USING MUSIC FEATURES FOR MANAGING REVISIONS AND VARIANTS IN MUSIC NOTATION SOFTWARE

Paul Grünbacher
Johannes Kepler University Linz
Software Systems Engineering
Linz, Austria
paul.gruenbacher@jku.at

Rudolf Hanl
Johannes Kepler University Linz
Software Systems Engineering
Linz, Austria
rudi@ruka.at

Lukas Linsbauer
Technische Universität Braunschweig
Software Eng. and Automotive Inf.
Braunschweig, Germany
l.linsbauer@tu-braunschweig.de

ABSTRACT

Music engravers need to manage both revisions and variants of digital music artifacts created with music notation software. However, existing version control systems such as Git fail to manage fine-grained revisions and variants in a uniform manner. This paper presents an approach that uses music features and applies a variation control system in the domain of music notation. In particular, we extended the variation control system ECCO to support the evolution of digital music artifacts encoded in LilyPond. We illustrate music features using a running example. We present basic feature-oriented workflows and discuss the architecture and implementation of our LilyECCO tool. We further present a preliminary evaluation based on an existing LilyPond music artifact.

1. INTRODUCTION

Music notation tools encode music as digital artifacts using languages such as MEI, MusicXML, LilyPond, or Humdrum to name but a few [1]. As with any digital artifact, music notation tools face significant challenges of managing changes. In particular, the continuous evolution leads to many *versions* of music artifacts. Two kinds of versions can be distinguished [2]: *revisions* are the result of evolution in time, e.g., adding some dynamics to a note, correcting a slur, or fixing the pitch of a note. Revisions thus denote sequential versions representing a snapshot of the evolution of a music artifact. *Variants* on the other hand stem from evolution in space, e.g., adding lyrics in an additional language or adding a voice for an instrument needed for a new edition of a piece. Variants thus denote versions of music artifacts that need to exist concurrently.

The issue of managing revisions and variants becomes essential especially if a team of music engravers collaboratively defines and evolves digital music artifacts. Currently, engravers use general purpose version control systems such as Git to manage revisions and variants. However, while existing version control systems are powerful for handling revisions of digital artifacts, they have deficiencies with respect to managing variants. In particular, the available branching and forking mechanisms of Git and similar tools conceptually create clones. This duplication of artifacts considerably increases the maintenance effort as changes will need to be propagated manually to all clones.

Managing the evolution of music artifacts has many similarities with managing the evolution of artifacts in other domains. For instance, a number of *variation* control systems [3] have been conceived in the field of software engineering. Variation control systems provide capabilities for uniformly handling both revisions and variants based on decomposing digital artifacts into finer-grained variable entities called *features* [4]. In software engineering, for example, features are used to describe user-visible characteristics of software systems (e.g., specific functions). Features are then used to manage different revisions and variants of software systems. Properly decomposing a software system into features is seen as essential for the immediate and long-term success of software systems [4] and case studies demonstrate the usefulness and feasibility of feature-oriented version control for large-scale systems [5].

This paper investigates the suitability of applying features and variation control systems in the domain of music notation. We demonstrate the basic idea of features in music artifacts using a running example. We then show how music features can be used to manage revisions and variants of music artifacts. In particular, we extended the variation control system ECCO [6, 7] to support revisions and variants of music artifacts encoded in LilyPond [8]. We present the feature-oriented workflows and briefly discuss the architecture and implementation of our LilyECCO tool. We further present a preliminary evaluation based on a complex music artifact.

