The NANDRAD CodeGenerator

Andreas Nicolai
andreas.nicolai@tu-dresden.de

Version 1.0.0, June 2020

Table of Contents

1. Overview

What is the NANDRAD-Code-Generator? Basically, it is a tool to parse NANDRAD header files, pick up lines with special comments and then generate code for:

- translating enumeration values into strings, descriptions, colors and units (very handy for physical parameter lists)
- reading and writing xml data structures (elements and attributes) for designated member variables
- comparing data structures with default instances (so that only modified members are written to file), and other utility functions

1.1. How does it work?

The code generator is called as part of the CMake build chain. It detects changes in header files, and then updates the respective generated files, including the file `NANDRAD_KeywordList.cpp`. It places the class-specific generated files into a subdirectory `ncg` (for **N**ANDRAD **C**ode **G**enerator files) where they are compiled just as any other cpp file of the project.

Since the generated code is plain C++ code, it can be parsed and debugged with any IDE just as regular code. Just remember:

Any changes made to generated code will be overwritten!

2. Functionality Example

2.1. Keyword support

You may have a class header, that looks as follows:

```
class Interval {
public:
    /*! Parameters. */
    enum para_t {
        IP_START,       // Keyword: Start       [d] 'Start time point'
        IP_END,         // Keyword: End         [d] 'End time point'
        IP_STEPSIZE,    // Keyword: StepSize    [h] 'StepSize'
        NUM_IP
    };

    ...

    /*! The parameters defining the interval. */
    IBK::Parameter   m_para[NUM_IP];
};
```

Here, you have a list of enumeration values. These are used to index one of the parameters in the member variable array `m_para`. In the code, you will use these enumerations like:

```
double endtime = m_para[Interval::IP_END].value;
```

Sometimes, however, you need the keyword as a string, or need to get the unit written in the comment line. So, you write code like:

```
// construct a parameter
m_para[Interval::IP_END].set(
        NANDRAD::KeywordList::Keyword("Interval::para_t", Interval::IP_END), // the keyword is used as name
        24, // value
        NANDRAD::KeywordList::Unit("Interval::para_t", Interval::IP_END) // the unit
    );
```

Normally, the unit is only meaningful for user interface representation, or when the value is printed to the user as default output/disply unit.

Also useful is the description, mostly in informative/error messages. Recommended would be:

```
if (condition_failed) {
    IBK::IBK_Message(IBK::FormatString("Parameter %1 (%2) is required.")
        .arg(KeywordList::Keyword("Interval::para_t", Interval::IP_END))
        .arg(KeywordList::Description("Interval::para_t", Interval::IP_END)) );
}
```

Also, when parsing user data, for example from format:

```
<IBK:Parameter name="Start" unit="d">36</IBK:Parameter>
<IBK:Parameter name="End" unit="d">72</IBK:Parameter>
```

You may need code like:

```
nameAttrib = getAttributeText(xmlTag, "name"); // might be 'End'

// resolve enumeration value
Interval::para_t p = (Interval::para_t)KeywordList::Enumeration("Interval::para_t", nameAttrib);
```

The code generator will now create the implementation of the functions:

- `KeywordList::Enumeration`
- `KeywordList::Description`
- `KeywordList::Keyword`
- `KeywordList::Unit`
- `KeywordList::Color` - meaningful for coloring types in user interfaces
- `KeywordList::DefaultValue` - can be used as default value in user interfaces
- `KeywordList::Count` - returns number of enumeration values in this category
- `KeywordList::KeywordExists` - convenience function, same as comparing result of `Enumeration()` with -1

2.2. Usage

The header file `NANDRAD_KeywordList.h` is always the same and can be included directory. The corresponding implementation file `NANDRAD_KeywordList.cpp` is generated in the same directory as the NANDRAD header files.

2.3. CMake automation

The automatic update of the keyword list is triggered by a custom rule in the NANDRAD CMake project file:

```
# collect a list of all header files of the Nandrad library
file( GLOB Nandrad_HDRS ${PROJECT_SOURCE_DIR}/../../src/*.h )

# run the NandradCodeGenerator tool whenever the header files have changed
# to update the NANDRAD_KeywordList.h and NANDRAD_KeywordList.cpp file
add_custom_command (
    OUTPUT   ${PROJECT_SOURCE_DIR}/../../src/NANDRAD_KeywordList.cpp
    DEPENDS  ${Nandrad_HDRS} NandradCodeGenerator
    COMMAND  NandradCodeGenerator
    ARGS     NANDRAD ${PROJECT_SOURCE_DIR}/../../src
)
```

where `NandradCodeGenerator` is built as part of the tool chain as well. The rule has all header files as dependencies so that any change in any header file will result in a call to the code generator. The code generator will then create the file `NANDRAD_KeywordList.cpp`.

