Plaintext “Ecosystem” for Musicological Work
Proposed Additions to LilyPond and LaTeX

Urs Liska

Version 1.1
January 10, 2014

Introduction

Recently I have published a number of articles which more or less explicitly propose the use of plain text driven toolchains for serious scholarly work in the musical disciplines. In particular I'm talking of the essay "Plain Text Files in Music", published at the *Scores of Beauty* blog[1]. To some extent the present paper assumes some familiarity with the contents of that essay or its German version (which is rather compressed but explicitly focused on scholarly work).

Plain text tools like the LilyPond engraving software and the LaTeX typesetting system can be combined to very integrated, collaborative workflows using version control software such as Git. While existing possibilities are already very impressive I believe that from this perspective there *are* some ideas for possible improvements. This should in no way be mistaken for a list of shortcomings, quite the contrary. The fact that text based toolchains *do* allow for such considerations is awesome in itself, and I believe that through a number of developments one could head towards a near-perfect ecosystem with tools for completely integrated scholarly workflows.

I haven't reached the point yet to seriously think about interfacing with current trends in research concerned with the standardization of music encoding or real single-source and/or cross-media publishing. But take my word for it that this isn't for lack of interest, this area will be tackled too when time comes. I'm convinced that LilyPond and LaTeX are equally suitable for these challenges.

In this document I will outline a number of projects that I consider interesting or important with respect to integrating plain text workflows in scholarly work. Some of them are already in the planning or implementation phase, some are mere ideas that could be realized at any point if sufficient interest in them arises. Some of the ideas are small-scale projects that can be implemented "along the way", but there are also projects that would require a considerable amount of time, developer's power – and money. These would be best tackled in the context of larger (research oriented?) projects.

[1] `http://lilypondblog.org/2013/07/plain-text-files-in-music/`

Contents

1 Additions to LilyPond

1.1 `\annotate`

`\annotate` is intended be a tool to annotate music in the LilyPond input file. This can have considerable impact on the preparation of scholarly editions.

Recently I had to proof-read a scholarly edition I was about to finish. For any difference between the two I had to determine if there was an entry in the critical report, and if there wasn't any decide whether to fix the new edition or add an entry in the report. This involved referring to the old edition and the new edition, then the revision report *and* the section with remarks (containing the more important readings/corrections). Particularly due to the sheer number of issues this was a very tedious and also error-prone process, and I thought about how much easier it would be if I could edit the remarks directly in the input file.

I had already thought of such a project earlier, but this experience convinced me of its urgency and necessity. To achieve something useful would have several parts:

Design an Interface The first important step is to design an interface to *insert annotations* of all kinds in the LilyPond input file. Apart from documentation these annotations also serve as a communication channel between the participating editors. Annotation types are for example technical remarks ("How can I realize this notation?"), musicological discussion ("There is a pencil marking in the manuscript. Should this go into the score or only into the report?"), TODO items ("Please review manuscript B") or finally real entries for the critical report. Particularly I suggest starting annotations as initial remarks and gradually (but in the same place) elaborating them into critical remarks throughout the editing process.

There has already been significant progress with this stage, and it's quite clear how this interface will look like. For example it will be possible to write something like

```
\annotate \with {
  type = "critical-remark"
  context = "vc1"
  source = "A1"
  author = "Urs Liska"
  date = "2013-06-06"
  format = "plain"
  message = "originally g, corrected with pencil."
}
Accidental
ges1--
```

annotating the ♭ before the whole note `ges` with a “critical remark”. It is planned to have a set of recognized fields such as e. g. `type` or `message`, and allowing project maintainers to agree upon any other free-form fields (for example it might be useful to define a field “office” to denote which of several collaborating institutes should handle the issue). Recognized fields may be mandatory or optional, mandatory fields will have default values assigned to them. The `format` field should allow marking the text format of the message, e. g. plain text, Markdown, HTML or LaTeX.

Note: There are ideas to interface LilyPond with MEI[1], and it should be discussed if the `\annotate` design could be in any way oriented towards a clean integration with this encoding standard.

Process Annotations The second step is to process the annotations, which will be done during score compilation. There are several possible operations which will be configurable with switches:

- Print annotations to the console during compilation. This way other editors of the score will notice them. Additionally it’s then possible to click on the messages to navigate to the annotation in the input file.
- Highlight annotated objects through colors.
- Print annotations in the score, e. g. as ballon tips.
  These visual annotations won’t affect the layout, but may disturb reading of the score . Therefore it is desirable to output them to separate PDF layers (see 3.3).
- Generate external annotation files that can be used by the score editors or post-processed by secondary tools. These files are either (optionally) structured by the annotation type or split by them (i. e. producing one file for TODOs, one for generic annotations, one for critical remarks etc.). They can have different formats, e. g. PDF, HTML or XML, so they can be used in different ways. But all will have `textedit` links attached to each annotation so a reader can click on an annotation and be brought directly to the corresponding position in the input file. Let me stress this once more: compiling a score will result in (one or separate) file(s) with clickable lists of TODO items, editorial questions, critical remarks etc.

