libmusicxml2 architecture overview

xml2guido v2.3, xml2ly v0.9, xml2brl v0.01

September 4, 2020

This document shows the architecture of the libmusicxml2 library, to be found at https://github.com/grame-cncm/libmusicxml/tree/lilypond.

libmusicxml2 is written in C++11 and provides a set of music scores representations and converters between various textual music scores formats. Building it only requires a C++11 compiler and cmake.

1 Architecture

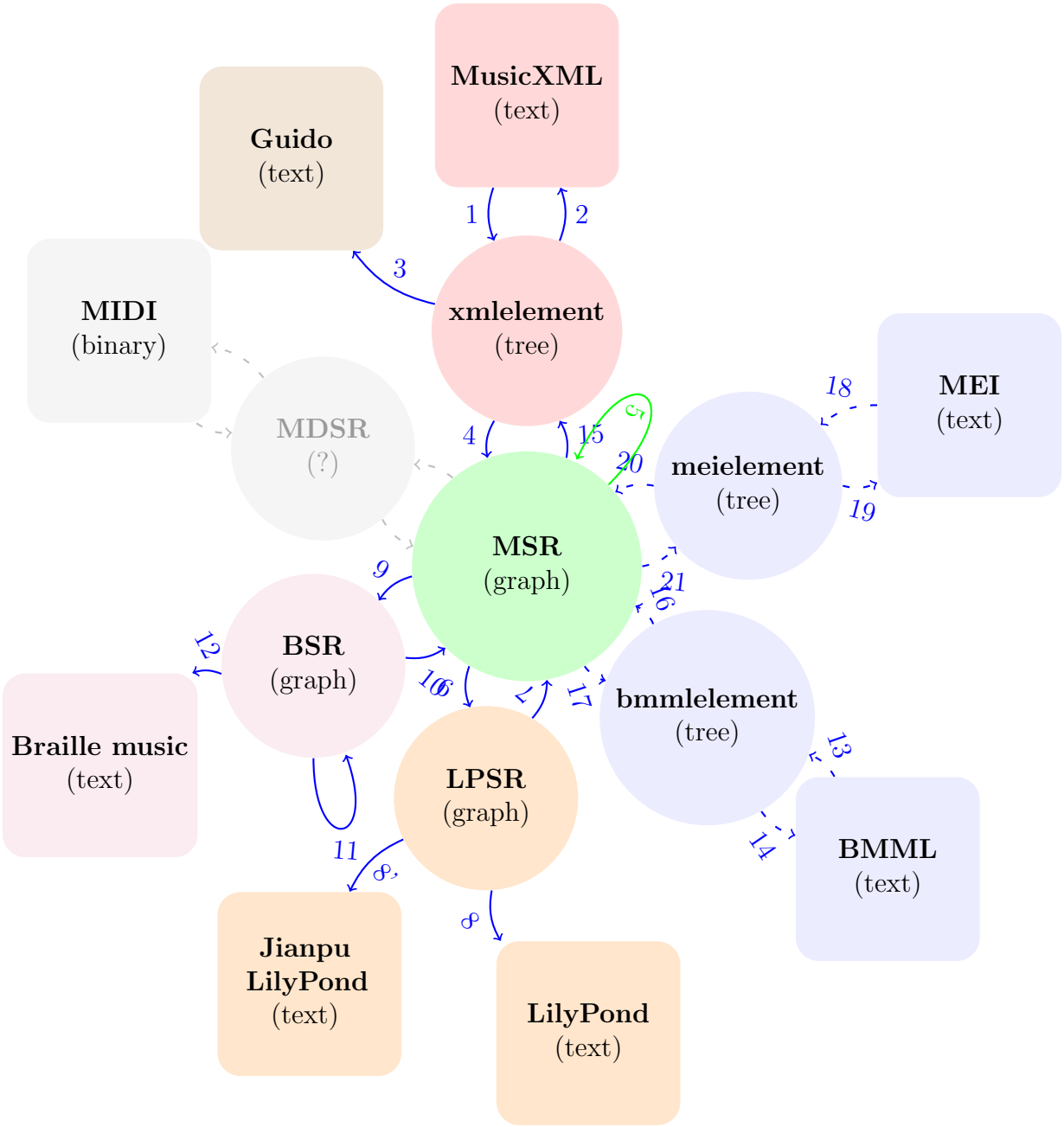
The picture at figure 1, page 2, shows how libmusicxml2 is structured. The dimmed, dashed arrows indicate items not yet available. The numbered arrows show the existing conversions between formats and representations.

