70th International Astronautical Congress, Washington D.C., United States, 21-25 October 2019.

IAC–19–D5.1.10

Creating a Reliable Data Type Framework for the OSRA Using Modern C++

Jan Sommer[a]*, Andreas Gerndt[a], Daniel Lüdtke[a]

[a]*German Aerospace Center (DLR), Simulation and Software Technology, Lilienthalplatz 7, 38108 Braunschweig, Germany*, jan.sommer@dlr.de
*Corresponding Author

Abstract

Ever increasing demands on the complexity of onboard software has lead the European Space Agency to define the On-Board Software Reference Architecture (OSRA) creating a common framework for modeling onboard software for space applications. OSRA provides tools for the description of onboard software (OSW) in a component-centric way, but leaves the implementation of the OSW itself or related auto-coding tools to other institutions. As a first step towards a code-generation framework from high level software models, we present source code mappings from the OSRA data type model to a C++ type system. The goal of the framework is to take care of type safety and value consistency issues and to provide an intuitive interface to the application developer for defining and working with data types, while at the same time having the target of auto-coding in mind. We use language features introduced with the modern C++ standards to allow for extensive validity checks at compile-time and additional checks at runtime.

For the integration with OSRA tools, we take an intermediate step transforming the graphically declared types of OSRA into an ASN.1 representation before generating the corresponding C++ source code. The integration is bidirectional, i.e. data types, which have been constructed solely in ASN.1 notation, can also be used inside OSRA models which helps maintaining more complex data structures in a textual format and enables us to use existing complex data sets from previous projects and from The Assert Set of Tools for Engineering (TASTE) project to test the feasibility and the limitations of the type system.

In the end, we present a type system which can be auto-generated and automatically avoids common sources of error like faulty initialization, out-of-bound access and accidental range overflows. Such errors cause compile-time errors if possible and runtime errors otherwise. In order to provide developers with a practical solution, efforts were made to facilitate integration with existing code bases or third party libraries which allows an iterative process of adaption.

We strive to generate complete onboard software projects from the OSRA component model. The data type system defined here provides therefore the basis for that endeavor as it determines the way components will exchange data and how developers will need to interact with them.

Keywords: Code Generation, Model-driven Software Development, C++

1. Introduction

With the next generation of onboard processing systems soon to arrive [1] and the increased use of commercial off-the-shelf (COTS) parts [2], the amount of processing power in spacecraft onboard systems will continue to rise significantly in the foreseeable future. As the processing power increases much faster than available bandwidth, the onboard systems will need to carry out more post-processing of generated data. The demand for longer autonomous periods of operation results in more complex behavior of the spacecraft and finally the more available processing power will also allow the application of sophisticated algorithms for spacecraft (attitude and orbit) control and instrument operation.

Obviously, the higher demands on the capabilities of the onboard processing system will directly carry over to increased demands on the onboard software. As a result, increased complexity of the software in terms of greatly increased number of lines of code and possible interference between software parts running on the onboard system can be expected. At the same time the development cycles for new spacecraft are reduced to often only few years [3, 4] and the software engineering is pushed further behind in the project schedule. This leaves software development with the need to produce more complex onboard software and consequently more software tests in a shorter amount of time while achieving high reliability of the software.

New development paradigms and tools can help to reduce the share of manual coding. Using extensive compile-time checks and auto-generation of boilerplate

code can support developers to avoid coding mistakes which are spotted only late in the project. The final goal is to allow the software developer to focus on implementing the functional code of his respective module without the burden to worry about too many details of cross coupling with other modules out of his expertise.

This paper presents the first steps toward this goal using OSRA drafted by ESA. In Section 2, we start with presenting the case for a safe data type system followed by an introduction to OSRA and the way its data types are defined in Section 3. In Section 4, we briefly discuss how to transform instances of data types from an OSRA model into an ASN.1 textual notation as an intermediate step before generating the final C++ code. Next, the implementation of the safe data types in C++ are presented in Section 5. The results are discussed in Section 6 including a short comparison with the features of the data type system of the Ada language as a point of reference. Finally, we present our conclusions and the way forward in Section 7.

2. The need for a safe data types system

Spacecraft onboard software or safety critical software in general strives to be robust by design, predictible and provide guarantees related to its behavior in nominal operation and failure conditions. Data types and data objects are fundamental building blocks of procedural programming and therefore share a responsibility towards those goals. They are used abundantly and may be shared and exchanged among many different and only loosely coupled modules. Therefore, ensuring consistency at all times in a reasonably complex application without proper support of the compiler or additional tools can prove to be cumbersome. In cases of transient value overruns, which only appear during tests on the embedded target after an extensive run time, it can be time consuming to track down the correct module introducing the error. Even when such a bug is identified and resolved, there is no guarantee that no further bugs of similar kind are still present in the source tree.

Moreover, unit-testing the interaction of many modules completely is extremely difficult, if possible at all, if the overall system becomes reasonably complex and especially if specialized hardware is involved. Integration tests might be able to cover such erroneous behavior but, as they often demand manual work and possibly a special hardware setup, integrating them into continuous integration systems often proves difficult. The work overhead to run such tests regularly and thoroughly can easily render them infeasible given only limited development resources.

One solution to reduce the amount of possible error sources are more rigorous compile-time checks on type compatibility between different modules complemented with additional runtime checks on value validity. Ideally, the checks are carried out automatically in order to remove the burden from developers to maintain such checks manually in a growing code base. Compile-time checks are especially valuable for this purpose. They are readily available on each development workstation, spot errors as early as possible, are easily integrated into a continuous integration environment, force themselves to be dealt with and do not introduce a runtime penalty. Rigorous value range checking can, if applied properly, reduce the amount of written unit tests significantly as many of them deal with simple range checking of input data to functions. If the data types can guarantee their value range, many of such tests are superfluous. Finally, a common way for data type definition with automatic value consistency checks creates a single interface for all developers to follow and increases maintainability.