Development of this hasn’t started yet, but it is clear that it will be done with Scheme, LilyPond’s built-in extension language. It is not clear yet whether all the mentioned external files will be created directly from within the LilyPond compilation run. Maybe it is more versatile of efficient to produce only a generic file that can be post-processed by an external (e. g. Python) script – which could of course be triggered from within LilyPond. The implementation of such a secondary tool should reuse existing technology as extensively as possible (e. g. the `pandoc` format converter[2]). Of course such a tool could do anything programmers can invent; for example one could think of a program that sends emails to affected team members whenever there is a certain change in annotations.

One particularly important use of the results will be the automatic inclusion of the list of critical remarks in a LaTeX document with the help of the `muscritreport` package (see 2.3 on page 9).

To sum up: `\annotate` will make it possible to produce a critical edition completely *within the actual score document!* If you consider the impact that version control can have for collaborative

[1] `http://www.music-encoding.org`
[2] `http://johnmacfarlane.net/pandoc/`

workflows this statement gains even more importance. And on top of this the effect will be increased with the development of a Frescobaldi interface to edit annotations directly from its music view (see 4.1 on page 13).

1.2 Editorial toolbox

During the preparation of a scholarly edition[3] I developed a set of commands used for editorial aspects. It's intent is two-fold: marking items in the score as "editor's amendments" in a consistent way and providing tools to handle them during the edition process. Actually `\annotate` and this are two closely related projects with a large intersection so they will be developed together, but I'll describe it separately here. Of course these editor's amendments will share the functionality with the previously described annotations, so you'll also get automatic lists of these amendments. And it's strongly desirable to have a single annotation taking care of the visible marking in the score *and* the entry for the critical report.

The development of this toolkit should be backed by the most broad scholarly discussion possible. The availability of diacritical marks in computer based engraving has always been an obstacle for scholarly editions. Developers of commercial notation programs usually have limited interest in spending resources for this marginal use case, but with LilyPond it is comparably simple to add a comprehensive set of commands.[4]

The commands of this toolkit will affect the appearance of notational objects in a predefined way (e. g. make them smaller, print them dashed etc.) or add diacritical marks like different kinds of parentheses. Similar to character styles in a DTP application such commands ensure consistent appearance throughout the score – and these styles can be modified as part of a "house style", guaranteeing consistency throughout multiple editions.

In combination with draft mode settings (see 3.2 on page 11) the affected items are highlighted by coloring during the edition process. And the icing on the cake will be that this will all be accessible through the graphical interface in Frescobaldi – one more component on the way of preparing complete editions within one scoring environment.

1.3 Functional Analysis

One engraving task that can't be done in a satisfactory manner yet is the typesetting of functional analysis symbols. While this isn't of great relevance for editions it is quite important for engraving examples for musicological books or preparing teaching/exam materials.

Depending on the desired quality level, completeness and versatility this may range from a small-scale to a quite ambitious project. For example it may be comparably trivial if one is content to attach analysis symbols to notes like e. g. articulations or dynamics. But if one wants the symbols to to align on a single baseline like lyrics do it becomes more complex, particularly when one wants to have several layers, for example in the case of modulations.

[3] `http://lilypondblog.org/category/fried-songs`

[4] See for example these posts about the definition of new historical ornaments: `http://lilypondblog.org/author/nsceaux/`.

In my “former life” I had a TrueType font – *FinalAnalyse* – to typeset analysis symbols with Finale, but a font based approach is out of question. The solution should definitely allow the use of *any* font to achieve a coherent appearance while being extremely flexible.

For single line analyses it might be sufficient to create commands that can be used in a `\Lyrics` context, but with a second layer this would seem too hackish (the content of the different “stanzas” would have to skip the music until they are being used, so the user would have manually to keep track of this. Actually I assume one would have to define a new context that can reside below or above any staff, that is aligned to a baseline like the lyrics and that allows splitting to several layers with optional brackets or boxes to mark the transition.

There haven’t been made any real attempts in this direction so far, but the project should be started together with a corresponding LaTeX package (see 2.4). It should also be able to handle scale-step analysis symbols.

