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

Jan Sommer, Andreas Gerndt, Daniel Lüdtke

# Overview

- Short introduction into OSRA and TASTE
- Addition of ASN.1 and ACN support to OSRA
- A data type framework using Modern C++
- An ACN encoder using Modern C++
- Results and discussion
- Future outlook

# Motivation

- Create reliable on-board software for spacecraft faster
- Facilitate the development for more complex systems for upcoming missions
- Introduce the use of model-driven software development
- Support software development with auto-generation of boiler-plate code
- Evaluate modern programming patterns for space-related applications

# Motivation for using TASTE and OSRA

## TASTE

- Provides a functional framework for modeling and generation of safety critical software
    - Data types
    - Function modeling
    - Hardware modeling
    - Code generators
    - Build system

- Was part of projects we participated in, e.g.:
    - PaTaS [1]
    - ESROCOS [2]

## OSRA

- Provides an EMF metamodel and editor for the modeling of spacecraft on-board software
- Supplemental documents describe:
    - Semantic of each metamodel entity
    - Defined OSW services
    - Requirements for an implementing execution platform

- Can be used to generate code for existing execution platform:
    - Evaluated application of OSRA for DLR's Tasking Framework [3,4]

# Relevant Features of TASTE and OSRA

**TASTE**

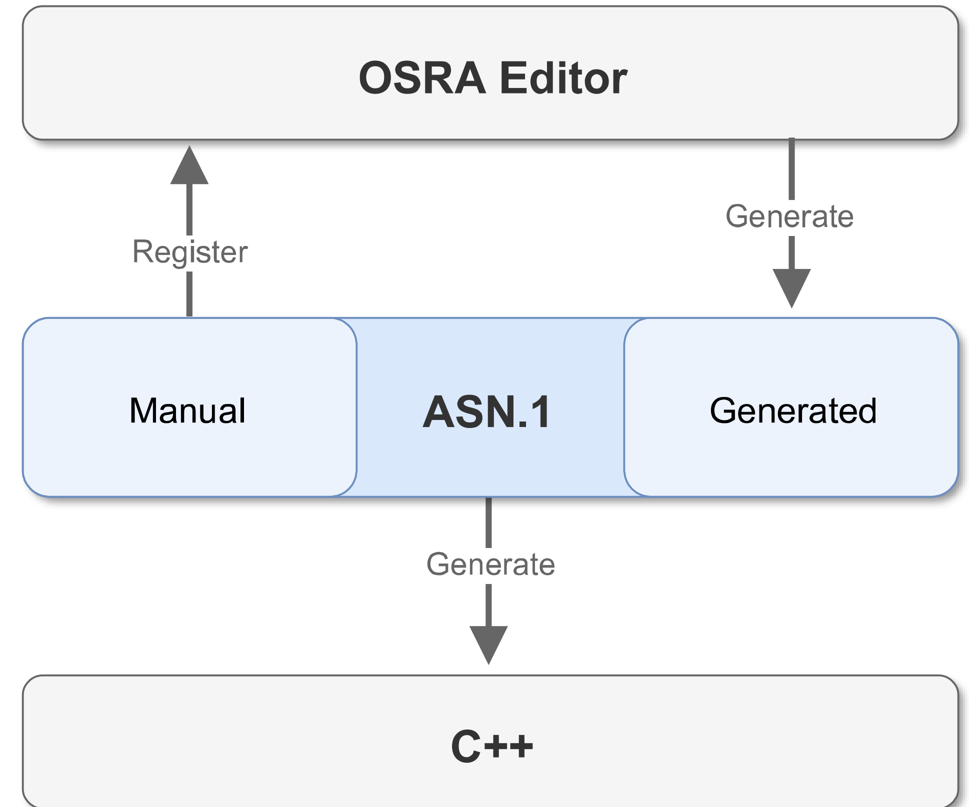
- Wide scope for possible applications
- Custom design tools
  - Uses standardized AADL as backend
- Uses ASN.1 for data type representation
- Allows custom encoding for data types using ACN
- Provides code generators for ADA and C

**OSRA**

- Designed for the modeling of spacecraft OSW
- Based on Eclipse, EMF and related plugins
- Uses an EMF-based graphical data type representation
- Has some basic encoding properties, e.g. bit size
- No code generators provided. To be implemented by the using entity

# Generation Workflow with ASN.1 Notation as Intermediate Step

- Leverage the experience of TASTE and use ASN.1 as a common data type notation
- Complement OSRA with ASN.1 capabilities:
  - Graphical types → generate ASN.1 notation
  - ASN.1 types → register as external type in editor
- Use a common code generator to produce the final C++ source code from the type definitions in ASN.1
- Xtext/Xtend framework used for code generation from OSRA model and for creating the ASN.1 grammar
- All parts interact using the Eclipse Modeling Framework
- Work carried out as part of [5]



