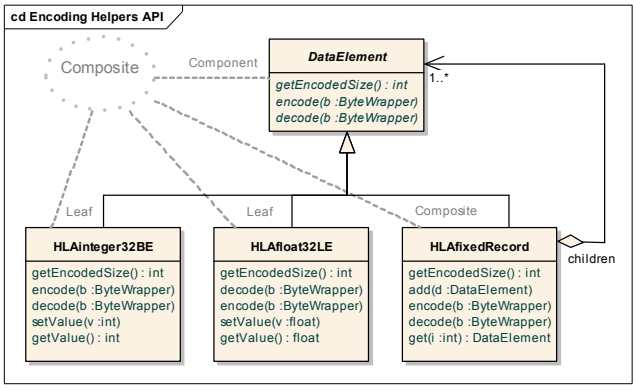
We suggest that the design of the Encoding Helpers API leads to an overly general solution that does not do

enough to exploit type systems to ensure correct 1516 encodings. As a result, developers must rely exclusively on runtime testing to find serialization mistakes, which increases development cost and risk.

This paper presents an alternative 1516 serialization API for the C++ language that uses the C++ type system to enforce 1516 datatype constraints. The API is called the Data Type Encoding Language (DTEL). Like the Encoding Helpers API, DTEL is designed to help reduce data serialization errors. DTEL enables a federate developer to express 1516 datatypes as C++ datatypes, and uses the C++ template compiler to generate correct serialization functions. DTEL reports many serialization mistakes at compile time, making these mistakes easier to find and fix, which greatly decreases development cost and risk.

2. The Encoding Helpers API

The design of the Encoding Helpers API is based on the Composite Design Pattern, where an abstract base class called DataElement defines a common interface to leaf objects representing basic 1516 datatypes (e.g., HLAinteger32BE, HLAfloat32LE, etc…) and composite objects representing constructed 1516 datatypes (e.g., a fixed record, fixed array, etc…) [4].



The Encoding Helpers API is designed to support multiple language bindings (e.g., Java and C++). Designing an API that ports easily to different languages involves compromise. Designers are usually limited to using a small number of features that are common to the set of target languages. Consequently, specific target language mechanisms that might make using an API easier, safer, and less costly may be sacrificed to support the goal of creating a portable API design.

This section presents several basic examples showing how the Encoding Helpers API is used, and then discusses some potential adverse consequences that result from the compromises made in its design.

2.1 General encoding helpers API usage

Using the Encoding Helpers API to serialize a basic datatype is straightforward. In this pseudo-code example, we are encoding a value of type HLAinteger32BE into a byte array:

```
// Create an HLAinteger32BE encoder object
encoder = new HLAinteger32BE()

// Give the encoder object a value to encode
encoder.setValue( 10 )

// Encode the value into a byte array
encoder.encode( byteArray )

// byteArray is ready for publication
```

And here is how we decode a basic data type:

```
// Reuse the encoder and byteArray objects from
// above, and assume byteArray holds an encoded
// value of type HLAinteger32BE
encoder.decode( byteArray )

// Get the decoded value
int value = encoder.getValue()
```

Serializing a constructed type is a little more complicated because it requires us to create a composite object. For example, assume we have a fixed record representation that looks like this:

Record name	Field		Encoding
	Name	Type	
Position	x	ftype	HLAfixedRecord
	y	ftype	
	z	ftype	

where `ftype` is a simple datatype of type HLAfloat32LE.

Here is the pseudo code for creating a fixed record encoder object for the Position datatype:

```
// Create an encoder object for each field
xEncoder = new HLAfloat32LE
yEncoder = new HLAfloat32LE
zEncoder = new HLAfloat32LE

// Create the fixed record encoder
posEncoder = new HLAfixedRecord

// Add the field encoder objects to the fixed
// record encoder object
posEncoder.add( xEncoder )
posEncoder.add( yEncoder )
posEncoder.add( zEncoder )

// Set the field values
xEncoder.setValue( 2.5 )
yEncoder.setValue( 3.5 )
zEncoder.setValue( 4.5 )
```

```
// Encode the record into a byte array
posEncoder.encode( byteArray )
```

And here is how we decode a Position record:

```
// Reuse the encoder and byteArray objects from
// above, and assume byteArray holds an encoded
// value of type Position
posEncoder.decode( byteArray )

// Extract the field values
float x = xEncoder.getValue()
float y = yEncoder.getValue()
float z = zEncoder.getValue()
```

Now that we have a feel for how to use the Encoding Helpers API, we look at some consequences of its design.

