Shared Data Types for OSRA and TASTE using Modern C++

Jan Sommer
Institute for Software Technology
DLR (German Aerospace Center)
Lilienthalplatz 7, 38108 Braunschweig
jan.sommer@dlr.de

Andreas Gerndt
Institute for Software Technology
DLR (German Aerospace Center)
Lilienthalplatz 7, 38108 Braunschweig
andreas.gerndt@dlr.de

Daniel Lüdtke
Institute for Software Technology
DLR (German Aerospace Center)
Lilienthalplatz 7, 38108 Braunschweig
daniel.luedtke@dlr.de

***Abstract*—The European Space Agency (ESA) currently provides two tools for the modeling of on-board software: The Assert Set of Tools for Engineering (TASTE) and the On-Board Software Reference Architecture (OSRA).**

For data type modeling, TASTE uses the standardized Abstract Syntax Notation One (ASN.1), while OSRA provides an internal eCore-based data type representation. Unfortunately, the interworking between the two frameworks lacks a mechanism to exchange data easily without duplicating the data type information.

In this work, we present our approach for the exchange of data types and data values between software developed with both tools. We show our additions to the OSRA infrastructure enabling the exchange of data types between OSRA and TASTE based on the same data type descriptions in ASN.1. This includes complementing the OSRA editor with the ability to read and write ASN.1 data type descriptions and to specify the data type encodings in TASTE's ASN.1 Control Notation.

Our previous implementation of the ASN.1 data types in Modern C++ has been extended with a prototypical implementation for the serialization of the data types compatible with TASTE's ACN encoded types.

As for the data types themselves, C++ metaprogramming techniques have been used for the encoder. This allows us to keep the code generators simple and maintainable. Some early results on the exchange of data between OSRA, enabled with our prototype generator, and the TASTE framework with its own ASN.1 compiler are presented and discussed.

I. Introduction

The demand for more complex on-board functionalities for future spacecraft continues to rise, increasing the burden on often small software teams. To handle such a complexity, model-driven methodologies can help to capture the overall architecture and design of the software. In a later step, they also allow auto-generating source code and documentation artifacts from the model, thereby relieving software developers from monotonous tasks.

In order to support model-based software development, the European Space Agency (ESA) provides The Assert Set of Tools for Engineering (TASTE) [1] and the On-Board Software Reference Architecture (OSRA) [2]. Both share some common design concepts like separation of concerns, component-based modeling and graphical tooling for the design tasks. OSRA targets mostly the design of spacecraft on-board software. At the same time, it leaves the concrete implementation of the code generators to the entity using OSRA. TASTE, on the other hand, provides a more generic framework, includes code generators for the C and Ada languages, and has been also applied in robotics applications [3]. Unfortunately, the interworking between the two frameworks lacks a mechanism to exchange data easily without duplicating the data type information. Our goal is to eliminate this problem and use the same data type descriptions across both frameworks and also have a compatible binary representation for the data types to allow seamless data exchange between applications developed using TASTE or OSRA.

In this paper, we first have a short comparison between the existing data type definition methods in OSRA and TASTE in Section II. In the following section, we present how we integrated support for ASN.1 and ACN to the OSRA model editor. In Section IV, we give a short overview of the data type system we developed previously [4] and which will be the foundation for the encoding infrastructure. Section V presents some of the concepts and implementation details behind our new ACN encoding prototype. In Section VI we discuss the current state of our prototype and its achieved results. Finally we give our conclusions and ideas about the future direction of our work in Section VII.

II. Data Type Definition in OSRA and TASTE

In complex software projects, data is exchanged by a plethora of software modules, potentially developed by different software teams. To ensure safe data exchange inside a single as well as across many different modules, essentially three basic problems need to be solved: First, there has to be a common source defining the data types, ideally with the option to define constraints. Secondly, there needs to be some way to determine whether a value is valid or not. Finally, a shared understanding about how to encode values of these data types for exchange across modules or even devices is necessary.

TASTE solves the first problem by describing its data types through the programming language agnostic Abstract Syntax Notation One (ASN.1) notation. The second one is addressed by generating test routines for validity checks during runtime. The last problem is solved by implementing encoding rules of ASN.1 or by using a custom description for the encoding

of data types using TASTE's own ASN.1 Control Notation (ACN).

OSRA provides mainly its metamodel, the Space Component Model (SCM), and a corresponding editor which also provides a modeling workflow for the user. Of the three mentioned problems, OSRA currently solves only the first problem through its graphical data type editor. We only present a short summary of OSRA's type modeling system here. A more in-depth analysis was carried out by us in [4].