2. MUSIC FEATURES

The question of what constitutes a feature depends on the application context and the domain of interest. For instance, a common definition from software engineering describes a feature as "a distinguishable characteristic of a concept (system, component, etc.) that is relevant to some stakeholder of the concept" [9]. Applying this rather general definition to music notation means to find distinguishable characteristics of music relevant to a music engraver. Figure 1 shows our running example demonstrating possible examples of music features. When studying the first few measures of this vocal score by Claude Debussy we

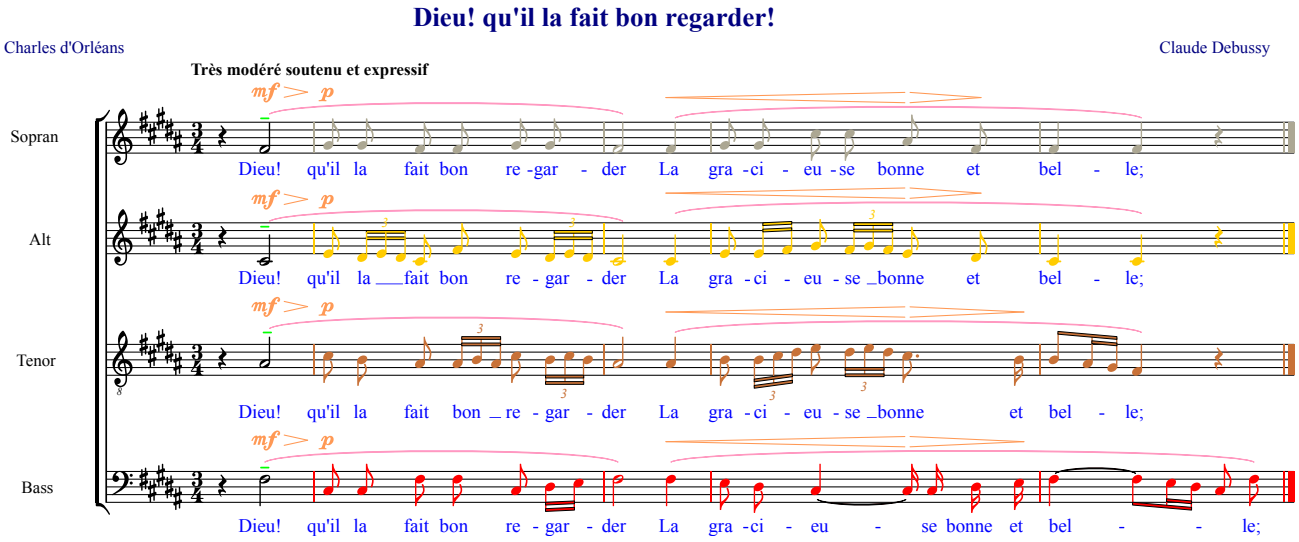


Figure 1. Ten music features in the piece *Dieu! qu'il la fait bon regarder!* (by composer Claude Debussy and poet Charles d'Orléans) – setup (black), bass notes (red), tenor notes (brown), alto notes (yellow), soprano notes (grey), dynamics (orange), slurs (pink), articulations (green), lyrics (light blue), and piece header (dark blue).

can find features for setting up the score (setup), for defining notes (soprano notes, alto notes, tenor notes, bass notes), for handling texts (lyrics, piece header), as well as features concerning articulations, dynamics, and slurs. In Figure 1 colors are used to visually indicate the mapping of elements in the score to features. The running example shows that a music artifact can be composed based on music features. The figure shows one particular version of the music artifact. However, different revisions and variants can be created by including or excluding different (revisions of) music features.

We now use this example to define and illustrate important concepts of music features based on existing work on feature-oriented software engineering [9, 3]:

Mandatory and optional music features. Common features that are present in all variants of a music artifact are referred to as mandatory features. For instance, we could assume that in our example this is the case for the feature soprano notes. Optional features on the other hand exist only in some variants, e.g., an English translation of the lyrics may not be provided in all its editions.

Revisions of music features. Revisions of features denote sequential versions and are the result of evolution in time, e.g., fixing the pitch of a note. For instance, the revision soprano notes.2 may fix a false note accidentally introduced by the engraver in the initial revision soprano notes.1.

Alternatives of music features. In many practical cases different alternatives of features are needed. For instance, one could imagine different editions of our running example created based on different translations of the lyrics.

Composing music features. Mandatory music features appear in all variants of a music artifact, while optional and alternative features result in different variants of a music artifact, which depend on the selection of the desired features by an engraver.