2.2 Type safety

Although the Encoding Helpers API reduces the need for error prone down-casting operations, it does not eliminate the need for them completely. In the fixed record example above, casting was avoided by assuming we had access to the component encoders (i.e., `xEncoder`, `yEncoder`, and `zEncoder`). If we remove this assumption, then down-casting is required. For example:

```
// Assume we have a reference to the posEncoder
// object from above, but not its component
// encoders
x = (HLAfloat32LE) posEncoder.get(0)
y = (HLAfloat32LE) posEncoder.get(1)
z = (HLAfloat32LE) posEncoder.get(2)
```

The problem here is that if the OMT definition of Position changes, then the runtime structure of `posEncoder` will change, and the code used to retrieve `posEncoder` component encoders is likely to be incorrect. Discovering these errors takes time because the host language type system (i.e., the compiler) can not catch down-casting errors. The only way to discover a casting error is by code inspection, unit testing, or debugging during system integration, all of which increase the cost of writing and maintaining code. Also, it is reasonable to expect that the probability of making a casting error increases as the complexity of the serialization object increases.

2.3 Abstraction penalty

The generality of the Encoding Helpers API makes the resulting serialization objects less efficient in both time and space than a hand-coded solution. This "abstraction penalty" is especially high for the C++ language. As a result, it is unlikely that a developer would use an Encoding Helpers based object to manage application state. Instead, a federate developer is likely to maintain a mapping between application state objects and serialization objects. The following C++ style pseudo code illustrates this idea:

```
// A POD type to hold application state data
struct PositionType{ float x, y, z }

// A PositionType encoder function
encodePosition( ByteArray b, PositionType p )
{
  HLAfloat32LE x, y, z;
  HLAfixedRecord encoder;
  encoder.add( x );
  encoder.add( y );
  encoder.add( z );

  // Align application state data with the
  // correct encoder components and encode
  x.setValue( p.x )
  y.setValue( p.y )
  z.setValue( p.z )
  encoder.encode( b )
}
```

Code will cost less to implement and maintain if functions like `encodePosition` can be avoided.

2.4 Syntax

The code to build and navigate an Encoding Helpers object can introduce a lot of syntactical noise, making the code harder to read and maintain. For example, consider what it takes to access an element of an array of Position objects.

```
// Example 2.4.1
// Assume byteArray holds an encoded array of
// Position objects and that encoder is
// constructed to properly decode byteArray

encoder.decode( byteArray )                    [1]

// Get the value of field y from the 4th record
float val =                                    [2]
((HLAfloat32LE)
  ((HLAfixedRecord)
    encoder.get(3)          // 4th record
  ).get(1)                  // field y
).getValue()
```

Something closer to this syntax would be better:

```
float val = encoder[3].y
```

The presence of so many casting operations in line 2 is a reminder that we get no help from the compiler to ensure that we are navigating the encoder structure correctly. It is easy to imagine programmer intent getting lost in the mass of object instantiations, casting operations, and intermediate object declarations required to deal with even moderately complex serialization objects.

Code that is difficult to read costs more [5].

2.5 Serialization object mutability

Serialization objects are mutable at runtime. Once a serialization object is constructed per a 1516 datatype definition, there is no way to "freeze" the object's structure to prevent a program from inadvertently altering it. This also means that decoder objects must be told what type of objects to create during de-serialization via a factory object. Careful programming and the judicious use of creational patterns (e.g., the Factory pattern) is the best a developer can do to encapsulate the runtime structure of a serialization object. At the very least, this implies that additional coding and testing must be done to ensure the runtime integrity of serialization objects.

2.6 Too many compromises?

As the preceding examples demonstrate, the Encoding Helpers API makes compromises in expressiveness, efficiency, and programming safety by largely ignoring host language type systems in favor of greater language neutrality. In particular, no attempt is made to use generic programming techniques to enforce correct 1516 datatype semantics. This is understandable, because type systems and support for generic programming vary widely among programming languages. It is also regrettable given the nature of the problem the Encoding Helpers API is designed to solve.

Implementing correct serialization is tedious, detail-oriented work. This makes it especially unfortunate that the Encoding Helpers API does not leverage the strong type checking systems offered by languages like C++ and Java to find serialization errors earlier in the development cycle. Detecting serialization errors at compile time can significantly reduce the cost of implementing and maintaining correct 1516 data serialization code.

3. The DTEL API

The DTEL API trades language portability for greater programming safety, runtime efficiency, and expressiveness. DTEL uses generic programming techniques that leverage the C++ type and template systems to compute correct 1516 datatype serialization functions at compile time (more on what this means in sections 4 and 5).

This section introduces the DTEL API with examples to give a feel for what coding with DTEL looks like and to see how the DTEL API differs from the Encoding Helpers API. This section ends with a discussion of how the DTEL approach addresses the issues raised in section 2.

3.1 General DTEL API usage

Here is what encoding an HLAinteger32BE looks like in DTEL:

```
// Declare a byte array container
dtel::byte_array byteArray;

// Declare and initialize the HLAintger32BE
// value to be encoded
dtel::HLAinteger32BE value = 5;

// Encode
dtel::encode( byteArray, value );

// byteArray is ready for publication
```

This code is very similar to the HLAinteger32BE encoding example in section 2.1. The important difference is that the DTEL API does not use objects to represent 1516 basic datatypes – it uses C++ datatypes. In fact DTEL represents 1516 basic datatypes using built-in C++ types. As a result, 1516 basic datatype representations in DTEL are as efficient as built-in C++ types and support the same operations.