3. Read/write to XML support, and other utility functions

A second task for the code generator is to create functions for serialization of data structures to XML files. Hereby, the TinyXML-library is used.

3.1. Read/write code

Since reading/writing XML code is pretty straight forward, much of this code writing can be generalized. Let's take a look at a simple example.

Class Sensor, with declarations of `readXML()` and `writeXML()` functions

```
class Sensor {
public:
    /*! Default constructor. */
    Sensor() : m_id(NANDRAD::INVALID_ID) {}

    void readXML(const TiXmlElement * element);
    TiXmlElement * writeXML(TiXmlElement * parent, bool detailedOutput) const;

    // *** PUBLIC MEMBER VARIABLES ***

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id;
    /*! Name of the measured quantity */
    std::string                         m_quantity;
};
```

The two members are written into file as follows:

```
<Sensor id="12">
    <Quantity>Temperature</Quantity>
</Sensor>
```

The implementation looks as follows:

Implementation of `Sensor::readXML()`

```
void Sensor::readXML(const TiXmlElement * element) {
    FUNCID(Sensor::readXML);

    try {
        // read attributes
        const TiXmlAttribute * attrib = TiXmlAttribute::attributeByName(element, "id");
        if (attrib != nullptr) {
            throw IBK::Exception(IBK::FormatString(XML_READ_ERROR).arg(element->Row()).arg(
                IBK::FormatString("Missing 'id' attribute.") ), FUNC_ID);
        }
        // convert attribute to value
        try {
            m_id = IBK::string2val<unsigned int>(attrib->ValueStr());
        } catch (IBK::Exception & ex) {
            throw IBK::Exception(ex, IBK::FormatString(XML_READ_ERROR).arg(element->Row()).arg(
                IBK::FormatString("Invalid value for 'id' attribute.") ), FUNC_ID);
        }

        // read parameters
        for (const TiXmlElement * c = element->FirstChildElement(); c; c = c->NextSiblingElement()) {
            // determine data based on element name
            std::string cname = c->Value();
            if (cname == "Quantity") {
                m_quantity = c->GetText();
                if (m_quantity.empty())
                    throw IBK::Exception(IBK::FormatString(XML_READ_ERROR).arg(element->Row()).arg(
                        "Tag 'Quantity' must not be empty."), FUNC_ID);
            }
            else {
                throw IBK::Exception(IBK::FormatString(XML_READ_ERROR).arg(element->Row()).arg(
                    IBK::FormatString("Undefined tag '%1'.").arg(cname) ), FUNC_ID);
            }
        }
    }
    catch (IBK::Exception & ex) {
        throw IBK::Exception(ex, IBK::FormatString("Error reading 'Sensor' element."), FUNC_ID);
    }
    catch (std::exception & ex) {
        throw IBK::Exception(IBK::FormatString("%1\nError reading 'Sensor' element.").arg(ex.what()), FUNC_ID);
    }
}
```

In this function there is a lot of code that is repeated nearly identical in all files of the data model. For example, reading of attributes, converting them to number values (including error checking), testing for known child elements (and error handling) and the outer exception catch clauses. Similarly, this looks for the `writeXML()` function.

Implementation of `Sensor::writeXML()`

```
TiXmlElement * Sensor::writeXML(TiXmlElement * parent, bool /*detailedOutput*/) const {
    TiXmlElement * e = new TiXmlElement("Sensor");
    parent->LinkEndChild(e);

    e->SetAttribute("id", IBK::val2string<unsigned int>(m_id));

    if (!m_quantity.empty())
        TiXmlElement::appendSingleAttributeElement(e, "Quantity", nullptr, std::string(), m_quantity);
    return e;
}
```

In order for the code generator to create these two functions, we need to add some comments to original class. Let's start with the `writeXML()` function:

1. XML-Element name is always the same as the class name, so that's known to the code generator
2. `m_id` should be written as attribute to `Sensor`, we need to tell the code generator that this is an attribute. The conversion `unsigned int` to string is known from the type declaration.
3. `m_quantity` should be written as child-element to `Sensor`. We also need to tell the code generator, that this is to be an element. Also, we need to write the code to check that the attribute must not be empty before writing it. We should tell the code generator somehow, that the value must not appear as empty string `""` in the xml file.