The data type part of the OSRA metamodel has a complex class hierarchy with several layers of abstract classes, which are used in other parts of the metamodel to refer to generic types.

It provides common numerical metatypes for integer and floating point numbers and also allows to constrain the value space of those types. Other supported elementary metatypes include representations for boolean types, enumerations, and fixed point values. The OSRA metamodel also allows the modeling of several kinds of array and string types as well as common structured and union types.

For every type definition, it is also possible to instantiate a constant of that type, which will be available for further use in the model.

Additionally, there are two special metatypes available: the `AliasType` that allows to declare a new type with the same features as a reference type, but with a new name; and the `ExternalType` that allows to register data types which have not been defined through the OSRA model itself. For the latter, the OSRA model cannot determine any properties but relies on the external model to check validity. It only knows the registered types' names which is enough to use them like native types throughout the model (for example for interface definitions).

The OSRA editor provides a graphical interface to define concrete types and constants from the described metatypes. Fig. 1 shows an example of such a data type diagram, which looks similar to data type diagrams in UML [5].

For every type or constant defined in such a way, a corresponding element is added to the underlying eCore model including all the range properties and base type references.

III. Introducing OSRA to ASN.1 and ACN

Although graphical modeling of data types is intuitive, it can become difficult to understand and maintain when complex structured types and type hierarchies have to be modeled. Furthermore, the data types are, by default, only available in their internal eCore representation. This allows easy access to the types' model entities within Eclipse plugins using the Eclipse Modeling Framework (EMF). However, exchanging the types with external tools, such as TASTE, is not trivial.

As our goal is to allow the exchange of data between software developed with TASTE or OSRA without the need for manual interventions, a common intermediate representation is necessary. The straight forward decisions was to use ASN.1 for that purpose. Within the frame of the TASTE project,

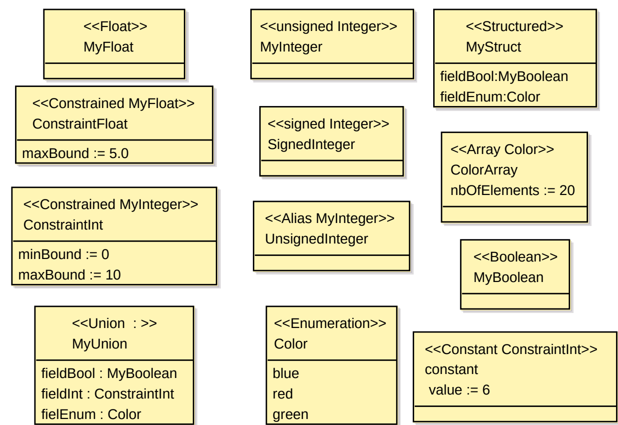

Fig. 1. Example of OSRA data types modeled in the SCM editor with simple numerical types, constrainted numerical types, a structured type and an array type.

ESA already evaluated different programming language independent data type notations and decided to use ASN.1 [1]. TASTE only selected a subset of ASN.1 which provides a reasonable feature set for space applications where type sizes and value sets must be known at compile-time. This reduced feature set also means less complexity in the necessary ASN.1 grammar. By adopting this approach for OSRA as well, only OSRA needs to be complemented with an additional ASN.1 infrastructure. No changes are necessary for TASTE.

Within the ESA funded PaTaS activity [6], an intial ASN.1 grammar implementation in Xtext was already started. It became a starting point to add an ASN.1 editor to OSRA. Like OSRA, Xtext is also based on the Eclipse Modeling Framework. Therefore, the ASN.1 grammar can be tightly integrated with OSRA. The bi-directional implementation works as follows: For all data types, which were created with the graphical model editor of OSRA, a corresponding type in ASN.1 textual notation is generated using Xtend code generators [7]. For all data types which have been manually written and for which an ASN.1 representation exists, a corresponding `ExternalType` is registered in the OSRA model. In this way, all types are available in the OSRA model and can be used in, e.g., interface modeling. All types have also a representation in ASN.1, which means they can be shared with external tools, such as TASTE. This work was carried out as part of [4] and allowed the exchange of data types between OSRA and TASTE.

The upper part of Fig. 2 depicts these relationships between the type representation in OSRA and ASN.1.