# Data Type Implementation using Modern C++

**Requirements for reliable data type system:**

- Support for numerical ranges
- Compile-time checks
- Runtime checks
- Memory management, i.e. no dynamic allocation
- Support for existing C/C++ libraries
- Clear and intuitive API and easy to generate

**Why modern C++?**

- Modern C++ refers to C++11 and later standards
- Templating system has been extended significantly and allows template meta programming
- Feature rich compile-time evaluations possible
- Now modern compiler available for most hardware platforms
- Good tool support and large community
- Only dependent on recent C++ compiler

# Example 1: Numeric Types

## ASN.1

```
Int1 ::= INTEGER(-10..-5 | 0..10)
```

## C++ usage:

```
// Usage:
Int1 i1 = make_value(Int1, 5); // Ok

// compile-time error
Int1 j1 = make_value(Int1, 11);
Int1 k1 = 20; // Runtime error
```

## C++ encoding for Integer types

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

```
class Int1: public Integer<int, Range<-10, -5>, Range<0,10> > {
  using BaseType = Integer;
  using BaseType::BaseType;
};
```

Generated

```
void integerFunc(Int1 v);
void legacyFunc(int v);
```

# Example 2: Sequence Types

## ASN.1

```
TSeq ::= SEQUENCE {
    fieldInt   TInt,
               fieldFloat TFloat,
               fieldArr   TArr
}
```

## C++ usage:

```
// 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;
```

## C++ encoding for Integer types

Generated

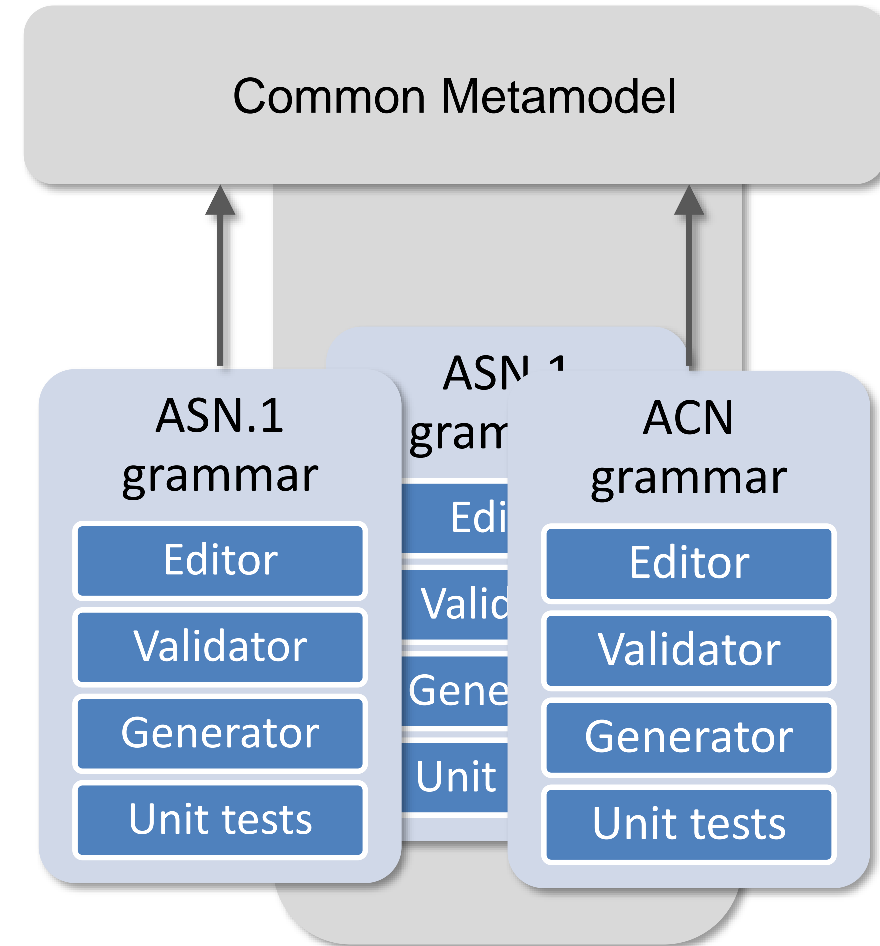
```
class TSeq: public Sequence <TInt, TFloat, TArr>
{
public:
using Base = Sequence <TInt, TFloat , TArr>;
using Base::Sequence;

    TSeq(const Sequence& other) :
    Sequence(other) {}
    // Field to field name mappings
    TInt& fieldInt()
    { return Base::get<0>(*this); }

    TFloat& fieldFloat()
    { return Base::get<1>(*this); }

    TArr& fieldArr()
    { return Base::get<2>(*this); }
};
```