Serializing a constructed type is more complicated. Recall the definition of the Position fixed record from section 2. Here is how we translate Position into an equivalent DTEL datatype, and then encode it:

```
// Define ftype
typedef HLAfloat32LE ftype;

// Declare fixed record field types
struct x : dtel::field< ftype >{};
struct y : dtel::field< ftype >{};
struct z : dtel::field< ftype >{};

// Define the fixed record type
typedef dtel::fixed_record< x, y, z >
  PositionType;

// Declare an instance of PositionType,
// initialize it, and encode it
PositionType pos;

pos.field< x >() = 1.5f;
pos.field< y >() = 2.5f;
pos.field< z >() = 3.5f;

dtel::byte_array byteArray;
dtel::encode( byteArray, pos );

// byteArray is ready for publication
```

Compare this code to the Encoding Helpers fixed record example in section 2.1. Here again, the big difference is that the DTEL code replaces serialization objects with C++ datatype definitions.

Here is a fixed array of `PositionType` records with 5 elements:

```
// Example 3.1.1
dtel::fixed_array<PositionType, 5> PosArrayType;
```

The [] operator is used for element access. For example:

```
// Example 3.1.2
PosArrayType posArray;                    [1]

// Set a value of the 3rd element
posArray[2].field< x >() = 1.5f;          [2]
dtel::encode( byteArray, posArray );      [3]
```

Here is a variable array of PositionType records:

```
// Example 3.1.3
dtel::variable_array<
  std::vector< PositionType > > VarPosArrayType;
```

In this example, we use a standard template library (STL) vector to manage our variable length container of PositionType records. We are free to use any container with the same abilities as std::vector.

Elements are added to a variable array using the push_back method. For example:

```
// Example 3.1.4
VarPosArrayType varPosArray;

// Add an element
varPosArray.push_back( PositionType() );
```

Arbitrarily complex types can be created by nesting DTEL types. For example, here is a 5 by 5 matrix of PositionType records:

```
// Example 3.1.5
// Reuse PosArrayType from above
dtel::fixed_array<PosArrayType,5> PosMatrixType;
```

Here is how we access elements of the matrix:

```
// Example 3.1.6
PosMatrixType posMatrix;                  [1]
posMatrix[1][2].field< z >() = 4.5f;      [2]
dtel::encode( byteArray, posMatrix );     [3]
```

Variant records are also supported by DTEL, but their syntax is a little more complicated. In order to save space, we do not show a variant record example.

Now that we have a feel for how to use the DTEL API, we look at how the DTEL approach addresses the issues from section 2.

3.2 Type safety

DTEL eliminates the need to perform down-casting operations. As we shall see when we explore DTEL's implementation in sections 4 and 5, all casting operations in DTEL are up-casts, so no type information is lost when retrieving elements of a constructed type. For instance, line 2 from example 3.1.2:

```
posArray[2].field< x >()
```

results in a reference to a value whose type is HLAfloat32LE.

In effect, DTEL aligns the C++ type system with the 1516 type system so that C++ type violations become 1516 type violations and are flagged at compile time. For example, if the datatype of field z of the Position fixed record is changed from HLAfloat32LE to HLAoctet, then line 2 from example 3.1.6

```
posMatrix[1][2].field< z >() = 4.5f;
```

becomes a compile time warning about a possible loss of data due to a conversion from float to unsigned char.

With DTEL, a federate developer is free to modify and change 1516 OMT datatype definitions and their corresponding DTEL definitions without having to worry about accidentally introduction down-casting errors. This makes code easier to maintain and eliminates the need for code inspections or unit tests designed to catch down-casting errors.

3.3 Abstraction penalty

DTEL has a near-zero abstraction penalty. There are no runtime costs associated with defining DTEL based 1516 datatype representations and serialization functions, and DTEL datatypes tend to be the same size as equivalent, general purpose C++ datatypes. Compiler optimizations and function inlining can result in serialization functions that are as efficient as hand-coded solutions.

With DTEL, a C++ federate developer can choose to use the same objects to manage application state and serialization, eliminating the need for functions like encodePosition from section 2.3.

3.4 Syntax

We assert that the DTEL syntax supports a straightforward translation from 1516 OMT datatype definitions to DTEL definitions. The DTEL syntax is concise, especially when compared to the Encoder Helpers API and particularly when accessing serialization components. For example, compare example 2.4.1 to its DTEL equivalent given here:

```
// encoder is a DTEL fixed array of PositionType
// fixed records declared like this:
// dtel::fixed_array<PositionType, 5>
```

```
// Get the value of field y from the 4th record

HLAfloat32LE &val = encoder[3].field< y >();
```

This comes pretty close to our ideal syntax. Also note that field access is by name and not by a numeric index. A developer can change a fixed record's field order without breaking the code used to access the fields of that record.