In the past, the Ada programming language has played an important role for onboard processing systems in space applications. It provides a type system which allows extensive compile-time checks and requires explicit type casts for most conversions. Certified compilers have to fulfill many conditions covered in the Ada language standard [5] and thereby can give guarantees about the validity of data objects in most situations.

However, although Ada was designed with safety critical applications in mind, the C programming language has become more dominant in recent years due to the increased use of COTS systems and their respective development environments as well as an overall significantly larger development community. Compared to Ada, the C/C++ type system is comparably loose. It does not provide an easy way to limit the value range of numerical types and thus does not automatically check for validity during compile-time, during initialization or at runtime. The lack of in-built support for type constraints and corresponding checks for basic types also progresses further into structured and array types.

At the German Aerospace Center (DLR), we already have experience and successful flight heritage with onboard processing systems programmed in a safe subset of C++; even for mission critical software parts as part of the BIRD [6], TET-1 [4], BIROS [7] and the Eu:CROPIS [3] satellite missions. With nowadays widespread availability of compiler suites with support for modern C++ (i.e. C++11 standard and later) even for embedded targets, it is now possible to look for new programming patterns which could prove valuable in the field of spacecraft onboard software.

In the following, we present a use case which leverages the object oriented nature of C++ and newly introduced language features to build a data type system with a similar feature set as the one of Ada. A short overview of the used C++ language features is given in Section 5.1. The data type system should fulfill 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 and express their intent in source code helps to increase the 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 prohibits 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 using 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 burdening 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 conversion to the basic types of C++.

In case the data types are to be used in a project in the future, the success of the adoption depends largely on the acceptance of the interface by developers. If the interface is perceived as too cumbersome, chances are that more effort is spent circumventing the API than using it even though there might be objective benefits. Therefore, the formulation focuses not only on the definition of a robust data type system, but also on the usability of the interface. As the goal is to generate ready-to-use data types from an OSRA model, the design of the data type system will have auto-generation in mind as well. A short overview of the relevant parts of OSRA is given in the next section.

3. The Onboard Software Reference Architecture

This section introduces the Onboard Software Reference Architecture. It provides a short overview of OSRA in general before going into more detail about the OSRA data type system, as that is the most relevant section for the presented work.

3.1 *Overview*

Within the frame of the Space Avionics Open Interface Architecture (SAVOIR) initiative, the European Space Agency formulates and establishes standards for avionics development and operation [8]. The long term goal in this process is to increase the reuseability and competition in the European space industry through streamlined processes for avionic systems [9]. The SAVOIR Advisory group and subgroups, which are responsible for developing standards for specific topics, are filled with representatives from ESA, national space agencies of ESA member states (e.g. DLR, CNES) and representatives of prime- and subcontractors. In these efforts the SAVOIR-FAIRE subgroup is responsible for the definition of the Onboard Software Reference Architecture (OSRA).

OSRA has been developed over the years within the COrDeT studies [10]. The first final release of the corresponding metamodel has been released in late 2017, followed by the publication of an associated model editor in 2019. Its main goal is to describe the onboard software for a spacecraft project during its complete life cycle using a model-driven approach. It uses the separation of concerns principle to separate the high level software architectural design, the functional parts, the non-functional parts (i.e. scheduling policy, synchronization and communication between components), hardware abstraction and finally software deployment on the modeled hardware from one another.

It is published as an eCore metamodel [11] together with a graphical eclipse-based model editor to generate compliant software models and to enforce a certain development workflow [12]. However, it does not not provide any means to generate source code from a model. The implementation of the code generation from the model and the selection of a corresponding execution platform is left to the user of the metamodel.

OSRA promotes a component-based software development approach, thus the core entities of the model are software components. Each component represents an independent functional software unit. Multiple OSRA components communicate exclusively through defined common interfaces where values of previously modeled data types are exchanged.

During the modeling phase, the first step in the provided editor's workflow in a new OSRA modeling project is the definition of data types for the project. The defined data types are later used to define the data objects for exchange between the components through their dedicated interfaces and for the definition of configuration constants. A robust data type system which fulfills the requirements of OSRA and can be generated from an OSRA model is therefore also the first step towards a fully compliant OSRA development framework. In a previous work, we investigated OSRA's requirements towards an implementing scheduling sys-

tem, especially with regard to current DLR software technologies [13]. This paper concentrates on the data type system of OSRA. Therefore, other parts, i.e. components, hardware modeling, deployment, etc. mentioned above are not further investigated.

3.2 *The OSRA data type system*

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.

That means, besides the common numerical meta types like `IntegerType` and `FloatType` it also provides constrained versions of these meta types which restrict the range of allowed values although the actual bit representation would allow the representation of a larger value set. Further elementary meta types include representations for booleans, enumerations and fixed point values.

`ArrayTypes` are always of fixed length, but it is allowed to specify an index range which does not necessarily start at 0, e.g. an `ArrayType` could be specified to have an index range $[-2, 2]$.

The `UnconstrainedArrayType` represents a meta type where the array size is not known at compile-time, but rather set during instantiation of an object at runtime. Although the name might suggest otherwise, the `UnconstrainedArrayType` still has a maximum number of elements which will be allocated for each object. The size constraint passed to the object constructor only reduces the number of available elements of the allocated storage [11]. Otherwise, the requirement for a fixed size for the type at compile-time could not be fulfilled. In case the number of used elements are much smaller, using this type could lead to a significant overallocation of memory. For indexing, the same rules apply as for the `ArrayType`.

For the construction of complex data types, the `StructuredType` and the `UnionType` meta classes can be used. Their described behavior is consistent with the behavior known from statically typed languages like C/C++ or Ada. Additionally, OSRA defines two string meta types, namely `BoundedLengthString` and `FixedLenghtString`. However, their feature set is essentially the one of a corresponding array type without the freedom in choosing the index range.