For the exchange of encoded data, not only data type descriptions but also a common encoding scheme is needed. ASN.1 provides several standardized encoding rules, e.g., the Binary Encoding Rules (BER) or the Packet Encoding Rules (PER) [8]. However, in our domain of spacecraft onboard software none of those standardized ASN.1 encoding rules are used on a regular basis. Instead, custom protocol formats are prevalent for nearly all communication. For this

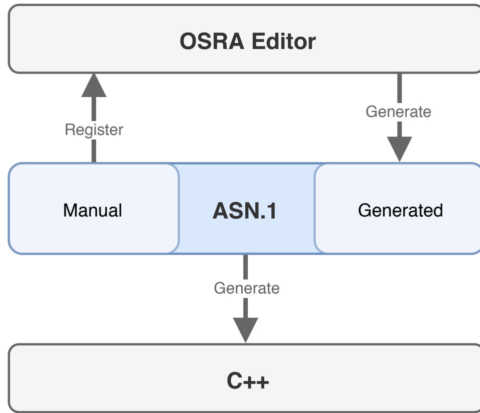

Fig. 2. Flow between the different generation layers

purpose, TASTE already defined its ASN.1 Control Notation, which allows to define a custom encoding format for previously defined ASN.1 types [9], [10]. It offers a great amount of flexibility for encoding rules while still maintaining a reasonably simple grammar. Using ACN as a target for our prototype implementation also provided a better use case for the application of C++ metaprogramming techniques for the encoder compared to the more narrowly defined encoding rules like PER [8].

The usual workflow in Xtext is that a grammar is defined from which a corresponding eCore metamodel is generated. For the new ACN grammar to be able to cross-reference existing data types of ASN.1 it has to share the same eCore metamodel as the ASN.1 grammar. Having two separate, albeit closely related, grammars generating each their own metamodel generates creates a conflict. We stopped this automatic generation process and switched to a manually maintained common metamodel instead. Based on the original ASN.1 metamodel, the ACN entities were gradually added and then the corresponding grammar developed. This approach requires more manual work when changing either a grammar or the metamodel, but it gives greater flexibility [7].

As a result we have now one common eCore metamodel for ASN.1 and ACN, but two separate grammar definitions which only populate their respective part of a model instance. However, in the metamodel, the entities of one grammar are aware of the entities of the other grammar. For example, an ACN encoding definition can refer and resolve the corresponding ASN.1 type defined in another file. In this solution each grammar has also their own respective editor plugins which provides syntax highlighting, auto-completion and on-the-fly validation in the editor. Which editor plugin is active is decided based on the file name extension. As ASN.1 files end with the `asn` extension, ACN uses files with `acn` respectively.

IV. A Data Type System using Modern C++

In the last decade, the C++ programming language underwent several revisions which increased the feature set of the core language significantly. Therefore, the term *Modern C++* is commonly applied if features of the most recent C++ standards, i.e. C++11, C++14 or C++17, are used [11]. Especially the templating system became much more powerful in modern C++ compared to previous standards [11].

At the German Aerospace Center (DLR), we used C++ even for mission critical software parts as part of the BIRD [12], TET-1 [13], BIROS [14] and the Eu:CROPIS [15] satellite missions. With increasing compiler support even for embedded targets and real-time operating systems, the question arises which new features can be useful for on-board software and how they can be applied.

In [4], we investigated how these new features could be used to create a type system which allows types to be easily generated from ASN.1 and which fulfills the following criteria:

- **Support for numerical ranges:** Restricting the values of numerical types to certain ranges allows the type system to check and ensure the validity of data.
- **Clear and intuitive API:** Making it easy for developers to understand the API even with limited knowledge of template metaprogramming. It also increases maintainability and the motivation to adopt the new type system.
- **Compile-time checks:** Spotting sources of errors like type incompatibilities and possible range overflows during compile-time prevents introducing those errors into the source code right at the beginning. It also relieves developers from manually checking for such conditions in the source code and in unit tests.
- **Runtime checks:** Runtime checks shall ensure that the value of a data object stays within its bounds even after modifications. The checks should be carried out automatically by the type itself instead of requiring the developer to trigger it manually.
- **Memory management:** Each data type shall have a known size at compile-time in order to allow for static memory allocation. Data objects should always create a deep copy of its values during copy assignments.
- **Compatibility to existing C/C++ code:** Source code using the new data type system will need to interact with existing libraries. Therefore, the data types shall have support for conversions to the basic types of C++.

To make this paper more self-contained, we show next some key implementation details, but do not repeat the results presented in [4]. We will concentrate on numerical types, structured types and union types, as they represent the situations which are most common or have the potential for a high degree of complexity. For a more in-depth description of the type system, we refer again to [4].

Using mostly variadic templates, static assertions, type traits, and several recursive template metaprogramming patterns [11], it was possible to create types which are aware of constraints. The adherence to the constraints is checked during operations like assignment or type conversion; in many situations already at compile time.