3.5 Serialization object mutability

DTEL datatypes are C++ datatypes, so their definitions are fixed at runtime. No special care is needed to insure the runtime integrity of a DTEL datatype.

Also, there is no need to use special creational patterns to inform decoder functions what type of objects to create during the decoding process.

3.6 Fewer compromises

The DTEL API trades language portability for fewer compromises in the areas of programming safety, runtime efficiency, and expressiveness.

Next, we discuss how DTEL eliminates the compromises of the Encoding Helpers API by examining DTEL's implementation. Section 4 introduces the C++ programming techniques that are the technical foundation for DTEL's implementation. Section 5 is a design walkthrough used to demonstrate how we arrived at DTEL's API syntax.

Note: understanding the material in sections 4 and 5 is not a prerequisite to effectively using the DTEL API.

4. DTEL Technical Foundations

DTEL is a very different API from the Encoding Helpers API, because DTEL replaces runtime constructed serialization objects with C++ datatype definitions and uses the C++ template compiler to generate correct 1516 serialization functions from those datatype definitions.

This section presents a cursory overview of the C++ programming concepts that enable DTEL to perform much of its work at compile time. Two main ideas are discussed: C++ template metaprogramming and domain specific embedded languages (DSEL, not to be confused with DTEL).

4.1 C++ Template metaprogramming and DSELs

DTEL uses C++ template metaprogramming to shift all the work of computing correct 1516 serialization functions from runtime to compile time [6].

A metaprogram is a program that manipulates code. A parser generator like YACC is a common example of a metaprogramming environment. YACC takes a metaprogram written in a parser description language and executes the metaprogram to produce C/C++ code. The parser description language is called the domain specific language (DSL) and the C/C++ code is called the host language.

The goal of a DSL is to allow a developer to express solutions using abstractions from a problem domain, resulting in code that is more concise, easier to write, and maintain.

YACC is an example of a metaprogramming environment where the DSL and host language differ. Therefore, YACC requires a developer to learn two different languages and it complicates the build system by requiring an extra step to invoke a DSL to host language translator program.

C++'s support for generic programming provides a powerful metaprogramming environment for defining DSLs in the host language. Consequently, a developer can create a DSL, code with it, and perform mainstream programming all within the C++ environment. The developer does not have to learn a new language to use a C++ based DSL and no additional build step is required. A C++ based DSL is called a domain specific *embedded* language, because the DSL is implemented in the host language.

DTEL is a DSEL.

4.2 C++ template metaprogramming/DSEL example

This section presents a simple C++ template metaprogram, based on an example from [6]. This example gives a feel for what metaprogramming in C++ looks like and how it is used to define a DSEL.

We want a declarative language that lets us represent integer literals using binary numbers. In other words, we want a statement like the following to interpret the numerals `1001` as an unsigned binary number and assign it to the variable `nine`, like this:

```
unsigned nine = binary(1001);
```

Our domain is unsigned binary number representation, and our DSEL definition looks like this:

```
// File: binary.hpp
template <unsigned long N>
struct binary {
  static unsigned const value
    = binary<N/10>::value << 1 | N%10; };
```

```
// Specialization terminates recursion
template <>
struct binary<0> {
  static unsigned const value = 0; };
```

Here is how we use our DSEL:

```
#include "binary.hpp"
int main() {
  unsigned five = binary<0101>::value;
  unsigned nine = binary<1001>::value;
}
```

Covering everything that is going on here is beyond the scope of this paper, so here are the highlights (for more information see [6] and [7]):

- C++ template metaprograms execute as the result of instantiating C++ templates (hence the name). The metaprogram in this example executes when the `::value` member of `binary<N>` is accessed. This causes the `binary<N>` template to be instantiated with the given value for N, which in turn causes another instantiation of `binary<N>` with a smaller value for N, until N reaches zero and then the `binary<0>` specialization terminates the recursion.

- C++ template metaprogramming is functional programming. C++ metaprograms are limited to using a compile time subset of C++ language features. As a result, all metadata is immutable, and all metafunctions have no retained state (i.e., no side effects).

- Recursion is idiomatic for C++ template metaprograms. Because metatdata is immutable, it is impossible to write a loop that terminates by examining some mutable state.

- Template specialization is used to perform conditional branching. The C++ template compiler selects the best matching template specialization when it instantiates a template. Therefore, template specialization can be used to conditionally select between alternatives. In our example above, template specialization is used to select an alternate version of the `binary<N>` template when N is zero.

The example of this section only demonstrates how a C++ template metaprogram can perform compile time numeric computations. As we shall see, the real power of C++ template metaprogramming comes from the ability to compute with C++ datatypes.