As soon as a type definition is added to an OSRA model, it is also possible to instantiate a constant of that type which will then be available for the construction of further types, e.g. to construct the bounds of a ranged integer or in other parts of the model.

Finally, there are two special meta types 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 can not determine any properties, but relies on the external model to check validity. It only knows about the registered types' name which is enough to use it like native types throughout the model (for example for interface definition).

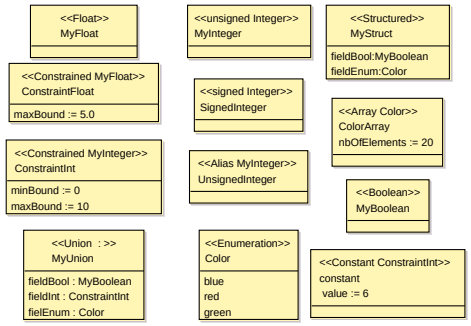

Fig. 1: Example of OSRA data types modeled in the SCM editor with simple numerical types, constrainted numerical types, a structured type and an array type.

Looking at the aforementioned OSRA type system, it essentially provides a basic type system with a feature set similar to that of the Ada language. The OSRA editor provides a graphical interface to define concrete types and constants from the described meta types. Fig. 1 shows an example of such a data type diagram. 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.

4. ASN.1 as intermediate representation

As presented in the previous section OSRA comes with a graphical editor for the definition of data types (see Fig. 1), which shares many similarities with the modeling of *datatypes* in UML [14]. While such a representation is intuitive for modeling use cases where structured types only contain a limited number of fields, keeping it clear for large complex structured type, like state vectors in avionics systems, can be challenging. In UML often *class* classifiers are used even for data type modeling with the convention to only use attributes and without operations. With *class*es it is possible to use inheritance relations in the diagram to assemble complex types from multiple simpler types and thereby improve readability. A similar scheme is not foreseen with the type system in OSRA.

Therefore, a textual notation for the data type definitions would be preferred in such situations. Another case for a textual notation can be made from the maintainability point of view. A human readable plain text format is very well suited for tracking with common version control systems and can be edited with simple text editors.

We decided to have an intermediate textual represen-

tation for the data types which is integrated bidirectionally with the OSRA metamodel. Types defined through the graphical editor need to be converted into this intermediate format, but also the types written directly with the textual notation by a user have to appear inside the graphical editor, so they can be used, for example, for interface definitions. Such a situation is exactly the foreseen use case for the `ExternalType` introduced in the previous section. From this intermediate representation, a single code generator then produces the corresponding types implemented in C++ as described in Section 5.

Within the frame of the TASTE project [15], ESA already evaluated different programming language independent data type notation schemes and decided to use ASN.1 [15]. The benefits of ASN.1 are that it is widespread in the telecommunication industry and it is an officially standardized notation format. 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. The ASN.1 C-Code generator of the TASTE project proves the feasibility of that approach [16].

With selecting ASN.1 for our textual data type notation, we can benefit from these experiences and also have the option to work towards compatibility between our OSRA generated applications and TASTE generated applications in terms of data exchange. In order to integrate ASN.1 into the OSRA workflow, we used the Xtext framework to model the grammar of ASN.1 which is then used to generate an eCore metamodel of the ASN.1 grammar. Code generators can then be easily attached to the grammar using the Xtend framework [17]. As noted earlier, the OSRA metamodel is also implemented using eCore from the Eclipse Modeling Framework. Thanks to that circumstance, it is also possible to use the same Xtext/Xtend code generator facilities to generate the ASN.1 text files from an OSRA model.

The OSRA data type system has less features than the ASN.1 grammar. A direct translation from OSRA to ASN.1 is therefore possible. Generating corresponding ASN.1 representations for data types defined with the graphical editor is therefore comparably straightforward. For the reverse direction, a generator simply creates for every manually entered ASN.1 type an `ExternalType` element inside the OSRA model.

Fig. 2 depicts these relationships between the different type representation and the generators.

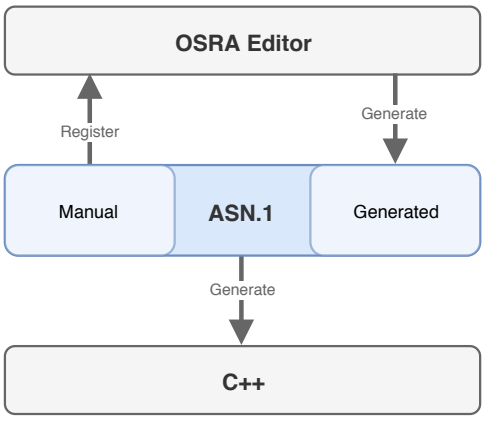

Fig. 2: Flow between the different generation layers

5. Mapping of OSRA data types to C++

There exist already several tools and commercial applications for generating C++ code from ASN.1 descriptions. What they have in common is that they focus foremost on the encoding of the data types to byte streams, which is the primary purpose of ASN.1. The role of their generated data types is only to assemble the values to pass to the encoder. They are not meant to replace the standard type system of C++ and provide only limited functionality for value validation. Most tools also allow for data types of unknown size at compile-time, for example, array types without a specified maximum of elements, which would violate common restrictions in space applications as they would require dynamic memory allocation.

ESA developed an ASN1-to-C compiler within the TASTE project for space applications. It supports a subset of the ASN.1 feature set, so that it does not use dynamic memory allocation and the size of all generated types is known at compile-time. However, the C-API requires developers to manually check the validity of variables through a function call at runtime, which is also used by the TASTE framework to check values before exchanging data between different functional modules. Compile-time checks are only possible in a very limited manner constraint by the feature set of the C language.