This way the types make sure that they hold valid data at all times. The complexity of these compile time checks are hidden in a general base class, keeping the user interface reasonably

simple. New types are declared usually with only few lines of code. For example, Listing 1 presents the definition of constrained numerical types. First, two integer types declared. As can be seen, the range constraints can be directly part of the type declaration. The first one has two disjoint ranges, the second one only a single range. Both inherit from the templated base class `Integer`. It provides all the range checking mechanisms customized for each type automatically. The two `using` statements in the types body bring in the constructors of the base class, allowing type construction from standard C++ integer constants or variables without the need to define any constructors explicitly. Type conversion from `Int2` to `Int1` are allowed. The reverse is forbidden and produces an error at compile time.

Next, a floating point type is declared. It is not yet possible to specify the ranges directly in the declaration since current C++ standards do not allow floating point numbers as non-type template parameters. Therefore, a corresponding range structure needs to be declared before the real type can be created.

```
1  // Simple definition of constraint integer types
2  class Int1: public Integer<int, Range<-10, -5>,
       Range<0,10> > {
3    using BaseType = Integer;
4    using BaseType::BaseType;
5  };
6  class Int2: public Integer<int, Range<1,5> > {
7    using BaseType = Integer;
8    using BaseType::BaseType;
9  };
10
11 // Not possible:
12 // class F1: public Real<float, Range<0.5, 1.0>
13
14 // Define ranges
15 struct Range1 {
16   static constexpr float min = 0.5f;
17   static constexpr float max = 1000.5f;
18   static constexpr bool minOpen = false;
19   static constexpr bool maxOpen = false;
20 };
21
22 // Define ranged float
23 class TFloat: public Real<float,Range1> {
24   using BaseType = Float<Range1>;
25   using BaseType::BaseType;
26 };
```

Listing 1. Example declaration of numerical types

As variadic templates are used, there is no fixed limit to how many ranges can be assigned to a single numerical type. Of course, each range will increase the cost for validity checking at compile time and runtime. However, from experience it is likely that for most types one or two ranges will usually suffice in most situations. It is also quite obvious, that the implementation of the code-generator generating the C++ code of Listing 1 is straight-forward and simple.

Listing 2 presents an example declaration of an ASN.1 `SEQUENCE` type, which semantically corresponds to a plain `struct` in C++. Here we do not only map a field name to a type like it is done in normal C++ structs, but the type is based on a tuple implementation which means it can also refer to its fields via their position. The accessor methods mimic the behavior of a normal C++ struct and map a name to each of the tuples field. For example, the name `fieldInt` is assigned to the field zero of the tuple, which will return a value of type `TInt`.

```
1  class TSeq
2  : public Sequence<TInt, TFloat>
3  {
4  public:
5    // Inherit constructors
6    using Base = Sequence<TInt, TFloat>;
7    using Base::Sequence;
8
9    // Define for validation constructor
10   TSeq(const Sequence& other) :
11   Sequence(other) {}
12
13   // Field to field name mapping
14   TInt& fieldInt()
15   {  return Base::get<0>(*this); }
16
17   TFloat& fieldFloat()
18   {  return Base::get<1>(*this); }
19 };
```

Listing 2. Example declaration of a sequences type

Finally, Listing 3 shows the basic declaration for `CHOICE` types. A `CHOICE` in ASN.1 is similar to a union type in C++. In our implementation, an object of such type will keep track of the currently active field and report an error if the program tries to read from an inactive field. Although the declaration is nearly the same to the one of the `SEQUENCE` type in Listing 2, the behavior is very different. The return types of the acessor methods will decay to the type of their respective field only if the field is currently active. Otherwise they will trigger the runtime error. For both, the `SEQUENCE` and the `CHOICE` types the validation functions and other behavioral properties are all inherited from the respective base classes `Sequence` and `Choice`.

```
1  class TChoice
2  : public Choice<TInt, TFloat>
3  {
4  public:
5    using BaseType = Choice<TInt, TFloat>;
6    using BaseType::ReturnT;
7
8    ReturnT<0>& fieldInt()
9    { return this->getChoice<0>(); }
10
11   ReturnT<1>& fieldFloat()
12   { return this->getChoice<1>(); }
13
14 };
```

Listing 3. Declaration of a choice type

All presented declarations have in common that they are easily generated using the information captured in the ASN.1 description. This scenario is depicted in the lower half of Figure 2. It is worth pointing out that the ASN.1-to-C++ code generators can be kept very simple, thereby reducing their maintenance complexity. In the next section we present how the previously mentioned ACN encoding rules can be applied to such types and how C++ template metaprogramming can

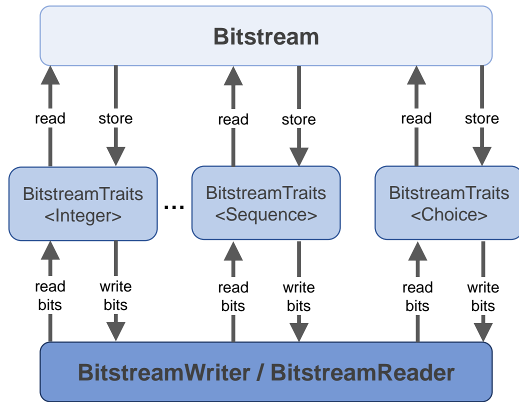

Fig. 3. Overview of the 3 layers for ACN type encoding and decoding.

also help to create customized encoders for each type requiring only little code generated from the ACN description.