2 Additions to LaTeX

2.1 lilyglyphs

`lilyglyphs`[1] is a package that can insert *any* notational element from LilyPond in the continuous text of LaTeX documents. It is particularly useful for typesetting critical reports, analytical essays or educational material.

`lilyglyphs` has already been released and is available in some LaTeX distributions. One aspect of future development is to increase the coverage of predefined commands (you can always use ad-hoc symbols by accessing the glyphs from LilyPond's music notation font directly but predefined commands are more convenient). Another plan is to extend the package to be compatible with the projected SMuFL standard[2] and to include Steinberg's *Bravura* font[3].

2.2 musicexamples

`musicexamples`[4] is a package that manages music examples within LaTeX documents. In its current state it can already print music examples in a variety of ways while managing captions, references and a list of musicexamples. Examples can be fixed or "floating" (i. e. floating through the text to find the optimal position). Specific care has been taken to handle full- and multi-page examples, with explicit support for scores that should explicitly start on odd or even pages.

One development plan is to make the package aware of changes to the used music examples (that is, over those created with LilyPond) and recompile them if necessary. Another item on the TODO list is to provide specific support on the LilyPond side to streamline the creation of consistent music examples for a given publication (series).

One major project will be the ability to manage LilyPond source code directly in the LaTeX document through the built-in extension language Lua. LaTeX will then deteremine itself if a music example has to be recompiled with LilyPond and perform the necessary housekeeping as part of the regular LaTeX run. The intention is to create a replacement to `lilypond-book`, a Python script shipped with LilyPond. There are several advantages of such a built-in solution over the external script: It will be possible to directly compile the document one is editing, without the need for the external script preprocessing the document. This will make several path issues obsolete that are somewhat problematic with `lilypond-book`.

LaTeX's will then use LilyPond to compile the score fragments whenever necessary and replace the source code with the resulting image files. This will be particularly useful for documents with large numbers of music examples, especially short ones.

[1] `http://ctan.org/pkg/lilyglyphs`
[2] `http://www.smufl.org`
[3] `http://www.smufl.org/fonts/`
[4] `http://www.openlilylib.org/musicexamples`

2.3 muscritreport

`muscritreport` will be a LaTeX package to typeset critical reports. It will benefit from the fact that – unlike with word processors or DTP programs – LaTeX commands can have arguments. This can be used to treat entries for a critical report as "records" of a "dataset". It will be structured similar to LaTeX's existing BibTeX bibliography tools.

The data for the critical report can thus be stored in a structured format that is completely independent from its appearance in the printed edition. In fact it's independent from its use altogether: using generic storage formats enables us to use the entries not only for the printed report but also to populate a database or to serve requests from a web frontend. At the same time the visual appearance for a printed edition or a "house style" is completely configurable.

One important aspect that will make `muscritreport` a unique tool is that the storage files for the critical report entries can be generated from annotations within a LilyPond score's input file (through `\annotate`, see 1.1). This way it will be possible to completely prepare a scholarly edition within one common code base!

2.4 riemann

`riemann` is the equivalent to the planned analysis symbols addition to LilyPond (see 1.3). The only difference is that it will be organized as a LaTeX package. The intention is to make it as versatile as possible, making use of LaTeX's argument capabilities and creating something similar or actually using LaTeX's famous mathematics mode.
Development of the LaTeX and the LilyPond part should probably be done together, in one project.

3 Core LilyPond Improvements

While LilyPond is a very good piece of software it is not perfect – but this doesn't come as a surprise, because hey, which software is?

The LilyPond developers are quite eager to improve the program, fix bugs and add improvements, their community is active but of course too small. However, I see a few areas of improvement from the perspective of scholarly musicology that aren't in the focus of the development team. But with some interest, manpower and money most of it could be realized on comparably short notice. So the following sections describe a few of these areas, discussing what has been done and what it might take to tackle them.

3.1 Data Exchange, Particularly MusicXML

Actually this section deals with the only issue I'd actually call a real deficiency with LilyPond: It can't export scores to other than graphical formats (PDF, PNG, SVG, EPS). Often this isn't a big deal because LilyPond's output is of such a high quality that there shouldn't be any need for post-processing. The only use case sometimes heard of is editing the final score with real graphics programs – which can open the PDF or SVG files.