In our approach we focus first on a C++ type system which automatically ensures value correctness at runtime without needing developers initiating a validation. The logic for encoding of types to a byte stream will be added in another step. The remainder of this section presents the mapping of the different type classes to concrete implementations in C++ and gives brief examples of the types instantiation, initialization, compile-time and runtime behavior. OSRA's terminology for type definitions differs compared to the one in ASN.1. For example, a `StructuredType` in OSRA is represented by a `SEQUENCE` in ASN.1 [18]. We decided to use the ASN.1 terminology for the implementation of the different types in C++ as it is the terminology of the intermediate representation. A user who does not work with the graphical editor for type definitions does not need to know about those relationships and can directly relate the terms that she is used to in her textual notation. A user who is only familiar with the graphical editor for type definition can still identify the types in the generated C++ code as it has the same type name.

5.1 *Relevant features of modern C++*

"Modern C++" refers to the new language features introduced with the C++11 and later standards. While still being backwards compatible to previous C++ standards, they allow the use of new programming patterns. A comprehensive discussion of the new C++11 features would be out of scope of this paper. This section will focus on the most relevant features used for the design of the data type system.

Already in classic C++, it was possible to use templates in order to define classes and their capabilities for use with different data types and numerical values. The advantage of templates is that template expressions have to be evaluated at compile-time, thus if a template is used wrongly the compiler will abort compilation which prevents broken code to be introduced into the final binary. However, with the introduction of C++11 many new features were introduced to the templating system which massively increased the flexibility and capabilities of template meta programming [19].

For example, static assertions provide a powerful tool to check for custom conditions at compile-time. If the condition is not met, a compile-time error is thrown. The very extensive set of type traits, which is part of the standard library, allows to check properties of given template parameter types at compile-time, alter them and use the results to define types or as conditions for static assertions. The introduction of the `constexpr` keyword is not only a replacement for constant variables, but also allows to define expressions which act like functions but can be evaluated at compile-time if called with compile-time constants and at runtime otherwise. The C++14 and C++17 standard further increased the capabilities of `constexpr` now allowing `if` expressions making complex compile-time evaluations possible. A possibly small but very effective new feature, which is used extensively, is the extension of the `using` keyword for the use of type declarations.

One of the most relevant new features, however, are variadic templates. Their introduction opens the possibility to define template classes which have an unspecified number of template parameters. This allows to create lists of types or non-type parameters and use recursive techniques to, e.g., sort or modify the list by certain criteria at compile-time [19]. Together with the `constexpr` and static assertions, this already allows us to create a powerful compile-time framework which can check ranges, type compatibility or detect other conversion errors and throw a compile-time error if a rule is violated.

It also allows to hide much of the functionality of types in templated base classes and let the concrete type simply inherit the functionality. Essentially, the compiler then generates all the specialized code for a concrete type. This has two main benefits: First, the ASN.1 to C++ code generator can be kept comparably simple which increases its maintainability. Second, as the compiler generates most parts of the functional code, it also has the required context information for compile-time optimization. In fact, many of the helper classes which provide for a convenient user interface are not instantiated in the final machine code, but are optimized out. The machine code is therefore still very compact while the programming interface remains expressive.

5.2 *Type mapping to C++*

This subsection presents the mapping of the basic type classes of ASN.1, namely the `INTEGER`, `REAL`, `SEQUENCE OF`, `SEQUENCE` and `CHOICE` type classes [18], to a corresponding implementation in C++. One key property for all presented types is the guarantee of a default constructor which always initializes a variable to a valid state. If a value is passed for initialization it is tested at compile-time, if possible, or latest at runtime. Once a type is properly initialized, subsequent assignments of values are checked at runtime in order to ensure validity of the held value for the whole life time of an object.

One of the main goals of the developed meta types is code-generation friendliness. After all, the final type declarations are to be generated from their respective model representation in ASN.1.

5.2.1 *Integer Types*

The Integer types of OSRA allow for the definition of a single range. The `INTEGER` type class of ASN.1 allows more complex type constraints like multiple ranges and single value constraints and combinations of both. For example, the following type definition allows only values in the range 1 to 9 (as it is an open interval), 20 to 30 and the value 42.

```
Index ::= INTEGER((0<..<10)|(20..30)|42)
```

Mapping OSRA Integer types to ASN.1 is fairly straightforward. They only allow one range constraint per type which can easily be satisfied through the ASN.1 notation as shown above. On the C++ side a simple template for the definition of ranges has been defined.

```
template<typename BT, BT rmin, BT rmax,
 bool rminOpen=false, bool rmaxOpen=false>
struct Range
{
...
```

Through template meta programming techniques several compile-time checks are implemented for the Ranges. This includes basic sanity checks, e.g., that the lower bound is smaller than the upper bound and compatibility checks, e.g., if one Range is a true subset of another and if a set of Ranges is a subset of another set of Ranges. A single value constraint can simply be expressed as a range where the upper and lower bound have the same number.

The general integer type class is then declared for a given base type and zero or more ranges through variadic template parameters

```
template <typename BT, typename... Ranges>
class Integer
{
...
```

Using the compile-time checks for the Ranges it is now possible to check at compile-time if any two integer types are compatible, i.e. before conversion of one integer to another it is checked if the target type can hold all values of the source type. If it is possible, the types are converted implicitly. `constexpr`-methods are used to check variable values during compile-time where possible and runtime otherwise. However, initializing an integer variable with a constant integer literal, which is not explicitly declared as `constexpr`, does not trigger the compile-time checks (see initialization of `k1` in Listing 1). Therefore, in order to enforce compile-time checks of the initial value, a small helper macro `make_value` is used which forces the compiler to conduct compile-time evaluation of the passed initial value. If the passed initial value is not a compile-time constant it will result in a compile-time error. An example of the usage and behavior of the integer types is presented in Listing 1.

```
using Int1 = Integer<int, Range<-10, -5>,
        Range<0,10> >;
using Int2 = Integer<int, Range<1,5> >;

void integerFunc(Int2 v);
void legacyFunc(int v);
...
Int1 i1 = make_value(Int1, 5); // Ok
// compile-time error:
Int1 j1 = make_value(Int1, 11);
Int1 k1 = 20; // Runtime error

Int2 i2 = make_value(Int2, 5);
i1 = i2;   // Ok
i2 = i1;   // compile-time error
i1 = 3*i2; // Runtime error
integerFunc(i1); // compile-time error
integerFunc(i2); // Ok

legacyFunc(i1); // Ok
legacyFunc(i2); // Ok
```