4.3 The boost metaprogramming library (MPL)

As the previous example illustrates, C++ template metaprogramming can be challenging. The syntax is awkward, functional programming is different from mainstream object-oriented programming, and C++ template metaprogramming does not enjoy the same level of tool support as conventional C++ programming. Fortunately there is help in the form of a library of tools called the Boost Metaprogramming Library (MPL) [7].

MPL is used extensively in the implementation of DTEL.

Generally, MPL is to C++ template metaprogramming as the standard template library (STL) is to general purpose C++ programming [8].

(MPL is one of many high quality, peer reviewed, open source C++ libraries distributed by the Boost organization at http://www.boost.org)

5. DTEL Design Walkthrough

This section walks through the process of designing and implementing the parts of DTEL that support building correct 1516 fixed record serialization functions using the concepts and techniques introduced above. Note that DTEL supports all 1516 types - basic, simple, enumerated, and all constructed types - but this section focuses mainly on DTEL's support for fixed records. Also note that DTEL users are not required to reconstruct the design presented here, or even understand it.

We start by identifying the key abstractions of our domain. Then we discuss how we arrived at the design of DTEL's syntax to express our domain abstractions. And finally, we implement the C++ template metaprogram necessary to process DTEL statements to create correct and efficient 1516 fixed record serialization functions.

5.1 Domain abstractions

The elements of our domain come directly from [2]. We repeat some of them here for convenience.

Basic Datatype (i.e., Basic Data Representations) - are fundamental datatypes used to define all other OMT datatypes. These datatypes are characterized by their size in bits, interpretation, multi-byte ordering (i.e., big or little endian), and bit-oriented encoding rules.

Simple Datatypes - define scalar data items. Simple datatypes have a basic datatype representation which completely determines the simple datatype's encoding.

Fixed Record Datatypes - describe a vector of heterogeneous datatypes called fields, where each field has an OMT datatype. Fixed record datatypes are also characterized by an encoding rule that describes how the

fields are ordered and aligned. Fields are encoded according to their datatype.

Octet Boundary - a datatype's octet boundary is used to compute its proper alignment within a constructed type.

HLAfixedRecord - the 1516 standard encoding for fixed record datatypes. Fields are encoded in the order they are listed. The first field is at offset 0. Zero or more padding bytes are added after each field (except the last field) to ensure the next field is aligned with the smallest offset that is a multiple of its octet boundary. Here is a fixed record padding example, where HLAboolean's octet boundary is 4:

Field 0				**Field 1**			
Byte				**Byte**			
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
HLAoctet	0	0	0	HLAboolean			

5.2 DTEL design goals

Our objective is to design a DSEL that lets users express 1516 datatype representations directly in their C++ applications. In sections 2 and 3, we discussed many of the issues we want our API to address. For clarity, here is a more formal statement of our design goals. We want DTEL to be:

1. Type Safe - We want to eliminate the need to perform error-prone, down-casting operations. We want to use the C++ compiler to catch encoding mistakes early in the development cycle.

2. Efficient – We want our solution to be efficient enough to give users the option of representing application state data using DTEL datatypes, thus eliminating the need to maintain separate state data and serialization objects. At the very least, users should not have to sacrifice runtime efficiency for a good, clean abstraction of the problem domain, particularly in a language like C++ that is specifically designed to support abstractions with minimal runtime costs.

3. Declarative – We want users to express 1516 datatypes using simple, declarative statements that assert what needs to be serialized, not how it is to be serialized.

4. Expressive - It should be easy to translate 1516 datatype representations into DTEL representations.

5. Maintainable - A simple change to a 1516 datatype representation should require only a simple change to its equivalent DTEL representation.

6. Extensible - The 1516 standard allows for user defined basic data representations and user defined encoding policies. DTEL should allow new basic data representations to be added with ease and for standard 1516 encoding policies to be replaced by user defined encoding policies.

7. Scalable - We want our solution to support large HLA object models with many different datatype representations, as well as large, complex, constructed datatype representations.

5.3 DTEL syntax

Our challenge is to design the right syntax for our DSEL that allows users to express 1516 datatypes as C++ datatypes. A good, clean syntax will go a long way to achieving design goals 3, 4, and 5.

First we must decide how to represent 1516 basic datatypes. Ideally we would use built-in C++ types as is, but this does not work because built-in C++ types do not carry all the information we need to generate correct serialization functions. The answer is to wrap a built-in type T inside of a new type that carries the information we need. The new type can be made to behave just like T, so we do not lose efficiency or type safety. Wrapping types like this is a generic programming technique known as adding type traits [9], [10].

Here is how we represent basic data in DTEL using type traits:

```
template< typename T,
          int Size_In_Bits,
          typename Endian >
struct basic_data {
  typedef Endian endian;
  typedef mpl::int<Size_In_Bits> size_in_bits;
  typedef T value_type;
  typedef compute_octet_boundary<
    Size_In_Bits
  >::type octet_boundary; };
```

Explaining everything that is going on here is beyond the scope of this paper. The important point is that we now have a way to declare types that carry all the 1516 basic data traits (bolded) we need to enforce standard encoding policy and that these traits are available at compile time.

