Maintainer's guide to `libmusicxml2`

Jacques Menu

March 25, 2021

Abstract

This document presents the design principles and architecture of `libmusicxml2`, as well as information needed to maintain it. It is part of the `libmusicxml2` documentation, to be found at https://github.com/grame-cncm/libmusicxml/tree/lilypond/doc.

In the `libmusicxml2` library, the source code specific to `xml2ly` can be found at https://github.com/grame-cncm/libmusicxml/tree/lilypond/src/lilypond and https://github.com/grame-cncm/libmusicxml/tree/lilypond/src/interface.

All the examples mentioned can be downloaded from https://github.com/grame-cncm/libmusicxml/tree/lilypond/files/samples/musicxml. They are grouped by subject in sub-directories, such as `basic/HelloWorld.xml`.

Contents

List of Figures

Listings

1 Acknowledgements

Many thanks to Dominique Fober, the designer and maintainer of the `libmusicxml2` library!

2 Prerequisites

In order to maintain `libmusicxml2`, one needs to do the following:

- obtain a working knowledge of C++ programming. The code base of `libmusicxml2` uses classes, simple inheritance, and templates;
- study the architecture of `libmusicxml2`, which can be seen in figure 1, page 3, and is presented in more detail in https://github.com/grame-cncm/libmusicxml/blob/lilypond/doc/libmusicxmlArchitecture/libmusicxmlArchitecture.pdf

3 A historical note

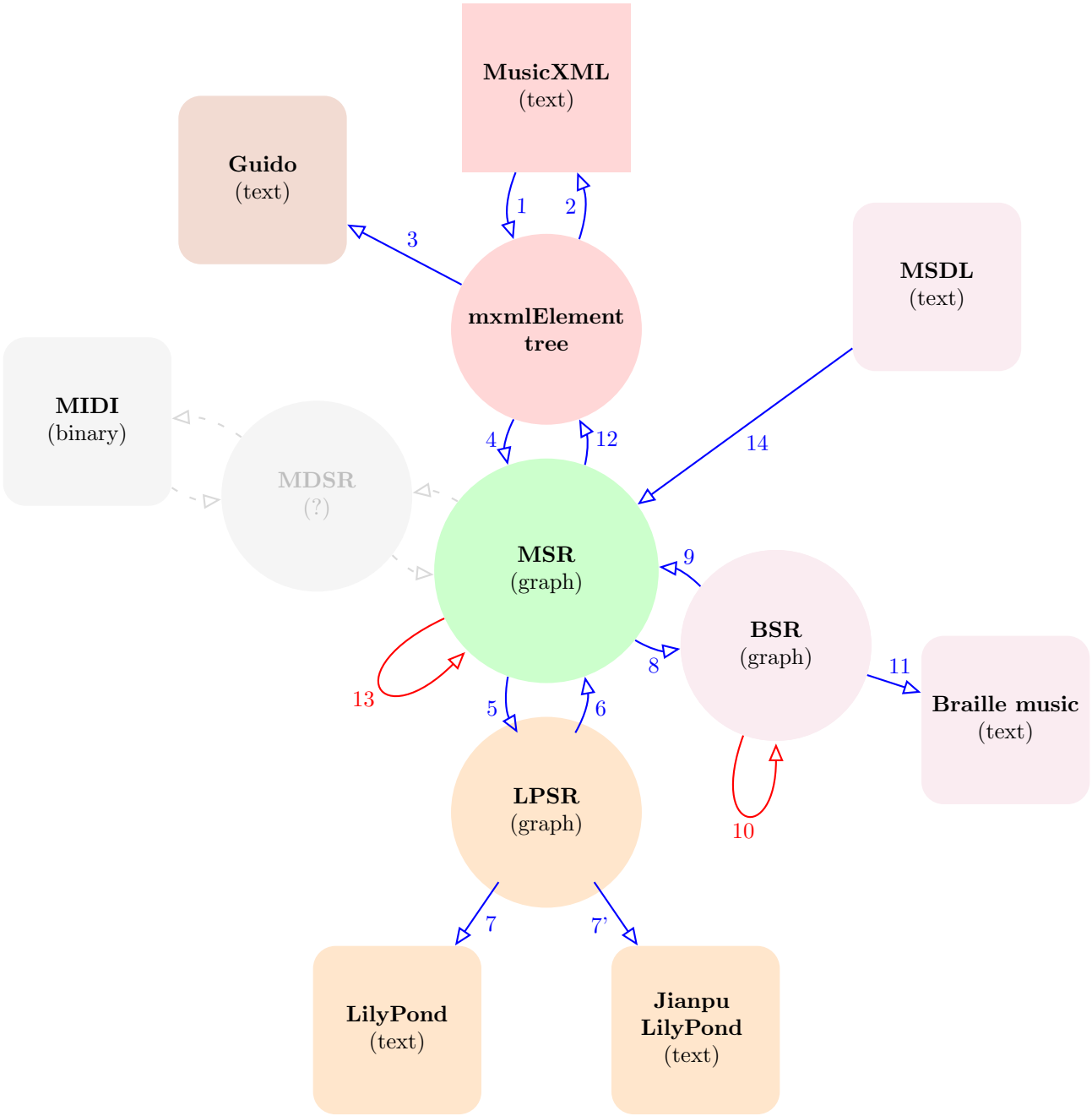
Dominique Fober created `libmusicxml2` long before this author had the need for a library to read MusicXML data, in order to convert it to LilyPond.