Listing 1: Example use of integer type implementation

Concrete types are created with the `using` statement as shown in the first two lines. As the templated meta class stays the same for all integer types, this also makes the code generator from ASN.1 very straightforward. It simply needs to generate the declarative line. The following lines show some common usage scenarios of the types. Possible range violations during conversions are directly recognized by the compiler and produce an error. This includes the conversion during assignment and conversions during function calls. The types automatically convert into their base types, if necessary, which allows a convenient interaction with legacy code.

In these cases, of course, after the conversion to a C++ standard type, no further checks are possible for this part of the program, but once the result of, for example, a function call is assigned to an integer class, runtime checks are enabled again. This behavior accounts for the fact that any move to a new type system will be an iterative process, thus interoperability with existing code bases or third party libraries is important.

5.2.2 *Real Types*

For `REAL` types, which represent floating point variables, essentially the same approach as for `INTEGER` types is used. The C++ language does allow non-type template arguments only to be integer literals. As a result, the bounds of the Range type can not be defined using floating point constants similar to the bound definition of Integer types in the previous sub-section. A work-around is a non-template declaration of the Range class with bounds directly as constant expressions. An example is given in Listing 2. After that, the usage of `REAL` types behaves in the same manner as for `INTEGER` types with float or double as possible base types.

```
// Not possible:
using F1 = Real<float, Range<0.5f, 1.0f>;

// Define ranges
struct Range1 {
    static constexpr float min = 0.5f;
    static constexpr float max = 1000.5f;
    static constexpr bool minOpen = false;
    static constexpr bool maxOpen = false;
};

struct Range2 {
    static constexpr float min = 5.5f;
    static constexpr float max = 400.0f;
    static const bool minOpen = false;
    static const bool maxOpen = false;
};

// Define ranged float
using RFloat = Real<float,Range1,Range2>;
RFloat f1 = make_value(RFloat, 42.0);
...
```

Listing 2: Definition of real types

Nevertheless, generating the non-templated ranges for real types is an easy task as all necessary information is available in the description of the corresponding ASN.1 type.

5.2.3 *Sequence Of Types*

A `SEQUENCE OF` type is a list of consecutive variables of the same type, in other words an array. For array types several things have to be considered. First, the internal storage of the array data needs to be decided. The straightforward way of storing an array of integer types as a block of consecutive integer type objects has several drawbacks. The memory footprint would be significantly larger, as every object of an integer type holds its

own this-pointer. For example, for integer types with 8-bit integers as underlying representation the actual size of the type might be several times larger to a standard array due to alignment constraints of the class types. Moreover, the resulting array types could not easily convert into standard C++ arrays for use in legacy code.

Therefore, the functionality of the array types is split up in two parts: storage and presentation. The base class `SequenceOf` is the main interaction point for the user. It owns the data, but internally creates an array of the base type of the passed integer or real type (see Listing 3).

```
template<typename T, typename Range>
class SequenceOf;
{
typename T::BaseType mArray[N];
public:
...
```

Listing 3: SequenceOf template with storage definition

The memory footprint is then again the same as for a standard C++ array of equal size plus the this-pointer of the `SequenceOf` object. Conversion to standard C++ arrays for legacy functions is then trivial again.

Restricting mutable access to elements of the array to be always carried out with the appropriate range checks enabled is another use case to be fulfilled by the implemented class. This proves difficult as no object is available to pass by reference to the caller.

The C++ class template `FixedSpan` bridges this gap. It has a similar function as the span class of the Guidelines Support Library for C++. It holds internally a pointer to an element of a corresponding `SequenceOf` object and allows access only through a safe interface. The `FixedSpan` is also available to grant access to only a sub-range of the original `SequenceOf`, for example, in order to pass to a function.

Listing 4 provides a set of exemplary usage scenarios for the usage of the `SequenceOf` class and its interaction with the `FixedSpan` class.

```
using I1 = Integer<int, Range<0,100>>;
// Define 3 element array
using Array1 = SequenceOf<I1, Range
    <-1,1>>;

// Ok:
Array1 a1 = make_value(Array1, {0,1,2});
constexpr int raw[3] = {0,1,2};
Array1 a2 = make_value(Array1, raw); // Ok

// compile-time error:
Array1 a3 = make_value(Array1, {0,-1,2});
// Runtime error:
Array1 a4 = {0,-1,2};

// Set value via FixedSpan:
a1[1] = 42;
a1[0] = -1; // Runtime error

// Pass mutable span to function
foo(a1.toSpan<Range<0,1>>());

// Initialize array via span
using Array2 = SequenceOf<I1, Range<0,1>>;
Array2 b1 = a1.toSpan<Range<0,1>>();
```

Listing 4: Examplary usage of SequenceOf

Although the addition of the `FixedSpan` class looks like an additional level of indirection at first, in most cases the class is optimized out by the compiler which could be confirmed through the examination of the generated assembly code generated with the default optimization level 2. The code to translate a user index to the actual element in the internal data storage is done by the compiler itself based on the templated array subscript operator of the `SequenceOf` base class.

5.2.4 *Sequence types*

The types discussed until now were uniform and multiple instantiations of a template only differed in the passed template parameters. Hence, a concrete type could often be defined in a single line. On the contrary, structured types need to handle collections of different type classes. In ASN.1 the `SEQUENCE` types are collections which map their fields by name to a certain type. The concept is very similar to structs in C. In fact, the `SEQUENCE` types could be modeled comparably easy with structs, but this approach would introduce a couple of drawbacks. For example, all defined struct types would be independent without a common base class. That means, there is no common way to traverse through all fields which would make it necessary to generate the validation infrastructure individually for each struct type. This puts a significant burden on the code generator and would increase with any additional feature which would be introduced, e.g., serialization. Additionally, the resulting type declaration would be comparably large and complex.