# ACN Support for OSRA

- Standard ASN.1 encoding rules (BER, PER, etc.) have little relevance in space applications

- TASTE used again as orientation:
  - ACN provides support for custom protocols
  - →Add ACN support to OSRA
  - No changes in TASTE necessary

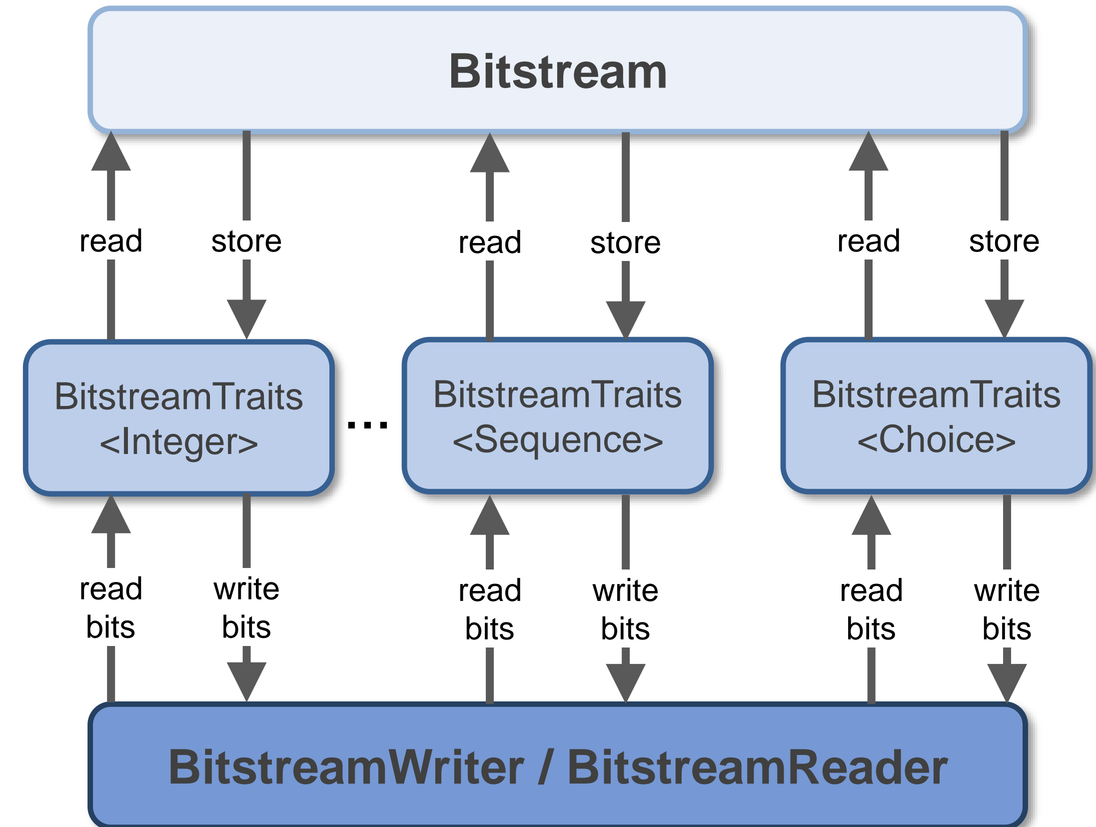
- Maintain a common eCore metamodel for ASN.1 and ACN entitites
  - Have two separate Xtext grammars for ASN.1 and ACN which work with the same metamodel
  - Includes separate editor plugins, validators, etc.
  - ACN grammar can cross-reference ASN.1 types



# ACN Encoding using Modern C++

**Requirements the ACN encoder:**

- Easy to use user interface
- Support for unaligned encoding
- Easy to generate from ACN model
- Memory management, i.e. no dynamic allocation
- Extensible for custom types
- No external dependencies



# User Interface of the ACN Encoder

```
// Create some variable
Int1 myInt(6);
TChoice myChoice;
myChoice.fieldMyInt() = 3;
TSeq mySeq = {5, 3.0};

// Allocate a buffer
uint8_t buffer[40] = {};

// Create a bitstream
BitStream stream(buffer);
```

```
// Encoding
stream.store(myInt);
stream.store(mySeq);
stream.store(myChoice);
```