The `*.cpp` files in `samples` were examples of the use of the library. Among them, `xml2guido` has been used since in various contexts. The diagram in figure 1, page 3, was created afterwards, and it would then have consisted of only MusicXML, mxmlTree and Guido, with passes 1, 2 and 3.

When tackling the conversion of MusicXML to LilyPond, this author created MSR as the central internal representation for music score. It is meant to capture the musical contents of score in fine-grain detail, to meet the needs of creating LilyPond code first, and braille music later. The only change made to the existing MusicXML tree representation has been to add an input line number to `xmlElement`.

The conversion from MSR to BSR music was two-pass from the beginning, first creating a BSR representation with unlimited line and page lenghs, and then constraining that in a second

Figure 1: `libmusicxml2` architecture



BSR would take the numbers of cell per line and lines per page into account. This was frozen in autumn 1999 due to the lack of interest from the numerous persons and bodies that this author contacted about `xml2brl`. The current status is the braille output is that the cells per line and lines per page values are ignored.

The creation of MusicXML code from MSR data was then added to close a loop with MusicXML2xml, with the idea that it would make `libmusicxml2` a kind of swiss knife for textual representations of music scores.

Having implemented a number of computer languages in the past, this author was then tempted to design MSDL, which stands for Music Scores Description Language. The word *description* has been prefered to *programming*, because not all musicians have programming skills. The basic aim of MSDL is to provide a musician-oriented way to describe a score that can be converted to various target textual forms.

`samples/Mikrokosmos3Wandering.cpp` has been written to check that the MSR API was rich enough to go this way. The API was enriched along the way.

Having MSR, LPSR and BSR available, as well as the capability to generate MusicXML, LilyPond Guidoand Braille music, made writing a first draft of the MSDL compiler, with version

number 1.001, rather easy. The initial output target languages were MusicXML, LilyPond, MusicXML and Braille music.

This document contains technical information about the internal working of the code added to `libmusicxml2` by this author as their contribution to this great piece of software.

4 Programming style and conventions

4.1 File naming conventions

Most file names start with an identification of the context they belong to, such as '`oah`', '`mxmlTree`', '`msr`', '`lpsr`', '`lilypond`', '`bsr`', '`braille`', '`xml2ly`' and '`xml2brl`'.

The '`*Oah.*`' files handle the options and help for the corresponding context, such as '`xml2lyOah.h/.cpp`'.

The '`traceOah.h/.cpp`', '`musicXMLOah.h/.cpp`', '`extra`' and '`general`' context are about the corresponding help groups.

There are a couple of 'globlal' files not related to any particular context: '`utilities.h/.cpp`', '`messagesHandling.h/.cpp`' and '`version.h/.cpp`'.

4.2 Source code layout

The following text-editing conventions are used:

- tabs are not used before the first non-space character in a line, two spaces are used instead;
- the code is not tightly packed: declarations in classes have the members' names aligned vertically, with many spaces before them if needed, and empty lines are used to separate successive activities in methods.

4.3 Defensive programming

The code base of `xml2ly` is *defensive programming* oriented, which means that:

- identifiers are explicit and long if needed – only very local ones are short, such as iteration loops indexes;
- the code is organized in sections, with an initial comment documenting what the code does;
- C++11's `auto` declaration is only seldom used. Writing the explicit types in a large code base helps the maintainer mastering the code;
- '`msgAssert()`' is used to do sanity checks, such as detect a null pointer prior to using it.