Instead, we propose a tuple-based implementation. This yields several advantages. As tuples allow to traverse all of their fields and their respective types at compile-time it is possible to move the validation infrastructure into the tuple base class. A concrete `SEQUENCE` type simply inherits from the base class and the compiler will generate the concrete validation functions on the fly. Serialization extensions could then be added in a similar fashion in the base class and without the need to change the C++ code generator.

However, tuples identify their fields not via a name, like structs, but through its respective position in the tuple. Hence, the C++ code generator needs to generate this mapping between field and field name. This is nonetheless a straightforward task.

In Listing 5 an exemplary implementation generated for a `SEQUENCE` type with the name `TSeq` is shown. There are several key elements to note: The type itself inherits from a tuple-based class `Sequence` and passes the types of all contained fields to it. This already allocates the memory for the fields and sets up the validation infrastructure of the type. In order to avoid reim-

plementing all constructors for tuple initialization, the constructors are also imported from the `Sequence` base class.

Next, one specialized constructor with the base type as parameter needs to be implemented. This allows to initialize variables of `TSeq` with the `make_value` macro in the same manner as for all previous types. Finally, the mapping of the field names is provided for each field in order to grant access to the stored values. The type of each field is known by the generator and can be used for the declaration to increase readability. However, it would have been also possible to use the more generic alternative declaration.

```
using TInt = Integer<int, Range<0, 10>>;
using TFloat = Float<Range1>;
using TArr = SequenceOf<TInt, Range<0,1>>;

class TSeq
  : public Sequence<TInt, TFloat, TArr>
{
public:
// Inherit constructors
using Base = Sequence<TInt, TFloat, TArr>;
using Base::Sequence;

  // Define for validation constructor
  TSeq(const Sequence& other) :
    Sequence(other) {}

  // Field to field name mapping
  TInt& fieldInt()
  {  return Base::get<0>(*this); }

  // Alternative declaration
  Base::element_t<1>& fieldFloat()
  {  return Base::get<1>(*this); }
...
// USAGE:
// compile-time check:
TSeq seq = make_value(TSeq,
            {1, 1.0, {1, 1}});
// Runtime check:
TSeq s1  =  {1, 1.0, {1, 1}};
// Runtime check:
s1.fieldInt() = 9;
```

Listing 5: Implementation of a sequences type

In the lower part of Listing 5 usage examples for variable initialization and assignment are given. As with previously mentioned types, if possible, a compile-time check of the initialization value is done using the `make_value` macro otherwise runtime checks are used. Access to the fields of the sequence is very close to the usage of plain structs. Only an additional pair of braces is necessary as the accessors are member functions. However, the same style is used for the `CHOICE` type implementation which provides for a consistent user interface. `Sequence` types do not implicitly convert into a corresponding legacy type. Although, it might be possible to ensure that the internal data layout of the underlying tuple is equal to that of a standard C-struct, this use case is seen as quite rarely applicable and therefore not seen as worth the effort.

5.2.5 *Choice types*

The `CHOICE` types have a similar structure as the `SEQUENCE` types from the user perspective. Their main difference is that only one of their typed fields is valid at a time. Therefore, one of the main error sources to prevent is accidentally accessing the internal data through the wrong field type.

The `CHOICE` implementation therefore keeps track of its current state and creates a runtime error if a field is accessed which has not been set previously. The type implementation uses the concepts of Discriminated Unions for internal storage of data as presented in [19], but has a positional-based access mechanism in order to allow multiple fields of the same type in a `CHOICE`. Internally, it manages an allocated buffer with the size of the largest field type. The variable value of the active field is created in or copied to this buffer and an internal discriminator set to keep track of the active field. Listing 6 provides a short overview of the main features of the `CHOICE` type implementation in C++.

```
using TInt = Integer<int, Range<0, 10>>;/
using TFloat = Float<Range1>;
using TArr = SequenceOf<TInt, Range<0,1>>;

class TChoice
  : public Choice<TInt, TFloat, TArray>
{
public:
using BaseType = Choice<TInt, TFloat,
    TArray>;
using BaseType::ReturnT;

    ReturnT<0>& fieldInt()
    { return this->getChoice<0>(); }

    ReturnT<1>& fieldFloat()
    { return this->getChoice<1>(); }
...

// USAGE:
// Create choice variable
TChoice c1;

// Set choice variable
c1.fieldInt() = make_value(TInt, 1);

// Read correct field:
TInt i1 = c1.fieldInt();
// Runtime error as fieldInt is active
TFloat f1 = c1.fieldFloat();

// Reassign c1
c1.fieldFloat() = make_value(TFloat, 1.1);
TFloat f1 = c1.fieldFloat();
```

Listing 6: Implementation of a choice type

Similarly to the `SEQUENCE` type implementation, it can be generated automatically with little effort. However, here the return type for the field access methods is not the actual type of the field, but a wrapper for it. The wrapper is in charge of deciding at runtime if the access is valid, i.e. if the current state of the choice variable is the one of the field. If yes, the wrapper is implicitly cast

to the result type. If not, it will create a runtime error.

The second function of the wrapper is to allow reassigning the choice variable to a potentially new field. If a new value is assigned to a previously inactive field, the wrappers assigment operator will destroy the value of the old field, update the variables internal state and copy the new value to its internal buffer.

5.2.6 *Enumerations*

The generation of `ENUMERATION` types is a straightforward task. C++11 introduced `enum classes`. Compared to the classic C and C++ `enums` they avoid naming conflicts between fields of different enumerations through their own dedicated namespace. Also, they already provide strong type safety and will only allow their own fields as values, i.e. reject simple integer values. The code generator simply creates an `enum class` for each `ENUMERATION` type with the corresponding mapping between field name and integer value.