2 Formats and representations

The formats supported by libmusicxml2 are:

Format	Description
MusicXML	a text containg markups such as <part-list>, <time> and <note>;
Guido	a text containg markups such as \barFormat, \tempo and \crescEnd;
LilyPond	a text containg commands such as \header, \override and \transpose;
Jianpu LilyPond	a text containg LilyPond commands and the use of lilypond-Jianpu (https://github.com/nybbs2003/lilypond-Jianpu/jianpu10a.ly) to obtain a Jianpu (numbered) score instead of the default western notation. lilypond-Jianpu should be accessible to LilyPond for it to produce the score;
Braille music	a text containg 6-dot cells, as described in http://www.brailleauthority.org/music/Music_Braille_Code_2015.pdf;
BMML	a text containg elements such as <score>, <score_header>, <part_list>, <part_data>, <score_data> – under development.
MEI	a text containg elements such as <meiHead>, <scoreDef> and <multiRest> – under development.

Figure 1: `libmusicxml2` architecture



The representations used by `libmusicxml2` are:

Representation	Description
MSR	Music Score Representation, in terms of part groups, parts, staves, voices, notes, etc. This is the heart of the multi-language converters provided by `libmusicxml2`;
xmlelement tree	a tree representing the MusicXML markups such as `<part-list>`, `<time>` and `<note>`;
bmmlelement tree	a tree representing the BMML markups such as `<part-list>`, `<time>` and `<note>`;
meielement tree	a tree representing the MEI markups such as `<part-list>`, `<time>` and `<note>`;
LPSR	LilyPond Score Representation, i.e. MSR plus LilyPond-specific items such as `\score` blocks;

BSR	Braille Score Representation, with pages, lines and 6-dots cells;
MDSR	MIDI Score Representation, to be designed.

3 Conversion passes

The numbers in the picture refer to so-called passes (compiler writing terminology), i.e. components of the library that convert a representation into another. The passes are numbered in the order they were added to the library:

Passes	Description
1	reads MusicXML data from a file or from standard input is '-' is supplied as the file name, and creates an xmlelement tree containg the same data;
2	converts an xmlelement tree into MusicXML data. This is a mere 'print()' operation;
3	converts an xmlelement tree into Guido text code, and writes it to standard output;
4	converts an xmlelement tree into and MSR representation. MusicXML represents how a score is to be drawn, while MSR represents the musical contents with great detail. This pass actually consists in two sub-passes: the first one builds an MSR skeleton containing empty voices and stanzas, and the second one the fills this with all the rest;
5	converts an MSR representation into another one, built from scratch. This allows the new representation to be different than the original one, for example to change the score after is has been scanned and exported as MusicXML data, or to add skip (invisible) notes to avoid the LilyPond issue #34. For simplicity and efficiency reasons, this pass is not present as such, but 'merges' within passes 6 and 9;
6	converts an MSR representation into an LPSR representation, which contains an MSR component build from the original MSR (pass 5). The BSR contains LilyPond-specific representations such as `\layout`, `\paper`, and `\score` blocks;
7	converts an LPSR representation into an MSR representation. There is nothing to do, since the former contains the latter as a component;
8	converts an LPSR representation into LilyPond text code, and writes it to standard output;
8'	converts an LPSR representation into LilyPond text code using `lilypond-Jianpu`, and writes it to standard output. This pass is run with `xml2ly -jianpu`;
9	converts an MSR representation into a BSR representation, which contains an MSR component build from the original MSR (pass 5). The BSR contains Braille music-specific representations such as pages, lines and 6-dot cells. The lines and pages are virtual, i.e. not limited in length;
10	converts a BSR representation into an MSR representation. There is nothing to do, since the former contains the latter as a component;
11	converts a BSR representation into another one, to adapt the number of cells per line and lines per page from virtual to physical. Currently, the result is a mere clone;
12	converts a BSR representation into Braille music text, and writes it to standard output;
13	reads BMML data from a file or from standard input is '-' is supplied as the file name, and creates an bmmlelement tree containg the same data;
14	converts an bmmlelement tree into BMML data. This is a mere 'print()' operation;
16	converts an bmmlelement tree into MSR data – in design phase;

17 converts MSR data into an bmmlelement tree – in design phase.

15 converts an MSR representation into an xmlelement tree – ongoing work;

18 reads MEI data from a file or from standard input is '-' is supplied as the file name, and creates an meielement tree containg the same data;

19 converts an meielement tree into MEI data. This is a mere 'print()' operation;

20 converts an meielement tree into MSR data – in design phase;

21 converts MSR data into an meielement tree – in design phase.