This leads to some annotation comments in the header file:

Class Sensor, with declarations of `readXML()` *and* `writeXML()` *functions*

```
class Sensor {
public:
    /*! Default constructor. */
    Sensor() : m_id(NANDRAD::INVALID_ID) {}

    void readXML(const TiXmlElement * element);
    TiXmlElement * writeXML(TiXmlElement * parent, bool detailedOutput) const;

    // *** PUBLIC MEMBER VARIABLES ***

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id;           // XML-A:
    /*! Name of the measured quantity */
    std::string                         m_quantity;     // XML-E:not-empty
};
```

The `// XML-A:` says: make this an attribute. The `// XML-E:` says: make this a child-element. The additional `not-empty` keyword means: do not allow this string to be empty (only meaningful for string data types, obviously).

With this given information, we can also generate the entire `readXML()` function.

Lastly, since the declaration for the `readXML()` and `writeXML()` functions are always the same, we can avoid typing errors by using a define:

Global code generator helpers

```
#define NANDRAD_READWRITE \
    void readXML(const TiXmlElement * element); \
    TiXmlElement * writeXML(TiXmlElement * parent, bool detailedOutput) const;
```

The header is now very short:

Class Sensor, using code generator

```
class Sensor {
public:
    /*! Default constructor. */
    Sensor() : m_id(NANDRAD::INVALID_ID) {}

    NANDRAD_READWRITE

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id;           // XML-A:
    /*! Name of the measured quantity */
    std::string                         m_quantity;     // XML-E:not-empty
};
```

The implementation file `NANDRAD_Sensor.cpp` is no longer needed and can be removed.

The code generator will create a file: `ncg_NANDRAD_Sensor.cpp` with the functions `Sensor::readXML()` and `Sensor::writeXML()`.

To avoid regenerating (and recompiling) all `ncg_*` files whenever *one* header file is modified, the code generator inspects the file creation times of the `ncg_XXX.cpp` file with the latest modification/creation data of the respective `ncg_XXX.h` file. The code is only generated, if header file is newer than the generated file.

3.2. "Is default" functionality

Suppose now a data member like `Sensor` is used in some other class in the hierarchy. There, the member variable may be uninitialized (sensor ID == `NANDRAD::INVALID_ID`). We want to avoid writing the `<Sensor>` xml tag alltogether in this case.

The typical code looks like:

```
// only write sensor if different from default
if (detailedOutput || m_sensor != Sensor()) {
    m_sensor.writeXML(element, detailedOutput);
}
```

For this code to work, we need a comparison operator defined in class `Sensor`, again something that the code generator provides:

```
bool Sensor::operator!=(const Sensor & other) const {
    if (m_id != other.m_id)             return true;
    if (m_quantity != other.m_quantity) return true;
    return false;
}
```

Again, the declaration is pretty standard and can be replaced by a define:

Global code generator helpers

```
#define NANDRAD_COMP(X) \
    bool operator!=(const X & other) const;
```

So the class declaration becomes:

Class Sensor, using code generator

```
class Sensor {
public:
    /*! Default constructor. */
    Sensor() : m_id(NANDRAD::INVALID_ID) {}

    NANDRAD_READWRITE
    NANDRAD_COMP(Sensor)

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id;           // XML-A:
    /*! Name of the measured quantity */
    std::string                         m_quantity;     // XML-E:not-empty
};
```

3.3. Custom read/write functionality

Sometimes, the default read/write code is not enough, because something special needs to be written/read as well. Here, you can simply use an additional define:

Global code generator helpers

```
#define NANDRAD_READWRITE_PRIVATE \
    void readXMLPrivate(const TiXmlElement * element); \
    TiXmlElement * writeXMLPrivate(TiXmlElement * parent, bool detailedOutput) const;
```

and implement `readXML()` and `writeXML()` manually, hereby re-using the auto-generated functionality. The header then looks like:

Class Sensor, using code generator with private read/write functions

```
class Sensor {
    NANDRAD_READWRITE_PRIVATE
public:
    /*! Default constructor. */
    Sensor() : m_id(NANDRAD::INVALID_ID) {}

    NANDRAD_READWRITE
    NANDRAD_COMP(Sensor)

    /*! Unique ID-number of the sensor.*/
    unsigned int                        m_id;           // XML-A:
    /*! Name of the measured quantity */
    std::string                         m_quantity;     // XML-E:not-empty
};
```

Implementation file `NANDRAD_Sensor.cpp`

```
void Sensor::readXML(const TiXmlElement * element) {
    // simply reuse generated code
    readXMLPrivate(element);

    // ... read other data from element
}


TiXmlElement * Sensor::writeXML(TiXmlElement * parent, bool detailedOutput) const {
    TiXmlElement * e = writeXMLPrivate(parent, detailedOutput);

    // .... append other data to e
    return e;
}
```

When the code generator creates `readXMLPrivate()` functions, the check for unknown elements is not generated (see original readXML-function, since it can be expected that other (custom) elements are present inside the parsed XML-tag.

4. Specifications

4.1. Keyword List Support

The parse requires fairly consistent code to be recognized, with the following rules. Look at the following example:

```
class MyClass {
public:


    enum parameterSet {
        PS_PARA1,      // Keyword: PARA1     'some lengthy description'
        PS_PARA2,      // Keyword: PARA2     [K] <#4512FF> {273.15} 'A temperature parameter'
        NUM_PS
    }

    enum otherPara_t {
        OP_P1,         // Keyword: P1
        OP_P2,         // Keyword: P2
        OP_P3,         // Keyword: P3
        NUM_OP
    }
...
}
```

Here are the rules/conventions (how the parser operates):

- a class scope is recognized by a string `class xxxx` (same line)
- an enum scope is recognized by a string `enum yyyy` (same line)
- a keyword specification is recognized by the string `// Keyword:` (with space between `//` and `Keyword:`!)
- either *all* enumeration values (except the line with `NUM_XXX`) must have a keyword specification, or *none* (the keyword spec is used to increment the enum counter)
- you **must not** assign a value to the enumeration like `MY_ENUM = 15,` - the parser does not support this format. With proper scoping, you won't need such assignments for parameter lists.

The parser isn't a c++ parser and does not know about comments. If the strings mentioned above are found inside a comment, the parser will not know the difference. As a consequence, the following code will confuse the parser and generate wrong keyword categories:

```
class MyClass {
public:

    /* Inside this
       class my stuff will work
       perfectly!
    */

    enum para_t {
    ...
    }
...
}
```

This will generate the keyword category `my::para_t` because `class my` is recognized as class scope. So, **do not do this**! Same applies to enum documentation.

Thankfully, documentation is to be placed above the class/enum declaration lines and should not interfere with the parsing.

When using class forward declarations, always put only the class declaration on a single line without comments afterwards:

```
// forward declarations
class OtherClass;
class OtherParentClass;
class YetAnotherClass;
```

The parser will detect forward declarations when the line is ended with a `;` character. Again, this should normally not be an issue, unless someone uses a forward declaration of a class *inside* a class scope.

4.2. Keyword Parameters

A keyword specification line has the following format:

```
KW_ENUM_VALUE,  // Keyword:   Keyword-Name  [unit]  <color>  {default value} 'description'
```

The `Keyword-Name` can be actually a list of white-space separated keywords that are used to convert to the enumeration value: for example:

```
SP_HEATCONDCOEFF, // Keyword: HEATCONDCOEFF ALPHA [W/m2K] 'Heat conduction coefficient'
```

Allows to convert strings `HEATCONDCOEFF` and `ALPHA` to enum value `SP_HEATCONDCOEFF`, but conversion from `SP_HEATCONDCOEFF` to string always yields the first keyword `HEATCONDCOEFF` in the list.

The remaining parameters *unit*, *color*, *default value* and *description* are **optional**. But if present, they must appear in the order shown above. This is just to avoid nesting problems and is strictly only required from the description, since this may potentially contain the characters `<>[]{}`.

The *default value* must be a floating point number in C locale format. Similarly as color and unit, this parameter is meaningful for user interfaces with somewhat generic parameter input handling.