V. ACN Encoding Rules using Modern C++

Our implementation for ACN encoding of types declared as shown in the previous section has essentially three layers: First, the `BitstreamWriter` and `BitstreamReader` classes, which are responsible for the low level memory accesses. For example, the `BitstreamWriter`'s only purpose is to write up to 8 given bits regardless of byte boundaries to a given memory address and a given bit offset. The `BitstreamReader` does the same operation, but in reverse. Second, a `Bitstream` class which represents the user interface.

It provides templated versions of a `store` and `read` member functions for a user to write to or read from a given memory space. It also keeps track of the current bit position for reading or writing in a given memory space. However, it does not interact with the `BitstreamWriter` and `BitstreamReader` directly. Instead, it redirects the type-specific implementation for the store and read operations to a templated metaclass `BitstreamTraits<T>` which represents the third layer. This metaclass is then partially specialized for the different C++ types presented in the previous section. A short overview of these layers is given in Figure 3.

In this way a user can store or read an ACN encoded type from a bitstream as shown in Listing 4. It is not necessary to explicitly instantiate the templated member function of the `Bitstream` class. It is automatically deduced from the type passed by the user. The actual encoding is then delegated to the corresponding `BitstreamTraits` class. In other words, all the complexities of the C++ template metaprogramming are effectively hidden from the user and only used in the background.

The ACN syntax has the following structure: Every encoding statement starts first with the type name of a previously defined ASN.1 type. It is followed by a set of properties in square brackets. The supported properties range from simple encoding properties like endianness or bit size of the encoded type to more advanced concepts like

```
1  Int1 myInt(6);
2  TChoice myChoice;
3  myChoice.fieldMyInt() = 3;
4  TSeq mySeq = {5, 3.0};
5  uint8_t buffer[40] = {};
6
7  BitStream stream(buffer);
8  // Encoding
9  stream.store(myInt);
10 stream.store(mySeq);
11 stream.store(myChoice);
12
13 // Decoding
14 stream.reset();
15 stream.read(myInt);
16 stream.read(mySeq);
17 stream.read(myChoice);
```

Listing 4. User perspective for ACN encoding/decoding types

the addition of fixed bit patterns between fields of a sequence or extra fields in a sequence which indicate the active type of a following choice type.

For example in Listing 5, the type `TMode` is encoded as 16 bit integer in little-endian format. The type `MySeq` has two fields. However, in the ACN encoding the extra field `reserved` is added which introduces a fixed bit pattern. Such extra fields are commonly used to model existing encoding protocols which often introduce fixed padding bits or similar at various places. Since those fields do not carry any information for the actual program, they are not part in the ASN.1 type definition. In line 12 the encoding rule for `MySeq` is overriding the previously defined encoding rule for `TMode`. This override is only applied while encoding `MySeq`. It does not affect the encoding of `TMode` values anywhere else. A complete description of all supported properties can be found in the ACN User Manual [10].

```
1  // ASN.1
2  TMode ::= INTEGER(0..10)
3  MySeq ::= SEQUENCE {
4    mode TMode,
5    status INTEGER (-10 .. 20)
6  }
7
8  // ACN
9  TMode [size 16, endianness little]
10 MySeq [] {
11   reserved NULL [pattern '1001'B] // Extra field
12   mode [encoding ASCII],
13   status [size 32, encoding twos-complement,
         endianness little]
14 }
```

Listing 5. Basic example for interaction between ASN.1 and ACN descriptions.

Our goal is to keep the code generator from the ACN model to C++ simple. Therefore, each `BitstreamTraits` is designed in a way that it can encode a complete class of types, e.g., all integer types or all sequence types which are created as shown in the previous section. Then, the code generator only needs to translate the actual ACN properties for each type to C++, so that they can be passed to the `BitstreamTraits` class and direct the encoding. That means, nearly everything regarding the encoding and decoding of the types is captured in the `BitstreamTraits` class and its specializations. Since

only the BitStream class uses those directly, most of the complexities are well hidden from the user as shown in Listing 4. We will concentrate for the remainder of this section on the interaction betwen the BitstreamTraits and the generated ACN encoding properties. In the following examples, we will also concentrate only on the encoding of the data types for brevity. The decoding of a passed binary stream to type values works conceptually in the same way.