3. INTENSIONAL AND EXTENSIONAL VERSIONING

We investigate if and how music features can be used for managing revisions and variants in music notation. The field of software configuration management has developed a wide range of methods and tools, which pursue two important versioning strategies [2]:

Extensional versioning assumes that all existing versions are explicitly enumerated. It then allows to retrieve all versions that have been created before. Examples of tools supporting extensional versioning are Git or Subversion, which keep track of the evolution history by assigning revisions to states of a system over time. However, in practice, evolution is rarely just a linear sequence of steps, and current tools thus provide branching mechanisms for dealing with variants: for instance, short-term feature branches exist for as long as it takes to develop a new feature in isolation. Once the new feature is finished, the branch is no longer used and merged with the original artifact. However, at this point the new feature becomes inseparable from the rest of the artifact, i.e., its location in the artifact is not managed explicitly. The purpose of long-term branches on the other hand is to create clones of existing artifacts, based on which variants are then created. However, long-term branches quickly leads to maintenance problems as updates and fixes need to be propagated to all variants.

Intensional versioning aims at overcoming these limitations by providing fine-grained mechanisms for managing variants, thereby avoiding feature branches or variant branches. Furthermore, this strategy allows to create versions that have not been explicitly enumerated and committed before. Examples of tools supporting intensional versioning are variation control systems such as ECCO or SuperMod [3]. They use concepts like features, configurations, and construction rules to compose arbitrary versions.

We will show in this paper that such composition is pos-

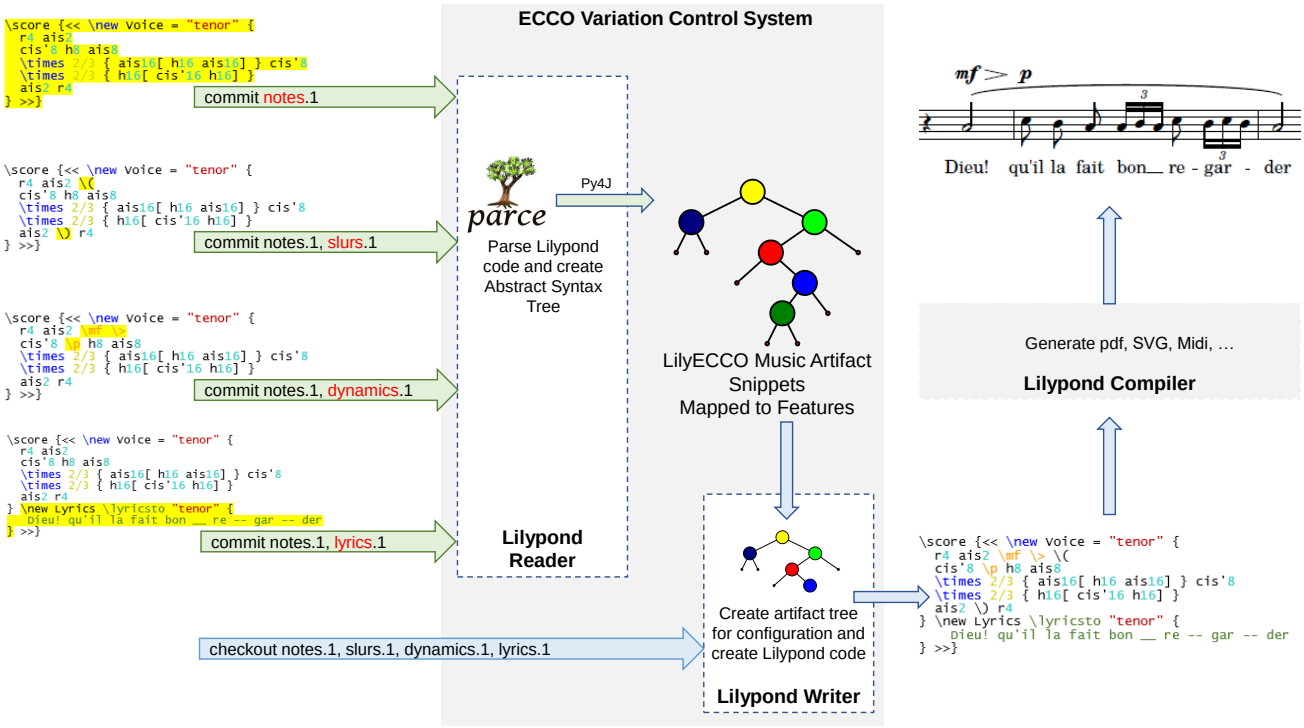


Figure 2. Workflow and architecture of LilyECCO.