With this template in hand, we can approximate a 1516 basic data representation table, where each "row" in the table is a C++ typedef. For example, here is the HLAfloat32BE entry:

```
typedef basic_data<
  float, 32, endian::big
> HLAfloat32BE;
```

The translation from HLA OMT representation to this representation is straightforward, which meets design goal 4. This implementation is also extensible. For example, we can add our own basic data representations like this:

```
typedef basic_data<
  unsigned short, 16, endian::little
> MyUnsignedShort;
```

Here we have added a basic datatype that represents a little endian, unsigned 16 bit integer.

Representing a simple datatype typically involves creating an alias for a basic type. For example:

```
typedef HLAinteger32LE LinearMeasureType;
```

Now that we have a way to represent basic and simple datatypes, how do we represent fixed records? The most appealing answer is to use C-style Plain-Old-Data (POD) `struct` types, like this:

```
struct PODType { HLAoctet f1; HLAfloat32BE f2; }
```

Unfortunately, this does not work because the C++ compiler can not dissect `PODType` to determine the number, types, and order of its fields, all of which are needed to compute correct 1516 serialization functions. We need is a way to create a `struct` that allows for compile time interrogation of its fields.

We start by observing that a `struct` is just an ordered sequence of types. The compiler can not insert fields into a `struct` directly, but we can use template metaprogramming techniques to get the compiler to create a `struct` using derivation. To see how this is done, we need a few helper templates.

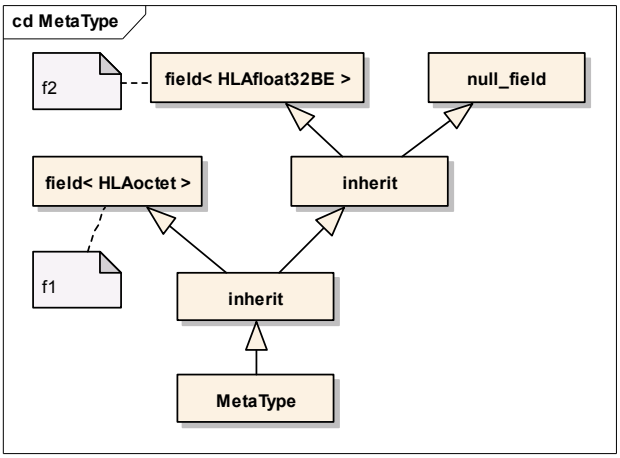
```
// Template to construct fields of type T
template< typename T >
struct field {
  T value;
  typedef T value_type; };

// Template to compose fields via derivation
template< typename T, typename U >
struct inherit : T, U {};

// Template to terminate construction
struct null_field {};
```

Our goal is to get the compiler to execute a metaprogram that takes a type sequence like < HLAoctet, HLAfloat32BE > as input and produces a `struct` type like the following as output:

```
struct MetaType :
  inherit<field<HLAoctet>,       // f1
    inherit<field<HLAfloat32BE>, // f2
              null_field > > {};
```

Notice how the structure of `MetaType` is built up through the recursive application of the `inherit` template.

A UML diagram makes things a little clearer.



An important question is how do we access the fields of a `struct` built in this way? The thing to observe is that type names replace field names. For example, consider how we access the value of the `HLAoctet` field (i.e, the equivalent of field `f1` in `PODType` above).

```
MetaType record;
HLAoctet &f1Value =
  static_cast< field<HLAoctet> &  >
  (record).value;   // Up-casting is ok
```

We must up-cast to `field<HLAoctet>` to access field "f1". Accessing fields like this is type safe, because the compiler can check up-casts for correctness.

We can make field access less awkward by reintroducing field names like this:

```
struct f1 : field<HLAoctet> {};
struct f2 : field<HLAfloat32BE> {};
```

Now `MetaType` becomes:

```
struct MetaType:
  inherit< f1,
    inherit< f2, null_field > > {};
```

And field access becomes:

```
HLAoctet &f1Value =
  static_cast<f1 &>(record).value;
```

This field access syntax is too ugly to be considered expressive and maintainable. We can clean it up by hiding it behind a function member template like this:

```
// Member of MetaType
template< typename F > F::value_type &field() {
  return( static_cast< F & >(*this).value );
}
```

And now field access is:

```
HLAoctet &f1Value = record.field< f1 >();
```

which is the syntax DTEL supports.