For the encoding in C++ to work correctly, the ACN properties have to be available to the BitstreamTraits class and have to have a mapping to the corresponding type. Since this information does not change during runtime, it can be stored as constant expressions and type members in a properties class.

Listing 6 shows this concept for an integer type. Using the template specialization of the class AcnEncoding the mapping between type TInt and the encoding properties is created in line 2.

```
1  template<>
2  struct AcnEncoding<TInt> : public  Properties<
3           Size<16>,
4           Endian<Endianness::little>,
5           Encode<Encoding::ascii>>
6  {};
```

Listing 6. Example ACN properties for an integer type

The Properties class provides default values for the type encoding, which can be overridden. For TInt this is done for the size, endianness and encoding properties in lines 3 to 5.

Listing 7 shows the relevant parts for the encoding of an integer type. First, the general template class for the BistreamTraits is declared, which only contains the general layout of the member functions for storing and reading. Then follows the partial template specialization for integer types starting in line 9. Since the ACN properties are constant and made available at compile time, the implementation can be dependent on those properties. In Listing 7, this is shown exemplary with the two separate store methods in line 14 and line 20 respectively. The former is selected by the C++ compiler only if the ACN property encoding is either pos-int or twos-complement. The latter is selected only if it is set to ASCII. For an integer with the properties of Listing 6, the latter would be the case and the store method for ASCII encoding would be selected by the compiler while the other one would be discarded and not compiled.

This also works for the encoding of more complex types as we will show in the following example. Listing 8 shows an ASN.1 and ACN description. It defines multiple integer types, a choice type with a corresponding enumeration, and a sequence type. In ACN, the type MySeq is assigned an extra bit pattern field reserved and an extra field modeType of type TSelect.

The latter is used in line 20 as the target of ACN's determinant property, which is a special property for CHOICE types. This property requires that an enumeration with the same field names as the CHOICE type exists. In Listing 8 this is the case with the enumeration TSelect in line 4.

```
1  //General template
2  template <typename T,
3  typename P = AcnEncoding<T>,
4  typename = void
5  >
6  struct BitstreamTraits
7  { ... };
8
9  // Specialization for integer types
10 template <typename Prop, typename... Ts>
11 struct BitstreamTraits<Integer<Ts...>,
       Properties >
12 {
13 // Store method for standard encodings (PosInt
       and TwosComplement)
14 template <typename P = Prop, std::enable_if_t<
       isStandardEncoding<P>, int> = 0>
15 static void
16 store(uint8_t*& buffer, Int data, size_t& offset
       )
17 {...}
18
19 // Store method for Ascii encoding
20 template <typename P = Prop, std::enable_if_t<
       isAsciiEncoding<P>, int> = 0 >
21 static void
22 store(uint8_t*& buffer, Int data, size_t& offset
       )
23 {...}
24 ...
25 };
```

Listing 7. BitstreamTraits implementation depending on properties.

The determinant property now encodes a value for the field modeType depending on the active field of select, i.e., 0 if mode is active and 1, if status is active.

```
1  // ASN.1
2  TMode ::= INTEGER(0..10)
3  TStatus ::= INTEGER(0..5)
4  TSelect ::= ENUMERATED{ mode(5), status(10) }
5
6  MySeq ::= SEQUENCE {
7      mode TMode,
8      select CHOICE {
9          mode TMode,
10         status TStatus }
11 }
12
13 // ACN
14 TMode [size 8]
15 TStatus [size 16]
16 MySeq [] {
17     modeType TSelect [],
18     mode [size 32],
19     reserved NULL [pattern '000'B],
20     select [determinant modeType]
21 }
```

Listing 8. ASN.1/ACN description for a more complex example.

Listing 9 shows the corresponding definition of the ACN properties in C++. For brevity, only the properties for the sequence type are shown. The other properties are very similar to the one shown in Listing 6. As explained in Section IV, in our C++ implementation of the SEQUENCE and CHOICE types, not only the field names are available but also the position of each field. This is used in the generation of the ACN properties.

For Listing 9, two extra fields are added. In line 12, the Determinant field is added at position 0, before field mode,

and acts as the determinant of field 1, i.e., field `select`. Additionally, a constant field of 3 bits is added in line 13 at position 1, i.e., before the `select` field. Similarly, the properties which override the usual encoding of field `mode` have a positional argument identifying the correct field as shown in line 16.