sible for music features, however, there are two key challenges: (i) automatically composing features relies on *creating precise and fine-grained mappings of features to elements* of music artifacts (cf. Figure 1). Creating such mappings manually quickly becomes infeasible, thus making tools indispensible; (ii) features are not independent of each other and so *handling interactions between features* [10] becomes essential. As a simple example consider the last two bars of the bass voice in the running example. The alignment of the lyrics works if both the tie and the lyrics are present, while problems of misaligning the text would occur in LilyPond if excluding the (fictitious) feature ties.

4. LILYECCO: MANAGING FEATURE-ORIENTED REVISIONS AND VARIANTS OF MUSIC

Our LilyECCO approach for feature-oriented version control of music uses LilyPond [8], a computer program and domain-specific language for music engraving. The native text-based input language for LilyPond is comprehensive and provides commands needed for engraving classical music, complex notation, early music, modern music, tablature, vocal music, or lead sheets. LilyPond automatically computes the details of music layout, thereby allowing composers, transcribers and publishers to focus on the music instead of tweaking the layout.

To illustrate the typical feature-oriented workflow of LilyECCO we use our running example to illustrate its evolution in different evolution steps, ultimately resulting in different revisions and variants. We demonstrate how a music engraver can incrementally add music features to a repository and later compose variants based on the repository by specifying a feature configuration. We also illustrate how LilyECCO internally manages snippets of music artifacts in a feature-oriented manner.

Adding music features to an ECCO repository is possible by executing the operation commit, thereby telling LilyECCO which features (and feature revisions) are contained in a change. The left part of Figure 2 illlustrates how a music engraver would commit features to the ECCO repository incrementally to create the running example. Due to space restrictions only partial code can be shown. The engraver first commits the notes, and then adds slurs, dynamics, and lyrics. After each step, the engraver commits changes by defining combinations of (revisions of) features, as shown by the commit commands on the green arrows. As soon as the engraver commits a change, the newly added or changed LilyPond source code must be mapped to the correct features. In LilyECCO, this is done using automated code analysis to extract the newly added source code artifacts and then map them to the newly created feature(s). For instance, when committing notes.1 and slurs.1, LilyECCO determines that the code marked as yellow (the newly added phrasing slur) needs to be mapped to the new feature slurs, while the unchanged code is still mapped to the feature notes.

Creating a music variant based on an ECCO repository is supported by executing the operation checkout, which composes (combinations of) features already stored in the repository. A music engraver can create music variants at any time. For example, in our running example shown in Figure 2 the engraver checks out a music variant based on a configuration expression defining the first revision of the features notes, slurs, dynamics, and lyrics as indicated by