4.4 JMI comments

Comments containg `JMI` indicates that the code may have to be reconsidered in the future, should a problem arise. They are removed when it becomes obvious that the code is fine. JMI was the acronym for the author's activity as a software contractor long time ago.

4.5 Code base structure

The `src` folder has the following structure:

- `cli` : tools for the command line
  - `Mikrokosmos3Wandering`
  - `xml2brl`
  - `xml2gmn`
  - `xml2ly`
  - `xml2xml`

- `elements` : creation of the C++ classes from the DTD's
- `factory` : mxmlTree data creation and sorting
- `files` : MusicXML files reading
- `generation` : support for various output kinds
  - `brailleGeneration`
  - `guidoGeneration`
  - `lilypondGeneration`
  - `msrGeneration`
  - `multiGeneration`
  - `mxmltreeGeneration`
- `guido` : Guido support and tools
- `interface` : API functions to interface for `libmusicxml2` and the various executables
- `lib` : basic types used by `libmusicxml2`
- `manpage` : man page generation from the OAH 'data'
- `midisharelight-master`
- `oah` : object-oriented Options And Help support
- `operations` : support for transposition and MusicXML queries
- `parser` : the MusicXML parser, based on `flex` and `bison`
- `passes` : code for single-pass converters
  - `bsr2braille`
  - `bsr2bsr`
  - `lpsr2lilypond`
  - `msr2bsr`
  - `msr2lpsr`
  - `msr2msr`
  - `msr2mxmltree`
  - `mxml2mxmltree`
  - `mxmltree2guido`
  - `mxmltree2msr`
  - `mxmltree2mxml`
- `representations` : the various internal representations used by `libmusicxml2`
  - `bsr`
  - lpsr
  - `msdl`
  - `msdr`
  - `msr`
  - `msrapi`
  - `mxmltree`
- `translators` : the multi-pass translator combining those in `passes`
  - `msdl2braille`
  - `msdl2guido`
  - `msdl2lilypond`
  - `msdl2musicxml`
  - `msdlcompiler`
  - `msr2braille`

  - msr2guido
  - msr2lilypond
  - msr2musicxml
  - musicxml2braille
  - musicxml2guido
  - musicxml2lilypond
  - musicxml2musicxml

- utilities : various utilities, include indented output streams, and version history support
- visitors : two-phase visitors support and example visitors code
- wae : multilingual Warnings And Errors support, including exceptions handling

The name 'lilypond' was chosen by Dominique long before work started on xmlTobrl.

There's a single 'lilypond' folder to contain MSR, LPSR, BSR, xml2ly and xml2brl, even though BSR and braille music are a distinct branch. This has been preferred by Dominique as the manager of libmusicxml2.

5 Options and help (OAH)

OAH

6 The trace facility

libmusicxml2 is instrumented with an optionnal, full-fledged trace facility, with numerous options to display what is going on when using the library. One can build the library with or without trace, which applies to the whole code base.

6.1 Activating the trace

Tracing is controlled by TRACING_IS_ENABLED, defined or nor in src/oah/enableTracingIfDesired.h:

```
#ifndef ___enableTracingIfDesired___
#define ___enableTracingIfDesired___

#ifndef TRACING_IS_ENABLED
  // comment the following definition if no tracing is desired
  #define TRACING_IS_ENABLED
#endif

#endif
```

This file should be included when the trace facility is used:

```
#include "enableTracingIfDesired.h"
#ifdef TRACING_IS_ENABLED
  #include "traceOah.h"
#endif
```

The files src/oah/traceOah.h/.cpp contain the options to the trace facility itself. They provide

For example, xml2ly -insider -help-trace produces:

```
menu@macbookprojm > xml2ly -insider -help-trace
--- Help for group "OAH Trace" ---
  OAH Trace (-ht, -help-trace) (use this option to show this group)
    There are trace options transversal to the successive passes,
      showing what's going on in the various translation activities.
```

```
      They're provided as a help to the maintainers, as well as for the
    curious.
      The options in this group can be quite verbose, use them with
    small input data!
      All of them imply '-tpasses, -trace-passes'.
  -------------------------
    Options handling trace    (-htoh, -help-trace-options-handling):
      -toah, -trace-oah
            Write a trace of options and help handling to standard
    error.
            This option should best appear early.
      -toahd, -trace-oah-details
            Write a trace of options and help handling with more
    details to standard error.
            This option should best appear early.
    Score to voices           (-htstv, -help-trace-score-to-voices):
      -t<SHORT_NAME>, -trace-<LONG_NAME>
            Trace SHORT_NAME/LONG_NAME in books to voices.
      The 10 known SHORT_NAMEs are:
        book, scores, pgroups, pgroupsd, parts, staves, st, schanges,
        .
      The 10 known LONG_NAMEs are:
        -books, -scores, -part-groups, -part-groups-details,
        -parts, -staves, -staff-details, -staff-changes, -voices and
        -voices-details.
... ... ...
```