In ACN, by default enumerations are not encoded by their value but by the index of the field. Since there is no built-in mechanism in C++ for the determination of the index of a value in enumerations an additional type `MapTSelect` is declared in line 5 which captures this information, i.e., `mode` has the index 0 and `status` has the index 1. This class is passed to the determinant field in line 12 for encoding.

With all the encoding properties properly captured, the `BitstreamTraits` class for sequence types can now simply iterate over all its fields using their position. Before encoding each field, it checks if extra fields need to be encoded beforehand and if overrides exists for the field. That means, no knowledge of the field names is necessary to encode a sequence type and the encoding itself is kept generic. A similar mechanism is used to encode the active field of the choice type using its position.

```
1  enum class TSelect {
2      mode = 5,
3      status = 10
4  };
5  using MapTSelect = EnumMap<TSelect, TSelect::
       mode, TSelect::status>;
6  ...
7
8  template<>
9  struct AcnEncoding<MySeq> : public  Properties<>
10 {
11   using extras = Extras<
12     Determinant<0, 1, MapTSelect>,
13     Constant<1, BitField<3>, 0b000>
14   >;
15   using overrides = Overrides<
16     Override<0, Properties<Size<32>>
17   >;
18
19 };
```

Listing 9. C++ representation of the ACN properties of the example.

VI. Results and Discussion

To validate the correctness of our approach, we used our prototype and encoded a set of ASN.1 types with different encoding rules and compared the binary output with one obtained from TASTE. For the currently implemented feature set, the encodings of our prototype are compatible with TASTE.

In this paper we presented an extension to the data type framework for OSRA presented in [4]. First, we extended the existing ASN.1 metamodel with ACN encoding properties and added a simple ACN grammar in Xtext. From this grammar, a code generator generates the necessary C++ code to configure the encoding of the given type according to the ACN specifications. The goal was reached to keep the code generator from the ACN model to C++ small and maintainable. Only the actual properties need to be generated in C++.

On the C++ side, the user interface could be kept simple and easy to use. All the complexity of the encoding is hidden from the user in the relevant `BitstreamTraits` specialization for each type class. The currently supported feature set of our prototype does not support all ACN properties yet. The main missing features are the *present-when* property and the deep field access. However, from the implementation perspective they share many similarities with the *determinant* property and we do not expect any fundamental problems in their implementation.

C++ metaprogramming techniques are used extensively in our implementation. While in our previous work they were used to check type and value conformance at compile time, in this work they allow the configuration of the encoding for a type from the ACN properties at compile time. The main techniques used are partial template specialization and compile time selection of code paths based on the ACN properties. In other words, the C++ compiler knows at compile time exactly which ACN properties apply for each defined type. It can then discard all methods or conditional branches which depend on any other property. Although optimization was not our focus to during development, the compiler can optimize the code significantly even with only the standard optimization level of O2 enabled. For example, essentially all conditions based on ACN properties can be evaluated at compile time and only the code path needed for a specific configuration is compiled. Furthermore, for most numerical types and not too complex structured types, the encoding is completely inlined with only few instructions in assembly. That means in these cases, all of the classes shown in Figure 3 are optimized out by the C++ compiler and do not appear in the final binary.

As with the data type framework, the resulting C++ code, after all templates are resolved, does not use programming techniques which are usually forbidden in space applications, e.g., dynamic memory allocation. Also neither the data type framework nor the ACN encoding depend on any external libraries. Several type traits and metaprogamming utility classes from the C++ Standard Template Library (STL) are used for convenience, but all of them could also be replaced with custom implementations quite easily. Therefore, the compilation of the C++ code only depends on a recent C++ compiler to be available for the target platform.

VII. Conclusions and Future Work

In conclusion, OSRA was enabled not only to specify its data types in ASN.1 but also to use the ACN encoding rules to encode its data types for custom protocols. A code generator transforms the collected ACN properties as configuration input for the C++ encoding infrastructure. The encoding in C++ is implemented using Modern C++ metaprogramming techniques in order to keep the code generator simple and provide the user with an easy to use interface. This allows the exchange of ACN encoded data types with TASTE generated applications. With these data type system in place we now have a sound basis to look at the next layer of the OSRA metamodel, namely the interface and component type modeling, in more detail.

As mentioned previously, the ACN encoding infrastructure is currently in a prototypical state. In order to use it not just in examples but also in more sophisticated applications, the missing ACN features need to be implemented. Also the code base needs to be refactored to properly observe good coding practices like const correctness, consistent naming conventions and a test suite. Furthermore, besides ACN encoding, TASTE also supports standard ASN.1 encodings like BER, PER and XER. These could be complemented to our encoding infrastructure as well.