But in fact the current state isn't good social behaviour – "vendor lock-in" is one of the most-dreaded issues in the Free Software world. It becomes a problem when most publishing houses, the major ones in particular, insist on their well-tested workflows and only accept files in Finale or Sibelius format to finally print their scores. Given all the unique features of plaintext based workflows it would be desirable to convince these publishers to also accept LilyPond/LaTeX files for their publications, but of course one can't expect this to happen in one go. But the advantages of plaintext and version control are so massive that it still makes sense to go through the complete preparation process of an edition using these tools and only convert the material to the more common commercial formats as a last step.

Therefore an option to export scores to the common interchange format MusicXML is one of the most important LilyPond desiderata from the scholarly perspective. There has been discussion about exporting the "raw" musical structure versus the complete (supposedly superiour) layout information. But for the intended purpose the latter isn't necessary at all, quite the contrary. The most pure representation of the musical content will interfere the least with the presets and standards of a post-processing engraver and his tools.

A solution will have to take several factors into account. First it has to capture the "music stream" containing all musical information. Then it will have to parse the input files themselves because they contain information that isn't part of the music stream. Finally it will have to merge all available information to a reliable MusicXML representation.

The reason for this is that a MusicXML file has a one-to-one relationship between musical content and graphical result, which isn't the case with LilyPond files. A LilyPond file may contain music definitions that are used multiple times, possible in transpositions and/or metrical translations. Additionally it can contain functions that influence the musical content – at its extreme it can consist of a Scheme function that algorithically generates the whole score. So exporting a file to MusicXML actually requires to be working inside a LilyPond compilation run – otherwise one would have to reinvent the whole LilyPond parser.

Currently there is some work going on in this direction, but it can't be said yet how much work, discussion and development is necessary to come to a working, versatile and reliable solution that ensures usable results with minimal need for manual post-processing. Frescobaldi has a MusicXML export function, but this is so far only parsing the input files themselves. It can re-export music imported from a MusicXML file and seems already in a somewhat working state with that, but it can't reliably export arbitrary scores.

So this is a project that should be part of a "research" project that is able to raise enough funds to finish it successfully.

Once this is working – or at least clearly heading towards a working solution – it should be considered to enhance the concept to also embrace the MEI[1] concept. Using LilyPond as part of a MEI toolchain for scholarly editing seems to be an attractive idea. I would also say that editing LilyPond files in Frescobaldi is much more comfortable than editing XML files manually, while using a text based approach is vastly superior to using graphical tools like e. g. a Sibelius plug-in[2] because of the advantages of version control.

3.2 Implementation of a draft mode

There have been very successful experiments with using a "draft mode" for compiling LilyPond scores. During the development of a score many aspects of it can be visualized, mostly by the use of colors but also by adding elements that don't change the layout. For example we colored editorial additions, items to be discussed or items that have been manually positioned. Graphics that are added to the score may be the control-points of slurs etc., the reference points of objects or verbal comments.

While this is already working quite well there currently is a project to incorporate this functionality into LilyPond itself to make it generally available. As a first step a significant enhancement has been added to Frescobaldi's Preview Mode[3] which is designed to be moved to LilyPond itself as soon as it has proven to be sufficiently stable. This isn't exactly specific to scholarly work, but it has proven to positively affect work on scholarly editions.

3.3 Support for PDF Layers

The ability to create multiple layers[4] in the output file and selectively place items on them would increase the impact of some of the other improvements, in particular `lilypond-doc`, the editorial

[1] `http://music-encoding.org`

[2] `http://www.sibeliusblog.com/news/mei-plug-in-released-for-sibelius/`

[3] `http://lilypondblog.org/2013/10/preview-mode-preview/`

[4] These are officially called Optional Content Groups, see `http://acrobatusers.com/tutorials/print/creating-and-using-layers-ocgs-with-acrobat-javascript`

toolbox and the draft mode. One wouldn't have to compile a score *with or without* features such as annotations or control-points highlighting but could generate a layered file and easily switch layers on and off in the PDF viewer.

In a first step one would only implement the basic infrastructure to be able to print items on different layers, without affecting the layout at all. This will enable applications like those described in the previous paragraph.

In a next (optional) stage one could then extend the concept and think of layers that *do* or *do not* affect layout decisions (e. g. collision handling). With this technique one could start to think of advanced edition techniques such as to provide alternative readings in one document. But while the basic outputting technique is nearly trivial this whole topic is very complex and would only be manageable in the context of a research project. What makes this interesting is that this research could break new grounds for thinking about cross media publishing with scholarly editions – for which plaintext file formats are a very suitable foundation anyway.

Another approach to this idea would be to attach "layer" information to the objects in SVG output. This would make it possible to develop viewers/editors that can selectively show or hide items and groups of items. Other than with PDF output this might open up viable options for thinking into the direction of browser based edition presentation.

