Reimplementation of NXvalidate

Date: October, 8, 2015, Mark Koennecke

Requirements

NXvalidate is a tool to compare a NeXus file an application definition written in NXDL, a dialect of XML. The point is to verify that the NeXus file complies with the standard defined in the application definition. An earlier implementation of NXvalidate was based on XLST transforms and some Java code. This implementation never reached maturity and proved to be to complex to maintain. In addition there are new requirements:

- Some aspects of the validation process have become so complex that they can only be implemented in proper code. For example the validation of depends_on chains.
- There is a requirement to have the validation process available as a library to be included into other software. And to be implemented efficiently enough to validate a large number of NeXus files once they are created.

Therefore the NIAC decided to reimplement NXvalidate in an efficient programming language. As of now ANSI-C and python are languages to be considered.

TODO: decide upon the implementation language

Problem Analysis

At first glance the problem looks as simple as expressed in this morsel of pseudocode:

```
Load the NeXus and the NXDL file
    Foreach group in the NXDL file:
       Create a directory listing of the NXDL group
         Foreach entry in the NXDL group listing:
             If the entry is a group:
                      Find that group in the NeXus file, verify it,
                            and recurse into it
                 If the entry is a field:
                      Find the field in the NeXus file and compare it with the
                            NXDL description
```

Well, this is not really that simple. Moreover, at each step there are further complications. The general idea is to process as much of the NeXus and NXDL file as possible and report problems into the log along the way.

Loading NeXus and NXDL Files

This is simple for NeXus files. The only question to decide is if we use NAPI or the HDF-5 API.

TODO: Decide NAPI versus HDF-5. The author prefers NAPI.

For NXDL files, there are problems:

NXDL files can inherit from each other. Thus a number of NXDL files may need to be merged into a complete NXDL tree before doing the validation. Or we have to validate against the inheritance chain, thereby keeping track of fields and groups which have already been validated by the application definition of the derived class. The authors preference is to flatten the inheritance hierarchy out before validation.

TODO: opinions on this one?

How do we find NXDL files? Especially with inheritance an automatic way of locating NXDL files becomes a necessity. The extends property in the NXDL file only holds the name of the parent application definition. Related to this is another bootstrapping problem: by looking for the NXentry/definition or NXsubentry/definition fields nxvalidate can validate against the NeXus file alone. In this case we get an URL for the application definition. In principle we can make nxvalidate load the application definitions it requires from the NeXus WWW-presence. But this falls over when nxvalidate needs to operate in a protected network segment.

I checked all the current application definitions and all of them have the definition field.

TODO: make it a rule that the definition field is mandatory for an application definition.

TODO find a solution for this. The author suggests to make this overridable in the library by the application using the library. And to provide a default implementation looking for application definitions in a user provided directory. Providing a default implementation using a WWW-download carries the penalty of another dependency in the form of a WWW-client library.

We would be well advised not to implement our own XML parser. For C, the author suggests to use MXML as it is good enough and already is a NeXus dependency.

TODO: someone OK with this?

Validating Groups

The first issue to address here is where to start. Validation can be run at NXentry and NXsubentry level. Or both. There can be multiple instances in a given

NeXus file. Thus the default operation of nxvalidate should be to search for all NXentry and NXsubentry in a NeXus file and try to validate each. If there is no definition field exists in a NXentry or a NXsubentry or the referenced application definition cannot be found this is an error preventing further validation.

TODO: shall we provide for a means to validate a NXentry with a user supplied application definition?

Validation now implies that we need to compare the content of a group in a NXDL file with the content of the matching group in the NeXus file. Recursively following the hierarchy of the NXDL file.

One complication arises from the fact that groups can be specified either by name or by NXclass. This has to be accounted for.

When a mandatory field or group entry is missing, this is a validation error.

When an optional field or group is missing nxvalidate shall state this as information.

NeXus increasingly uses group attributes. If the application definitions asks for this, nxvalidate needs to test group attributes.

There may be additional fields or groups in a NeXus file, not required by the application definition. This ought to yield a warning or information message. We can check if the additional item is part of the base class of the current NeXus group and issue a differnt information then. But this implies that nxvalidate has access to the base class definitions too.