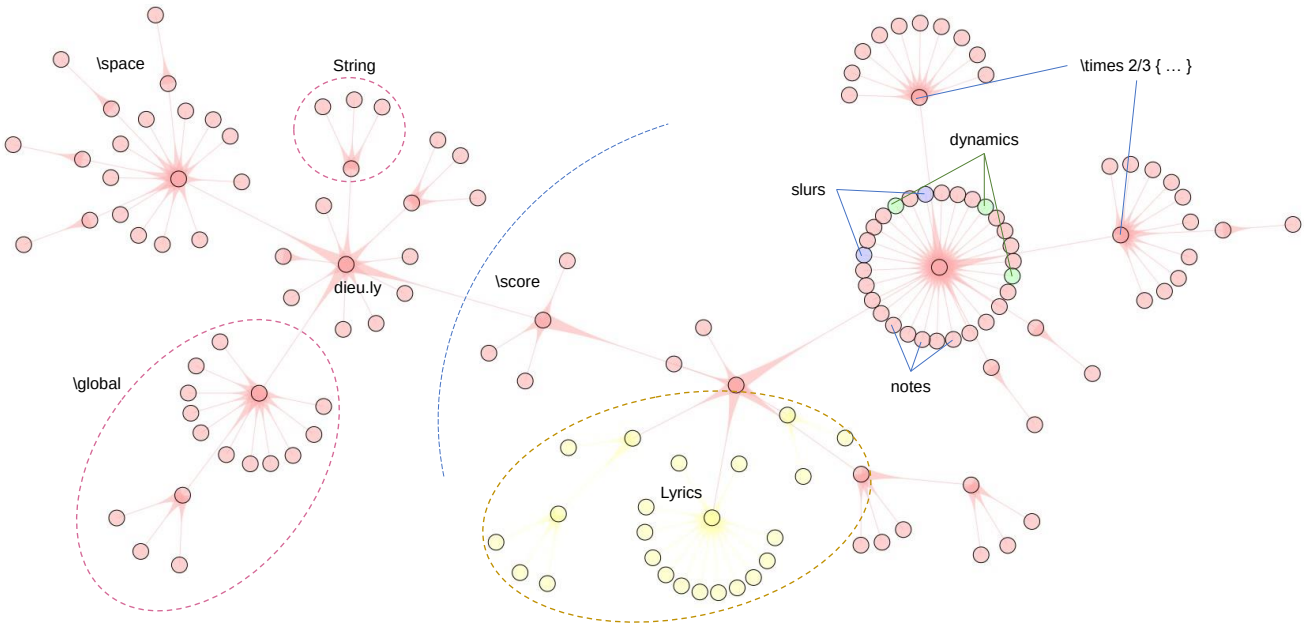


Figure 3. Selected music artifact snippets for the running example. Each node refers to an artifact snippet while colors represent features. Labels denote individual artifacts (e.g., \global, \times 2/3) as well as features like slurs or dynamics. The edges are directed and represent containment, thus resulting in a tree structure. Dashed lines are used to indicate artifacts subtrees (e.g., Lyrics).

the blue arrow. LilyECCO then automatically composes the code shown on the right. The workflow continues and the engraver can modify the generated code to create a new revision or variant, and again commit the changes to LilyECCO if desired.

Throughout the continuous evolution of a music artifact, the presented workflow is used to add new features or to extend and update existing ones. Conducting this feature-oriented workflow is infeasible without tool support for any non-trivial music artifact. For instance, feature mappings need to be continuously updated and feature interactions need to be managed. LilyECCO supports such a feature-oriented process.

5. LILYECCO ARCHITECTURE

LilyECCO's architecture shown in Figure 2 comprises the following key components:

Variation Control System. It has been shown that variation control systems can address both challenges of creating precise mappings and managing feature interactions [3]. We selected the variation control system ECCO for our purpose [6, 7]. It stores music features in a repository in the form of artifact graphs and maintains mappings between features and snippets of the music artifacts. ECCO further supports the evolution of features over time by considering feature revisions in the automatically computed traces.

For the purpose of illustration Figure 3 shows a simplified and partial view of the music artifact snippets of Debussy's piece after committing the features shown in Figure 1. This internal representation allows to determine how specific snippets of LilyPond code map to specific features. In particular, presence conditions determine whether the artifact snippets are part of a specific version. A presence condition is a propositional logic formula with feature revisions as literals [7].

We extended ECCO to allow storing and managing music features in an artifact graph structure shown in Figure 3. For that purpose, we developed a LilyPond Reader plugin allowing to recognize and store music artifacts, and a LilyPond Writer plugin to automatically compose LilyPond code for a chosen configuration:

LilyPond Reader. More technically, the Python package parce [11] parses LilyPond input into a tree structure based on the LilyPond language definition. This tree structure is then used by our LilyPond Reader plugin to analyze LilyPond input files and to create an artifact graph managed by ECCO. The ECCO system is implemented in the programming Java, while the parce parser is written in Python. We thus use Py4J (https://www.py4j.org) to execute parce (version 0.13) from our LilyPond Reader plugin written in Java.