6.2 Trace categories

6.3 Using the trace in practise

7 The two-phase visitors pattern

`libmusicxml2` uses a two-phase visitors pattern to traverse data structures such an `xmlElement tree` or an MSR description, handling each node in the structure in a systematic way. This is in contrast to a programmed top-down traversal. Such data structures traversals is actually data driven: a visitor can decide to 'see' only selected node types.

7.1 Basic mechanism

Visiting a node in a data structure is done in this order:

- first phase: visit the node for the fist time, top-down;
- visit the node contents, using the same two-phase visitors pattern;
- second phase: visit the node for the second time, bottom-up.

The first can be used to prepare data needed for the node contents visit, for example. Then the second phase can used such data, if relevant, as well as data created by the node contents visit, do consolidate the whole.

7.2 A first example

In `samples/countnotes.cpp`, counting the notes in MusicXML data needs only see `S_note` nodes. Class `countnotes` thus inherits only from a visitor for this type of node, and all the other node types are simply ignored.

The `countnotes::visitStart()` method only has to increment the notes count.

Listing 1: `samples/countnotes.cpp`

```
class countnotes :
```

```
  public visitor<S_note>
{
  public:
    int fCount;

    countnotes() : fCount (0) {}

    virtual ~countnotes () {}

    void visitStart ( S_note& elt )   { fCount++; }
};
```

7.3 A more complex example

Let's look at the `<scaling/>` MusicXML element:

```
    <scaling>
      <millimeters>7</millimeters>
      <tenths>40</tenths>
    </scaling>
```

It contains a `<millimeter/>` and a `<tenth/>` element. The latter two don't contain any other elements, so `visitStart()` is enough for them.

There is nothing to do on the visit start upon `<scaling/>`, so there is no such method. On the visit end upon `<scaling/>`, though, the values grabbed from the `<millimeter/>` and `<tenth/>` elements are used to create the `msrScaling` description.

Should a visit start method have been written, the execution order would have been:

```
  mxmlTree2msrTranslator::visitStart ( S_scaling& elt)
    mxmlTree2msrTranslator::visitStart ( S_millimeters& elt )
    mxmlTree2msrTranslator::visitStart ( S_tenths& elt )
  mxmlTree2msrTranslator::visitEnd ( S_scaling& elt)
```

or, depending on the order in which the subelements of `<scaling/>` are visited:

```
  mxmlTree2msrTranslator::visitStart ( S_scaling& elt)
    mxmlTree2msrTranslator::visitStart ( S_tenths& elt )
    mxmlTree2msrTranslator::visitStart ( S_millimeters& elt )
  mxmlTree2msrTranslator::visitEnd ( S_scaling& elt)
```

In `src/passes/mxmltree2msr/mxmlTree2msrTranslator.cpp`, visiting a `<scaling/>` element is handled this way:

Listing 2: Visiting `<scaling/>`

```
void mxmlTree2msrTranslator::visitStart ( S_millimeters& elt )
{
#ifdef TRACING_IS_ENABLED
  if (gGlobalMxmlTreeOahGroup->getTraceMusicXMLTreeVisitors ()) {
    gLogStream <<
      "--> Start visiting S_millimeters" <<
      ", line " << elt->getInputLineNumber () <<
      endl;
  }
#endif

  fCurrentMillimeters = (float)(*elt);
}

void mxmlTree2msrTranslator::visitStart ( S_tenths& elt )
{
#ifdef TRACING_IS_ENABLED
  if (gGlobalMxmlTreeOahGroup->getTraceMusicXMLTreeVisitors ()) {
    gLogStream <<
```

```
      "--> Start visiting S_tenths" <<
      ", line " << elt->getInputLineNumber () <<
      endl;
  }
#endif

  fCurrentTenths = (float)(*elt);
}

void mxmlTree2msrTranslator::visitEnd ( S_scaling& elt)
{
  int inputLineNumber =
    elt->getInputLineNumber ();

#ifdef TRACING_IS_ENABLED
  if (gGlobalMxmlTreeOahGroup->getTraceMusicXMLTreeVisitors ()) {
    gLogStream <<
      "--> End visiting S_scaling" <<
      ", line " << inputLineNumber <<
      endl;
  }
#endif

  // create a scaling
  S_msrScaling
    scaling =
      msrScaling::create (
        inputLineNumber,
        fCurrentMillimeters,
        fCurrentTenths);

#ifdef TRACING_IS_ENABLED
  if (gGlobalTraceOahGroup->getTraceGeometry ()) {
    gLogStream <<
      "There are " << fCurrentTenths <<
      " tenths for " <<  fCurrentMillimeters <<
      endl;
  }
#endif

  // set the MSR score's scaling
  fMsrScore->
    setScaling (scaling);
}
```