Templated store methods

```
// Decoding
stream.reset();
stream.read(myInt);
stream.read(mySeq);
stream.read(myChoice);
```

Templated read methods

**Complexity of C++ template metaprogramming is hidden from the user**

# Example 1: Encoding of Simple Integer Type

## ASN.1 / ACN

```
-- ASN.1
TInt ::= INTEGER(0..10)

-- ACN
TInt [encoding ASCII]
```

## C++ ACN properties

```
template<>
struct AcnEncoding<TInt> : public  Properties<
      Encode<Encoding::ascii>>
{};
```

Generated

## C++ encoding for Integer types

```
// Specialization for integer types
template <typename Prop, typename... Ts>
struct BitstreamTraits<Integer<Ts...>, Properties > {
      // Store method for standard encodings
      template <typename P = Prop,
      std::enable_if_t<isStandardEncoding<P>, int> = 0>
      static void
      store(uint8_t*& buffer, Int data, size_t& offset)
      {...}

      // Store method for Ascii encoding
      template <typename P = Prop,
      std::enable_if_t<isAsciiEncoding<P>, int> = 0 >
      static void
      store(uint8_t*& buffer, Int data, size_t& offset)
      {...}
};
```

# Example 2: Encoding of Complex Structured Type

## ASN.1 / ACN

```
-- ASN.1
TMode ::= INTEGER(0..10)
TStatus ::= INTEGER(0..5)
TSelect ::= ENUMERATED{ mode(5), status(10) }
MySeq ::= SEQUENCE {
   mode TMode,
   select CHOICE {
      mode TMode,
      status TStatus }
}
-- ACN
TMode [size 8]
TStatus [size 16]
MySeq [] {
   modeType TSelect [],
   mode [size 32],
   reserved NULL [pattern '000'B],
   select [determinant modeType]
}
```

## C++ ACN properties

```
enum class TSelect {
   mode = 5,
   status = 10
};
using MapTSelect = EnumMap<TSelect, TSelect::mode,
TSelect::status>;
```

Generated

```
template<>
struct AcnEncoding<MySeq> : public  Properties<>
{
 using extras = Extras<
  Determinant<0, 1, MapTSelect>,
  Constant<1, BitField<3>, 0b000>
 >;
 using overrides = Overrides<
  Override<0, Properties<Size<32>>>
 >;
};
```

# Results

- Validation of types encoded with our prototype
  - Tested with different type and combination of supported properties
  - Binary output is compatible with one obtained from TASTE
- Code generators from ACN to C++ are small and maintainable
  - Only the ACN properties need to be translated into C++
- User interface is simple and easy to use
- C++ metaprogramming allows compiler to optimize out most classes
- C++ code only dependent on recent C++ compiler (only some traits and helpers from STL used)

# Future Work & Outlook

- Implement missing ACN properties (present-when and deep field access)
- Add support for standard ASN.1 encodings (PER, BER)
- Refactor code to remove prototype status
- Goal is to generate compilable application skeleton from OSRA models with minimum capability set
  - Next step: Investigation of interface and component type modeling in OSRA
- C++20 is scheduled to be released until end of this year
  - Will be the next major revision of the language
  - Modules and concepts could prove valuable features for space applications
  - Further investigation when standard is published and compiler support is available

# References


- [1] Höflinger, Kilian Johann et.al. „*PaTaS: Quality Assurance for Model-driven Software Development*“. In: Proceedings of the International Astronautical Congress, IAC. 70th International Astronautical Congress
- [2] Muñoz Arancón, et. al. „*ESROCOS: A ROBOTIC OPERATING SYSTEM FOR SPACE AND TERRESTRIAL APPLICATIONS*“. Inforum2018, 03.-04. Sep. 2018, Coimbra, Portugal.
- [3] Sommer, Jan, Raghuraj Tarikere, Phaniraja Setty, et.al. „*Evaluation and Development of the OSRA Interaction Layer for Inter-Component Communication*“. In: IEEE Aerospace Conference Proceedings 2019. DOI: 10.1109/AERO.2019.8741823
- [4] Tasking Framework (open source version): https://github.com/DLR-SC/tasking-framework
- [5] Sommer, Jan, Gerndt, Andreas and Lüdtke, Daniel. „*Creating a Reliable Data Type Framework for the OSRA Using Modern C++*“. In: Proceedings of the International Astronautical Congress, IAC. 70th International Astronautical Congress

DLR

# Summary

- Short introduction into the data type in TASTE and OSRA
- Complemented OSRA editor with ASN.1/ACN support
- Short overview of our data type system using Modern C++
- Introduction of our ACN encoding prototype
- Outlook on future work towards a full generation of OSRA-based onboard software