4 Additions to Frescobaldi

Frescobaldi[1] is probably the most complete editing environment for LilyPond files today. Apart from its great interface and huge number of useful tools it is particularly interesting from the perspective of the current document. It is actively developed, uses a very usable, cross-platform application framework (PyQt) and is designed very extensibly. Recently there have been a number of contributors starting to work on it (myself included), and there are currently very promising improvements being made. In addition to this general environment it would be very easy to provide specific branched versions for an institution or a given project without having to irreversably fork one's own program from Frescobaldi.

While the following ideas aren't all specific to scholarly work I'll describe them anyway because they give a good impression of the current options and the direction of future development.

4.1 Graphical Editing Capabilities

While most LilyPond users are convinced of the virtues of the text based approach there *are* many things that can be done more easily in a graphical environment. One most notable area is the tweaking of Bézier curves – a task that is one of the most time-consuming when dealing with complex scores. Therefore it is a very important step that currently Frescobaldi is getting graphical editing capabilities. This will considerably shift our way to deal with LilyPond files and may make the whole thing more accessible for people not used to the text based approach.[2].

In a first step Frescobaldi (already) got an SVG based *Music View*. This was done because the objects in an SVG file can principally be edited graphically, which isn't possible in a PDF file. The fundamental idea is to relate graphical edits to textual tweaks of the input file. Thanks to two-way point-and-click we have a connection between a graphical object and the corresponding source code in the input file. If an object is edited graphically it is possible to determine its type and thus possible textual tweaks that can be applied. Depending on the type of object the editor should suggest different possible tweaks and insert appropriate values. Additionally there will be a semi-graphical editing panel that allows one to enter tweak values in a numerical way, the point being that the panel "knows" about possible types of tweaks that can be applied and takes care of inserting them correctly into the source code.

This will also be the user interface for editing annotations as described earlier in 1.1. And *if* a way would be developed to exchange LilyPond and MEI files this would also be the place to provide native editing interfaces for MEI features.

[1] `http://www.frescobaldi.org`

[2] For more information see the respective entry on Frescobaldi's issue tracker at `https://github.com/wbsoft/frescobaldi/issues/284`

4.2 Partial/automatic Compilation

One issue when working with tools that "compile" plaintext files is the time one has to wait for a successful compilation. This can't be dealt with purely by increasing processing power and more efficient algorithms because it's part of the fundamental concept. One strategy to deal with this is to find solutions to selectively compile only the relevant portions of the score that are currently being worked on. For example I have set up a project for the edition of a large orchestral score where any editor only works with a very small segment file (one voice over the range of one rehearsal mark). When actually working on that segment he will only compile this specific segment which will compile really fast, while the complete instrumental part or the score have to be compiled separately.

Frescobaldi development intends to experiment in this direction to offer partial and automatic compilation, and also (optional) automatic recompilation, for example triggered by graphical edits in the music view. Currently there is a basic implementation for an automatic recompilation which for small scores gives a user experience that is near the instant layout of graphical tools. But for scores that take a significant amount of time (i. e. anything beyond a few seconds) to compile this has to be combined with another approach of partial compilation. Of particular interest for our purpose would be options to compile parts of the score out of the layout context of the complete score. While this of course doesn't make any sense when working on the typographical perfection of a score it may significantly improve the working experience when dealing with the *content* side of an edition.

4.3 Support for Workflows with Version Control

Frescobaldi could be made aware of the Git project context a given score is in. That is, it could get a notion of different versions of a score, whether they are in different branches or at different points in history of a project. For example it would be possible to add information about the specific commit that a resulting PDF file belongs to and automatically open the source files in exactly the state they were in when producing a given score. This way it would be easy to compare different versions of a score side by side, e.g. the original and the proof-read or (more useful) the default with a beautified version.

Other Git features that could be incorporated are (for example): 1) Display the differences of a document against the last committed state and selectively stage or revert current changes. 2) *Git blame* to display who last modified a given fragment of the input file.

Additionally one could think about integrating work with remote repositories and develop the notion of a shared session in order to enable "live" collaboration on a score over the network. In order to be useful these ideas would have to be thoroughly discussed, but I think they could have an enormous impact on the way we think about collaboration in musicological work.

Currently I'm working on the foundation for such an enhancement by developing an infrastructure to manage *program versions* and automatical updates of Frescobaldi through Git. Right now it is possible to switch between different program versions that are present in Git *branches.* This makes it easier to inspect features that other contributors are currently developing. While "lacking" most of the versioning complexity of project management it is an opportunity to gain experience with the topic and will provide a solid foundation for future development.