The order of the visit of a node's subnodes is programmed in `browseData()` methods, such as:

Listing 3: `msrDoubleTremolo::browseData (basevisitor* v)`

```
void msrDoubleTremolo::browseData (basevisitor* v)
{
  if (fDoubleTremoloFirstElement) {
    // browse the first element
    msrBrowser<msrElement> browser (v);
    browser.browse (*fDoubleTremoloFirstElement);
  }

  if (fDoubleTremoloSecondElement) {
    // browse the second element
    msrBrowser<msrElement> browser (v);
    browser.browse (*fDoubleTremoloSecondElement);
  }
}
```

8 The MSR classes inheritance and use

The picture at figure 2, page 12, shows the hierarchy of the main MSR classes. The colors are used as follows:

- green: a score element that is expected to be found in a score representation, such as `msrStaff and msrChord`;
- pink: a element needed in MSR to structure the representation, such as `msrSegment` and `msrSyllable`;
- yellow: a base class with name `msr*Element` for elements that can be used in another class, such as `msrVoiceElement`;
- red: a link from a class to its base class. For example, `msrPart` is derived from `msrPartGroupElement`;
- blue: a link from a class to another that uses smart pointers to one or more instances the former. For example, an `msrTuplet` may be an element of an `msrGraceNotesGroup`.

When not shown for clarity, the common base class of all these classes is `msrElement`, that contains an integer input line number.

The `otherMeasureElements` classes are:

- part names:
  - `msrPartNameDisplay`
  - `msrPartAbbreviationDisplay`
- bars:
  - `msrBarCheck`
  - `msrBarNumberCheck`
  - `msrBarline`
  - `msrHiddenMeasureAndBarline`
- breaks:
  - `msrLineBreak`
  - `msrPageBreak`
- notes:
  - `msrVoiceStaffChange`
  - `msrOctaveShift`
- clefs, keys, times, tempo:
  - `msrClef`
  - `msrKey`
  - `msrTime`
  - `msrTempo`
- instruments:
  - `msrStaffDetails`
  - `msrScordatura`
  - `msrAccordionRegistration`
  - `msrHarpPedalsTuning`
  - `msrPedal`
  - `msrDamp`
  - `msrDampAll`
- lyrics:
  - `msrSyllable`
- rehearsals, segno and coda:

  - msrRehearsal
  - msrSegno
  - msrDalSegno
  - msrCoda
- others:
  - msrPrintLayout
  - msrEyeGlasses
  - msrStaffLevelElement
  - msrTranspose
  - msrTupletElement

9 xml2ly

9.1 Why xml2ly?

MusicXML (*Music eXtended Markup Language*) is a specification language meant to represent music scores by texts, readable both by humans and computers. It has been designed by the W3C Music Notation Community Group (https://www.w3.org/community/music-notation/) to help sharing music score files between applications, through export and import mechanisms.

The homepage to MusicXML is https://www.musicxml.com.

MusicXML data contains very detailed information about the music score, and it is quite verbose by nature. This makes creating such data by hand quite difficult, and this is done by applications actually.

The MusicXML data is not systematically checked for correctness. Checks are done, however, to ensure it won't crash due to missing values.

9.2 What xml2ly does

10 Passes organization

11 Translating MusicXML data to an mxmlTree

12 Translating an mxmlTree to an MSR

13 Translating an an MSR to a LPSR

This converter embeds a specific converter of MSR to MSR, to circumvent the famous LilyPond issue #34.

14 Translating an an MSR to another MSR

15 Translating an an LPSR to a LilyPond code

16 Overview of xml2brl

xml2brl is mentioned here, but not described in detail.

Figure 2: MSR classes hierarchy