The next question is can the compiler construct correct 1516 fixed record serialization functions from a datatype like `MetaType`? The answer is yes because as the `inherit` template is instantiated, it gives the compiler a place to insert a function that can operate on adjacent `field` members, like this:

```
// T and U represent adjacent fields
template< typename T, typename U >
struct inherit : T, U
{
   static void encode( byteArray, inherit &rec )
   {
      // Encode field T's value
      byteArray.encode( rec.field<T>() );

      // Align T with U by computing and
      // encoding the padding between T and U
      encode_pad<T,U>( byteArray );

      // Encode U's value - this is a recursive
      // call up the derivation chain
      U::encode( byteArray, rec );
   }
};
```

Expanding the definition of the `inherit` template as shown above results in an `encode` function that is created by traversing the `field` members of a structure created through the recursive application of the `inherit` template. Invoking the `encode` function encodes `T's` value, computes and encodes alignment bytes between `U` and `T`, and then encodes `U's` value. In this way, the entire structure is encoded in correct field order.

To complete this solution, we need to terminate the recursion. The last value of our input type sequence is always `null_field`. Therefore, we must give `null_field` an empty implementation of the `encode` function.

```
struct null_field {
  static void encode(byteArray, inherit &){} };
```

Now, when U equals `null_field`, calling `U::encode` does nothing.

Decoding is just doing the reverse of encoding.

5.4 DTEL fixed_record metaprogram

How can we get the compiler to generate `struct` types like `MetaType`? Fortunately, MPL gives us the tools we need.

MPL lets us define a type sequence like this:

```
mpl::vector< int, float, char >
```

MPL also provides a rich set of template metafunctions for manipulating type sequences in the same way that the STL provides a rich set of algorithms for processing collections of objects.

Using MPL's support for type sequences, we can write a metaprogram that recursively applies our `inherit` template over a type sequence to generate a `struct` type with the structure and behavior we want. Here is a pseudo code solution that explains how this works.

```
// A metaprogram that constructs 1516 fixed
// record representations

template< typename S >
struct fixed_record :
  mpl::reverse_fold<
    S,                  // type sequence
    null_field,
    inherit< _2, _1 >
  > {};
```

This template generates a `struct` from the type sequence `S` using a series of derivations under the control of a reverse-fold algorithm. The reverse-fold builds a derivation chain by applying the `inherit` template to the elements of `S`, where `null_field` terminates the derivation chain. For example, given:

```
struct f1 : field<HLAoctet> {};
struct f2 : field<HLAfloat32LE> {};
struct f3 : field<HLAinteger16BE> {};

mpl::vector<f1, f2, f3> // sequence S
```

then invoking (at compile time)

```
mpl::reverse_fold<
  S, null_field, inherit<_2, _1> >
```

results in a type that is structured like this:

- Fold 1 (first application of `inherit`)

```
inherit< f1,
  reverse_fold< <f2, f3>,
    null_field, inherit<_2, _1> > >
```

- Fold 2 (second application of `inherit`)

```
inherit< f1,
  inherit< f2,
    reverse_fold< <f3>,
       null_field,
       inherit<_2, _1> > > >
```

- Fold 3 (final application of `inherit`)

```
inherit< f1,
  inherit< f2,
    inherit< f3, null_field > > >
```

After fold 3, the type that results has the structure we want. Although this is a very exotic type, it meets C++'s definition of a POD type and is essentially equivalent to:

```
struct fixed_record {
  unsigned char f1;         // HLAoctet
  float         f2;         // HLAfloat32LE
  short         f3;         // HLAinteger16BE
};
```

Techniques similar to the ones shown here are used to generate correct serialization functions for the other 1516 constructed types. Note, however, that a user is not required to understand these metaprogramming mechanics when using the DTEL API.

6. Analysis

Have we achieved our design goals? Section 3 already contains some discussion about how DTEL meets our design goals; we continue our analysis here for completeness.

6.1 Design goals analysis

Type Safe - As we said in section 3, DTEL eliminates the need to perform down-casting operations under all circumstances.

Declarative -The DTEL syntax is purely declarative. All of the procedural logic for correct 1516 serialization is hidden in DTEL's implementation.

Efficient - DTEL has a near-zero abstraction penalty (see section 3).

Expressive – The DTEL syntax supports a straightforward translation from 1516 OMT datatype representations to DTEL representations (see section 3 examples).

Maintainable – Simple changes to 1516 representations require only simple changes to their corresponding DTEL representations.

Extensible – Earlier we demonstrated how easy it is to add new basic data representations to DTEL. We also said we wanted to be able to replace default 1516 encoding behavior with user defined behavior. As of this writing, DTEL does not support user defined encodings. However, it should be possible to support user defined encoding rules similar to the way the STL supports user defined allocators.

Scalable – Developers are free to use all of the mechanisms offered by C++ for managing large-scale software development e.g., namespaces, scoped typed definitions, etc…

Some other benefits of DTEL's design not covered in section 3 are:

- Byte array buffers for fixed sized types can be stack allocated, because the size of a fixed sized type is computed by DTEL at compile time.

- DTEL supports user defined serialization formats like XML, for example.