The new C++20 standard is scheduled to be officially released until the end of 2020 [16] which will be the next major milestone in the evolution of the language. Here the introduction of modules to C++ promises advantages to our implementation. First, with modules, the templates in header files do not need to be compiled for every translation unit which includes the header files. This should reduce the compile times for applications which would use our type system extensively. Also, modules make it easier to decide which part of an API is visible to a user. The number of helper classes for the C++ metaprogramming could be kept internally which should reduce the number of visible classes for a user greatly. Additionally, the introduction of concepts look like a promising idea to set expectations on template type parameters and check them at compile time. However, a more detailed analysis has to wait until the final C++20 standard is published and compiler with support for the new features are available.

Our goal is to be able to support a minimum capability set of OSRA and be able to generate complete compilable application skeletons from an OSRA model. Therefore, our next focus will be on the modeling of the interfaces and component types in OSRA and how these could be implemented with our presented type system as the basis for data exchange.

Acknowledgements

We thank our colleague Zain Haj Hammadeh for his comments that greatly improved the manuscript.

References

[1] Maxime Perrotin, Eric Conquet, Julien Delange, and Thanassis Tsiodras. Taste-an open-source tool-chain for embedded system and software development. In *Proceedings of the Embedded Real Time Software and Systems Conference (ERTS), Toulouse, France*, 2012.

[2] Marco Panunzio. Specification of the metamodel for the OSRA component model. Technical report, ESA, 2017.

[3] M. Muñoz Arancón, G. Montano, M. Wirkus, Kilian Johann Höflinger, D. Silveira, N. Tsiogkas, J. Hugues, H. Bruyninckx, I. Dragomir, and Taher Muhammad Abu. Esrocos: A robotic operating system for space and terrestrial applications. In *Inforum2018*, 2018.

[4] Jan Sommer, Andreas Gerndt, and Daniel Lüdtke. Creating a reliable data type framework for the osra using modern c++. In *70th International Astronautical Congress (IAC)*, Proceedings of the International Astronautical Congress, October 2019.

[5] Object Management Group (OMG). OMG Unified Modeling Language (OMG UML) Version 2.5.1. OMG Document Number formal/2017-12-05 (https://www.omg.org/spec/UML/2.5.1/PDF), 2017.

[6] Kilian Johann Höflinger, Arno Feiden, Jan Sommer, Ayush Mani Nepal, and Daniel Lüdtke. Patas: Quality assurance for model-driven software development. In *IAC 2019*, 2019.

[7] Lorenzo Bettini. *Implementing Domain Specific Languages with Xtext and Xtend*. Packt Publishing, 2016.

[8] Olivier Dubuisson. *ASN.1 Communication Between Heterogeneous Systems*. Morgan Kaufmann, 2000.

[9] George Mamais, Thanassis Tsiodras, David Lesens, and Maxime Perrotin. An ASN. 1 compiler for embedded/space systems. *Embedded Real Time Software and Systems—ERTS*, 2012.

[10] George Mamais, Maxim Perrotin , and Thanassis Tsiodras. ACN User Manual. Technical report, Neuropublic SA, 2015.

[11] David Vandevoorde, Nicolai M. Josuttis, and Douglas Gregor. *C++ Templates - The complete Guide*. Addison Wesley, 2017.

[12] K. Brieß, W. Bärwald, T. Gerlich, H. Jahn, F. Lura, and H. Studemund. The DLR small satellite mission BIRD. *Acta Astronautica*, 46(2):111 – 120, 2000. 2nd IAA International Symposium on Small Satellites for Earth Observation.

[13] S. Föckersperger, K. Lattner, C. Kaiser, S. Eckert, W. Bärwald, S. Ritzmann, P. Mühlbauer, M. Turk, and P. Willemsen. The modular German Microsatellite TET-1 for Technology on-orbit Verification. In *4S Symposium Small Satellites Systems and Services*, volume 660 of *ESA Special Publication*, page 14, August 2008.

[14] Hubert Reile, Eckehard Lorenz, and Thomas Terzibaschian. The Fire-Bird mission - A scientific mission for Earth observation and hot spot detection. In *Small Satellites for Earth Observation, Digest of the 9th International Symposium of the International Academy of Astronautics*, pages 184–196. Wissenschaft und Technik Verlag Berlin, 2013.

[15] Frank Dannemann and Michael Jetzschmann. Technology-driven design of a scalable small satellite platform. In *Small Satellites, Systems and Services (4S) Symposium, Valetta, Malta*, 2016.

[16] C++20: Current Status. https://isocpp.org/std/status. Accessed: 2020-09-12.