TODO: what is the desired behaviour with regards to additional fields?

Validating Fields

There are a number of things which need to be validated for fields except existence.

The first is the rank. This is an easy one.

The second is dimensionality. NXDL may use symbols for dimension fields. The first use of such a symbol in a NeXus file shall define the value for that actual dimension. This implies that we need to keep track of this information in a **symbol table**. Further uses of the same symbol should be looked up in the symbol table and the value compared. If there is a mismatch we have a validation error.

For number types NXDL uses symbols such as NX_FLOAT. These symbols must be compared with the actual field type, possibly using tables. Such tables should be externalized. But that would require another file which increases complexity for the user. Thus I suggest to hardcode this into the source files.

For some fields application definitions provide expected values. This needs to be validated too.

Then there must be a loop validating attributes against the NXDL description.

Validating Attributes

For the units attribute nxvalidate has the same problem as described above for number types.

TODO: where do I get a valid list of valid entries for unit symbols from? For example for NX_ANGLE? And all the others? The nxdlTypes.xsd just contains examples but no bullet proof list? Or do we defer the implementation of this test until we have a good list?

Otherwise, for each allowed NeXus attribute we need to check if it is well formed. This requires a special function for each NeXus attribute type.

If we have a depends_on attribute, we need to check that the other transformation attributes: vector, offset and transformation_type are present too.

Other Validations

NXvalidate may choose to validate the depends_on chain. This validation would start at the depends_on field in a group and verify that all elements in the depends_on chain are present and have the necessary attributes. And that the chain ends on a period.

TODO: do we want his?

API

The scope of the work will be NeXus validation library and an application for testing. I assume a C-library. If we decide for another implementation language please translate into the functional equivalent. The library API could look like this:

```
/*
 * NXvalidateInit initializes the validation library. Mainly
 * installs a default NXDL locator and a default logger.
 */
NXvalidateInit(void);
/* NXvalidateRun runs the validation. Outputs trouble to a log.
 * \param nexusFile The file to validate
 * \param nxdlFile The application deinition to validate against. Can be
 * NULL, then the validator searches the application definition in the
 * NeXus file.
 * \return 0 when validation succeeds, 1 else.
```

```
 */
int NXvalidateRun(char *nexusFile, char *nxdlFile);

typedef struct {
    char *neXusFile;
    char *nexusPath;
    char *nxdlFile;
    char *nxdlPath;
    int  severity;
} logEntry;

/*
 * This is the signature for a function processing a validation error or
 * information. The function cannot assume that it owns the data in the
 * logEntry structure.
 */
typedef void (*validateLogger)(logEntry l, void *userData);

/*
 * NXvalidateSetLogger sets a custom logger for validation
 * \param logger The logger to use
 * \param userData any data to pass to the custom logger
 */
void NXvalidateSetLogger(validateLogger logger, void *userData);

/*
 * This is the signature of a function retrieving an application definition.
 * It is supposed to return the content of the application definition as
 * string. Or NULL, if it cannot be located.
 */
typedef char* (RetrieveNXDL)(char *appDef);

/*
 * NXvalidateSetNXDLRetriever sets a user defined function for retrieving
 * application definition data.
 * \param retriever The retrieval function
 * \param userData Any data to pass to the retriever.
 */
void NXvalidateSetNXDLRetriever(RetrieveNXDL retriever, void *userData);
```

TODO: anyone content with this?

A possible upgrade is to define a validation context which holds the logger and retriever. This would allow differently configured validators to be run in different threads. I am not sure that we need to cater for this.

Implementation

I would implement along the following lines:

- ANSI-C for maximum portability
- MXML as XML parsing library.
- NAPI for NeXus file access.
- Use of a context structure for passing around context and state among functions. This makes the validator reentrant.
- Use my old stringdict for the symbol table. It is not efficient but good enough for the small umber of symbols expected.
- Hardcode mappings from NXDL types such as NX_FLOAT in code.

Otherwise this is a just a sizeable number of functions.

TODO: any comments?