- DTEL's deep interaction with the host language allows for customizations that blend DTEL functionality with the functionality provided by other libraries. For example, a DTEL variable array declared like this:

  ```
  dtel::variable_array< std::string > StrType;
  ```

  treats a variable of type `StrType` just like a `std::string` variable, but encodes it as a 1516 variable length array of HLAoctets.

6.2 DTEL design tradeoffs

DTEL's design comes with it own set of tradeoffs and compromises. Here is a summary of the most important ones.

Longer compile times - Compiling DTEL code will increase compile time and the amount of memory used by the compiler due to extensive template instantiations. As of this writing there has been no quantitative analysis of the effect of using DTEL on compile times, but it is estimated that using DTEL may increase compile times by 5 to 10 percent.

Larger object files - The object files generated from code that uses DTEL can be significantly larger when compiled with debug information due to the large symbols generated by the MPL. We have no quantitative analysis but we estimate object files to be 2 to 5 times larger.

Complex debug symbols - Debugging DTEL based code is more challenging due to the long and deeply nested debug symbols generated by the MPL.

"Scary" template compiler error messages - Template compiler messages resulting from the incorrect use of DTEL can be long and intimidating and can discourage a user. However, with a little practice it is possible to learn how to interpret template compiler messages to quickly diagnose problems (for more information, see [10]).

Portability - DTEL is written using standard C++ and should work with any standard conforming C++ compiler. DTEL is tested on these platforms:

- Microsoft visual C++ version 8 (Windows)
- GNU gcc version 3.4.4 (Windows and Linux)

As a general rule, it is likely that DTEL will port to any platform that supports the MPL. MPL is supported on variety of operating systems (e.g., Linux, Sun Solaris, and Windows) and compilers (e.g., GNU gcc, Visual C++, Intel, Borland, and sunpro). See http://www.boost.org/status/compiler_status.html, for more information.

7. Conclusions

Correct data serialization is a simulation-interoperability imperative. Data serialization errors can be difficult to find and fix which leads to increased development risk and cost.

Recall the common mistakes that lead to serialization errors identified by [1]:

1. Coding errors.
2. Incorrect interpretation of the data serialization specification.
3. A lack of understanding of the effects of the technical environment (e.g., data type systems, compiler options, and processor architecture) on serialization.

Both the Encoding Helpers API and the DTEL API do a good job of eliminating mistakes 2 and 3 by hiding 1516 serialization rules and technical environment details behind programming interfaces. We assert, however, that the DTEL API does a better job of eliminating coding errors (mistake 1), and comes with additional benefits.

The Encoding Helpers' overly general design creates new opportunities for coding errors. Its lack of type safety requires the use of error prone down-casting operations. More code must be written to compensate for its abstraction penalty and runtime mutability which, generally, means more opportunity to get things wrong. Finally, its awkward syntax leads to code that is difficult to read which, again, means more opportunity for error.

DTEL offers the C++ federate developer an alternative 1516 encoding API in the form of a DSEL that makes fewer compromises, particularly in the areas of static type safety, expressiveness, and runtime efficiency. DTEL lets a developer work at the same level of abstraction as the Encoding Helpers API, but with a near zero abstraction penalty and greater programming safety, which results in federate code that is less costly to write, test, maintain, and integrate with an HLA federation.

8. References

[1] Björn Möller, Mikael Karlsson, Björn Löfstrand: "Reducing Integration Time and Risk with the HLA Evolved Encoding Helpers", 2006 Spring Simulation Interoperability Workshop Proceedings, 06S-SIW-042.

[2] IEEE: "IEEE Standard for Modeling and Simulation (M&S) High Level Architecture (HLA) – Object Model Template (OMT) Specification", IEEE Std 1516.2-2000, www.ieee.org.

[3] HLA Evolved, www.sisostds.org.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides: Design Patterns Elements of Reusable Object-Oriented Software, Addison-Wesley, Boston 1995.

[5] Steve McConnell: Code Complete a Practical Handbook of Software Construction, Microsoft Press, Redmond Washington, 1993.

[6] David Abrahams, Aleksey Gurtovoy: C++ Template Metaprogrammin Concepts, Tools and Techniques from Boost and Beyond, Addison-Wesley, Boston 2004.

[7] Boost C++ Libraries, www.boost.org.

[8] Nicolai M. Josuttis: The C++ Standard Library A Turorial and Reference, Addison-Wesley, Boston 1999.

[9] Andrei Alexandrescu: Modern C++ Design Generic Programming and Design Patterns Applied, Addison-Wesley, Boston 2001.

[10] David Vandevoorde, Nicolai M. Josuttis: C++ Templates The Complete Guide, Addison-Wesley, Boston 2003.

Author Biography

JAY GRAHAM is a Sr. Principal Engineer at Digital Fusion, Inc. Huntsville, AL. Mr. Graham received a BSCS from Wichita State University in 1987.