5.2.7 *Constants*

All of the previously introduced type templates already have mechanisms to check valid values at compile-time. Therefore, constants can simply be defined through declaring a `constexpr` variable of a given type and assigning an initial value to it. Listing 7 shows several exemplary constants defined in that way using the types of the previous sections.

```
constexpr TInt intConst   = 5;
constexpr TFloat floatConst = 1.5;
constexpr TArr arrayConst = {5, 6};
constexpr TSeq seqConst
      = {1, 1.5, {5,6}};

TChoice c;
c.fieldInt() = intConst;
```

Listing 7: Examplary definition of constants values for numerical, array and sequence types.

As can be seen, the use of the `make_value` macro is not necessary as the compiler already knows to evaluate the expressions at compile-time due to the `constexpr` keyword. For `CHOICE` types creating constants is currently not possible. It could be realised easily, if the active field would be uniquely identifiable through the type of the assigned value. However, since we allow a `CHOICE` to have multiple distinct fields of the same type other solutions would be to either pass the index of the active field to the constructor, or provide static class functions which create a new object with the desired field set as active. Both solutions would either require the user to remember the index of the desired active field or require more effort for the C++ code generator for a use case with only little benefit. Instead we propose to create a constant of the desired initial value and assign that to the correct field in a second step as shown in the last line of Listing 7.

For a different design concept for `CHOICE` types which solves this issues using C++20 patterns see our comments in Section 7.

6. Discussion

In this paper, we presented a data type system for space applications with compatibility to the feature demands of OSRA in mind. In the following, we discuss the achieved results based on the criteria developed in Section 2. Afterward we shortly compare the features of our type system to the feature set of the Ada language.

6.1 *Results*

The data types do support numerical ranges. In fact, for primitive types like Integer and Real types it even supports the use of multiple disjoint ranges for a single type. For Array types the support is currently limited to a single range. It would be possible to implement support for multiple ranges for Array types as well, but at the moment we do not see any real use case where this would be a relevant need.

Evaluating the user-friendliness of the API is to a degree a matter of personal preference, but there are evident indicators for it being clear and intuitive: The types behave very similar to their standard C counterparts. The normal arithmetic operators can be used with the numerical type classes and the array subscript operator behaves naturally for Array types. `SEQUENCE` and `CHOICE` types use the same interface and behave very similar to standard C++ `structs`. The `make_value` macro is necessary in order to enforce compile-time checks of initial values, but it was made sure that the same macro can be used for all types in the same manner.

Type safety is ensured largely through compile-time checks during initialization of variables and conversions between types. The compile-time checks are realized through extensive use of modern C++ features like variadic templates, const expressions and static assertions. A benefit of this approach is that most of the functionality could be collected in templated base classes. The compile-time checks are complemented with runtime checks for the remaining situations in which the values are not known at compile-time. The runtime checks are carried out automatically without the need for a user to initiate them manually and thereby ensure that the value of a variable is within its defined bounds for its entire life time.

Data types have a known size at compile-time, that includes structured types and even array types since OSRA requires them to have a maximum number of elements. Also, dynamic memory allocation is not used at any time. It provides compatibility to standard C++, i.e. numerical types and array types of numerical types can be used directly with existing libraries as they convert directly to their underlying base types.

Furthermore, we decided to use a subset of ASN.1 as an intermediate textual notation for data type description. It gives the user the freedom to either define data types in the graphical scheme of OSRA and then generate the corresponding ASN.1 notation of the data type or define the data types in ASN.1 directly. The written ASN.1 types are automatically registered as external types in the OSRA model. This way they are available for further use, for example, in the interface definition part of the general OSRA work flow and can be referenced in the same way as any other type. The ASN.1 notation is more feature rich e.g. in expressing constraints on types than the OSRA model, but the use of external types makes it possible to make use of all of these features without the need to change anything on the OSRA metamodel.

The ASN.1 grammar has been written using the Xtext/Xtend framework which automatically provides a text editor with syntax support for the language and the necessary infrastructure for code generation. In theory, the generation of ASN.1 types from an OSRA model could also be used as input for the data model for TASTE projects, but this has not been tested in the scope of this work.

Regarding the performance of the system, the runtime checks mentioned beforehand add some runtime penalty to the overall performance. In space applications, it is common that validity checks have to be performed abundantly at function entries, return values and during processing in order to ensure correct operation at all times and spot effects due to cosmic radiation. With types being automatically checked at runtime in our type system, this often means that the runtime check simply moves from the manual code into the type itself. Additionally, the extensive compile-time checks make runtime checks in many common situations unnecessary as they can give guarantees over a variable's value. The runtime penalty of the type validation is quite small since the compiler can inline most of the common checks and in our opinion the benefits of a consistent and safe type system outweigh the possible performance cost in most applications. For performance critical parts of the software it is still possible to use standard C++ types and convert the results of the performance critical part back to our type system. Then, only in these performance critical parts a developer has to manually deal with erroneous variable values, but other parts of the program are unaffected by it.

In conclusion, the presented data type system fulfills the general requirements on data types of OSRA, is automatically generated and uses many modern C++ features to provide strong type safety and type validity through compile-time and runtime checking. Using ASN.1 we complemented the OSRA editor with the possibility to use a textual notation to define data types. This gives us a good starting point for further exploring ESA's Onboard Software Reference Architecture and its applicability in a modern model-based software development workflow.

6.2 *Comparsion to Ada*

The feature set presented in Section 5 is in large parts similar to that of the basic type system of the Ada language [5]. Type compatibility is checked at compile-time as well as the correct initialization, albeit a macro is needed in order to enforce it. In Ada, however, type conversions need always an explicit cast unless the source type is derived from the target type. In the presented C++ framework this is not the case. For example, if the ranges of two integer types are compatible, an implicit conversion is allowed and carried out by the compiler. Listing 1 provides an example for that situation. This behavior is a deliberate design decision as it follows common behavior of C++. If the goal would be to model the Ada behavior more closely, this could also be achieved easily by marking the respective conversion operators of the types as `explicit`.