4 Executable converters

The converters provided by `libmusicxml2` are in the form of functions. Executable command-line applications are also supplied. They are shown in the table below, in which the ones not yet planned or under development are dimmed:

Input format	Output format						
	MusicXML	Guido	LilyPond	Jianpu LilyPond	Braille music	BMML	MEI
MusicXML	`xml2xml`	`xml2guido`	`xml2ly`	`xml2ly -jianpu`	`xml2brl`	`xml2bmml`	`xml2mei`
BMML	`bmml2xml`	`bmml2guido`	`bmml2ly`	`bmml2ly -jianpu`	`bmml2brl`	`bmml2bmml`	`bmml2mei`
MEI	`mei2xml`	`mei2guido`	`mei2ly`	`mei2ly -jianpu`	`mei2brl`	`mei2bmml`	`mei2mei`

The executables available, planned or under development in `libmusicxml2` are:

Converter	Description
`xml2guido`	converts MusicXML data to Guido code, using passes: 1 ⇒ 3
`xml2xml`	converts MusicXML data to MSR and back. This is useful to modify the data to suit the user's needs, such as fixing score scanning software limitations or to enhance the data: 1 ⇒ 4 ⇒ 15 ⇒ 2
`xml2ly`	performs the 4 hops from MusicXML to LilyPond to translate the former into the latter, using these passes: 1 ⇒ 4 ⇒ 6 ⇒ 8 The `-jianpu` option is supplied to create Jianpu (numbered) scores, in which the notes are represented by numbers instead of graphics; The `-loop` option is supplied to create MusicXML data back from the MusicXML data, as is done by `xml2xml`;
`xml2brl`	performs the 5 hops from MusicXML to Braille music to translate the former into the latter (draft); 1 ⇒ 4 ⇒ 5 ⇒ 9 ⇒ 11 ⇒ 12
`bmml2guido`	converts BMML data to Guido code, using passes: 13 ⇒ 16 ⇒ 15 ⇒ 3
`bmml2xml`	converts BMML data to MusicXML code, using passes: 13 ⇒ 16 ⇒ 15 ⇒ 2

`bmml2bmml` converts BMML data to MSR and back:
13 ⇒ 16 ⇒ 16 ⇒ 14

`bmml2ly` performs the 4 hops from BMML to LilyPond to translate the former into the latter, using these passes:
13 ⇒ 16 ⇒ 6 ⇒ 8
The `-jianpu` option is supplied to create Jianpu (numbered) scores, in which the notes are represented by numbers instead of graphics;
The `-loop` option is supplied to create BMML data back from the MEI data, as is done by `bmml2bmml`;

`bmml2brl` performs the 5 hops from BMML to Braille music to translate the former into the latter (draft):
13 ⇒ 16 ⇒ 9 ⇒ 11 ⇒ 12

`mei2guido` converts MEI data to Guido code, using passes:
18 ⇒ 20 ⇒ 15 ⇒ 3

`mei2xml` converts MEI data to Guido code, using passes:
18 ⇒ 20 ⇒ 15 ⇒ 2

`mei2mei` converts MEI data to MSR and back:
18 ⇒ 20 ⇒ 21 ⇒ 19

`mei2ly` performs the 4 hops from MEI to LilyPond to translate the former into the latter, using these passes:
18 ⇒ 20 ⇒ 6 ⇒ 8
The `-jianpu` option is supplied to create Jianpu (numbered) scores, in which the notes are represented by numbers instead of graphics;
The `-loop` option is supplied to create MEI data back from the MEI data, as is done by `mei2mei`;

`mei2brl` performs the 5 hops from MEI to Braille music to translate the former into the latter (draft):
18 ⇒ 20 ⇒ 9 ⇒ 11 ⇒ 12

`xml2bmml` performs the 4 hops from MusicXML to BMML to translate the former into the latter:
1 ⇒ 4 ⇒ 17 ⇒ 14

`xml2mei` performs the 4 hops from MusicXML to MEI to translate the former into the latter:
1 ⇒ 4 ⇒ 21 ⇒ 19

5 Basic tools

`libmusicxml2` supplies a number of basic tools using its features:

- `RandomMusic` generates an xmlelement tree containing random music, and writes it as MusicXML to standard output;
- `RandomChords` generates an xmlelement tree containing random two-note chords, and writes it as MusicXML to standard output;
- `MusicAndHarmonies.cpp` generates an xmlelement tree containing notes and harmonies, and writes it as MusicXML to standard output;
- other programs such as `countnotes`, `xmltranspose` and `partsummary` demonstrate the possibilities of the library, in particular those of the two-phase visitors pattern it uses.

It is to be noted that:

- LilyPond provides `midi2ly` to translate MIDI files to LilyPond code;
- LilyPond can generate MIDI files from its input.

6 Options and help

Having many executables with many options makes options and help handling a challenge.
This is why `libmusicxml2` uses it own OAH (Options And Help) object oriented infrastructure.

OAH organizes the options and the corresponding help in a hierarchy of groups, sub-groups and so-called atoms. OAH is introspective, thus help can be obtained for every group, sub-group or atom at will.

Each pass supplies a OAH group, containing its own options and help. The executable converters then aggregate the OAH groups of the passes they are composed of to offer their options and help to the user.