At runtime, similar to Ada, assignments are checked and if they are not valid a runtime error is generated and the program terminated. That includes the assignments to basic integer or floating point variables, out-of-bounds checks for array objects and access to fields of `SEQUENCE` or `CHOICE` types.

In Ada the discussed features are part of the language standard. Hence, the error messages given by the compiler or at runtime in case a failure condition is met are usually easy to understand and point directly to the origin of the rule violation. As our C++ type system has been implemented using template metaprogramming techniques error message tend to be comparably long and often obfuscated. Using `static_assert`, it is possible to create custom compile-time error messages in C++, but the capabilities are rather limited and often do not allow to fully explain the context of a rule violation. Nevertheless, the line responsible for the compile-time error is highlighted which helps finding the reason for the failure. Some possible ways to reduce the amount of clutter in order to produce more clear error messages are given in Section 7.

Finally, for Ada development usually Ada runtime libraries are needed for a certain target platform if the type system is to be used with higher level operating system functions like a scheduling system. The benefit of the presented C++ type framework here lies in the fact that it essentially only relies on direct language features. The few places where it uses parts of the standard library, namely the usage of `std::tuple` and some utilty traits can be easily replaced with custom implementations [19]. The type system can therefore be used independently of the operating system or even on bare metal target platforms if a modern C++ compiler is available.

7. Conclusions and future work

The presented data type system can serve as the base for further development regarding the implementation of the OSRA metamodel entities using modern C++ programming techniques and automatic code generation. For the template meta programming features of the C++11 to the newest C++17 standard have been used. Whereas the C++14 and C++17 standards focused mostly on extending existing features introduced in C++11, C++20 will be a new milestone regarding the C++ language evolution and is due to be released in 2020. The current working draft for C++20 includes several new language feature which could prove valuable for further development of the presented framework [20]. Most notably the introduction of concepts will introduce a way to formulate requirements onto template type parameters. This allows early checking of their properties and the production of more readable error messages for the user if several conditions are not met. Furthermore, the `using` keyword will finally be available for scoped enumerations [21]. With this feature the interfaces for `SEQUENCE` and `CHOICE` types could be simplified in a way that it is not necessary anymore to generate the mapping between every field name and its corresponding internal storage. An implementation for a `CHOICE` constant initializer would also be straightforward with this approach. Theoretically, this approach could have been implemented already with scoped enumerations as defined in C++11, but in order to access fields, long namespace expressions would be needed, reducing the readability of the source code significantly. A similar implementation with standard enumerations on the other hand would lack the type safety needed in order to catch erroneous field accesses due to similar field names in different types.

Furthermore, the data type system as it is now has no integrated means to serialize data. The interface was designed with future extensions in mind in order to allow exactly such kind of additions. These features only need to be implemented in the templated base classes in a generic way to be available to all concrete types derived from them. The TASTE project already developed a comparably easy to understand grammar for declaring the serialization rules for the data types defined in ASN.1 [16]. It has to be evaluated if this would be a feasible approach for us as well.

With a solid implementation for the data type system, further parts of the OSRA metamodel can be implemented. The next step would be to develop a sound design for the component interfaces using the data types as presented in this paper. Then, the components themselves have to be designed in C++. This includes the mapping of the non-functional properties of the components to a chosen execution platform where an initial evaluation has already been conducted [13]. This would then yield an implemented minimal capability set which supports the generation of a complete compilable application skeleton from an OSRA model after the data types, the interfaces, component types and component instances have been assembled in the model editor.

Acknowledgements

We thank our colleagues Zain Haj Hammadeh and Olaf Maibaum for their comments that greatly improved the manuscript.

References

[1] Jean-Luc Poupat. DAHLIA - Very High Performance Microprocessor for Space Applications. In *ADCSS 2017*, 2017.

[2] D. Lüdtke, K. Westerdorff, K. Stohlmann, A. Börner, O. Maibaum, T. Peng, B. Weps, G. Fey, and A. Gerndt. OBC-NG: Towards a reconfigurable on-board computing architecture for spacecraft. In *2014 IEEE Aerospace Conference*, March 2014.

[3] 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.

[4] 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.

[5] ISO 8652:2012(E). Ada Reference Manual. Standard, ISO, 2016.

[6] 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.

[7] Hubert Reile, Eckehard Lorenz, and Thomas Terzibaschian. The FireBird 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.

[8] E. SAVOIR website:. `http://savoir.estec.esa.int`.

[9] Peter Mendham and Andreas Jung. SAVOIR - FAIRE working group report (software) - Documentation status. In *ADCSS 2017*. ESA, 2017.

[10] Andreas Jung, Marco Panunzio, and Jean-Loup Terraillon. SAVOIR - FAIRE - On-Board Software Reference Architecture. Technical report, Savoir-faire working group, 2010.

[11] SAVOIR. SAVOIR OSRA - SCM Metamodel Specification, 2018.

[12] ESA and Obeo. Draft: User Manual of the OSRA SCM Model Editor, 2018.

[13] Jan Sommer, Raghuraj Tarikere Phaniraja Setty, Olaf Maibaum, Andrea Gerndt, and Daniel Ludtke. Evaluation and Development of the OSRA Interaction Layer for Inter-Component Communication. In *2019 IEEE Aerospace Conference*. IEEE, March 2019.

[14] 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.

[15] 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.

[16] 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.

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

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

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

[20] ISO/IEC. Working Draft, Standard for Programming Language C++, 2019.

[21] Gašper Ažman and Jonathan Müller. P1099R5 Using Enum. Technical report, CWG, 2019.