LilyPond Writer. The LilyPond Writer plugin for ECCO is capable of creating a music artifact tree for a specific configuration. This is done by checking out a combination of (revisions of) features. LilyPond code is automatically created for the features selected by the music engraver in a configuration expression.

LilyPond Compiler. Finally, as shown in Figure 2, the LilyPond compiler processes the LilyPond code generated by the LilyPond Writer to produce output such as PDF documents, SVG files, or MIDI files.

ECCO is available as an open source system at https://github.com/jku-isse/ecco. We also plan to release our LilyECCO extensions in the future.

6. EVALUATION

Our evaluation of the LilyECCO approach was guided by two research questions:

RQ1 – Correctness. Are the mappings of features to music artifacts computed correctly? We checked the correctness of the approach by creating and replaying the evolution history of a music artifact and then automatically composing different variants.

RQ2 – Performance. Does the approach scale for real-world music artifacts? We measured the performance of executing our approach to assess its usefulness in realistic workflows.

6.1 Data Set

For our evaluation we applied the approach to a fairly complex piece of vocal music. In particular, we chose the motet "Factus est Repente" by the Upper Austrian composer Balduin Sulzer as our data set. The piece was decidated to the vocal ensemble the first author is part of. The piece consists of two parts: the first part comprises six voices, while the second part comprises two voices. The complete piece comprises about 500 lines of code. Analyzing this piece with the parce parser used in our LilyECCO plugin results in 10,286 abstract syntax tree elements, with a maximum tree depth of six. Specifically, the most frequently occurring elements in this piece are 2,009 note pitches, 1,456 parts of lyrics, 1,170 durations of notes, 949 lyric hyphens, 688 built-in LilyPond commands (e.g., `\relative`), 632 brackets, 583 direction delimiters, 583 script literals, 354 beams, 265 numbers (e.g. 4/4), 246 rests, 188 slurs and 183 definitions of dotted notes.

We identified music features by considering both the structure of this piece (i.e., parts and voices) as well as the basic elements of musical scores (i.e., dynamics, lyrics, articulations, texts, etc.). Upon closer inspection, we identified 52 music features, which define note pitches and durations for the six voices of part 1 and the two voices of part 2 (e.g., partoneSopOneNotes, partoneBasOneNotes, parttwoSopTwoNotes), lyrics (e.g., partoneSopOneLyrics, parttwoSopTwoLyrics), score setup (header, scorePartOne, scorePartTwo, text), as well as articulations, dynamics, slurs, and beams for each voice for our evaluation.

In our experiment we regarded the complete piece as the final version of our revision history. Using the Frescobaldi editor (www.frescobaldi.org), we then manually removed one feature after the other from the LilyPond code, giving us an evolution history of 52 versions. Replaying this history reflects possible changes of a music engraver incrementally adding features to define the music artifact.

6.2 Research Method

Regarding the correctness (RQ1) of the feature-to-music mappings we performed three steps:

(i) We developed a script allowing us to automatically replay each evolution step of the data set. The script commits each version of the evolution history, thereby incrementally adding all features described above.

(ii) We checked out selected revisions and variants based on the music features defined in our data set using LilyECCO. Since LilyECCO can also produce LilyPond files that were never input by the engraver we selected examples of both *extensional and intensional versioning*: extensional versioning means to construct previously committed versions, while intensional versioning means to construct new versions based on feature combinations never committed before (cf. Section 3).

(iii) We manually inspected selected computed feature-to-music mappings in the ECCO repository. We also checked the integrity of the different variants by compiling the resulting LilyPond code. In case of syntax errors, we analyzed the reasons preventing successful compilation. Besides checking for syntax errors we also visually checked the variants in the resulting scores.

Regarding performance (RQ2) we measured the time required to analyze the different versions of the evolution history. Since the complexity of music artifacts grows over time, it is important to assess if later commits still scale, i.e., if the performance of the ECCO algorithms used to compute commonalities and differences between two successive commits allows to use LilyECCO in realistic workflows. We measured the time needed to parse the LilyPond code, and the time needed to commit the new version to the ECCO repository, which also includes the time needed to compute the commonalities and differences of different versions.

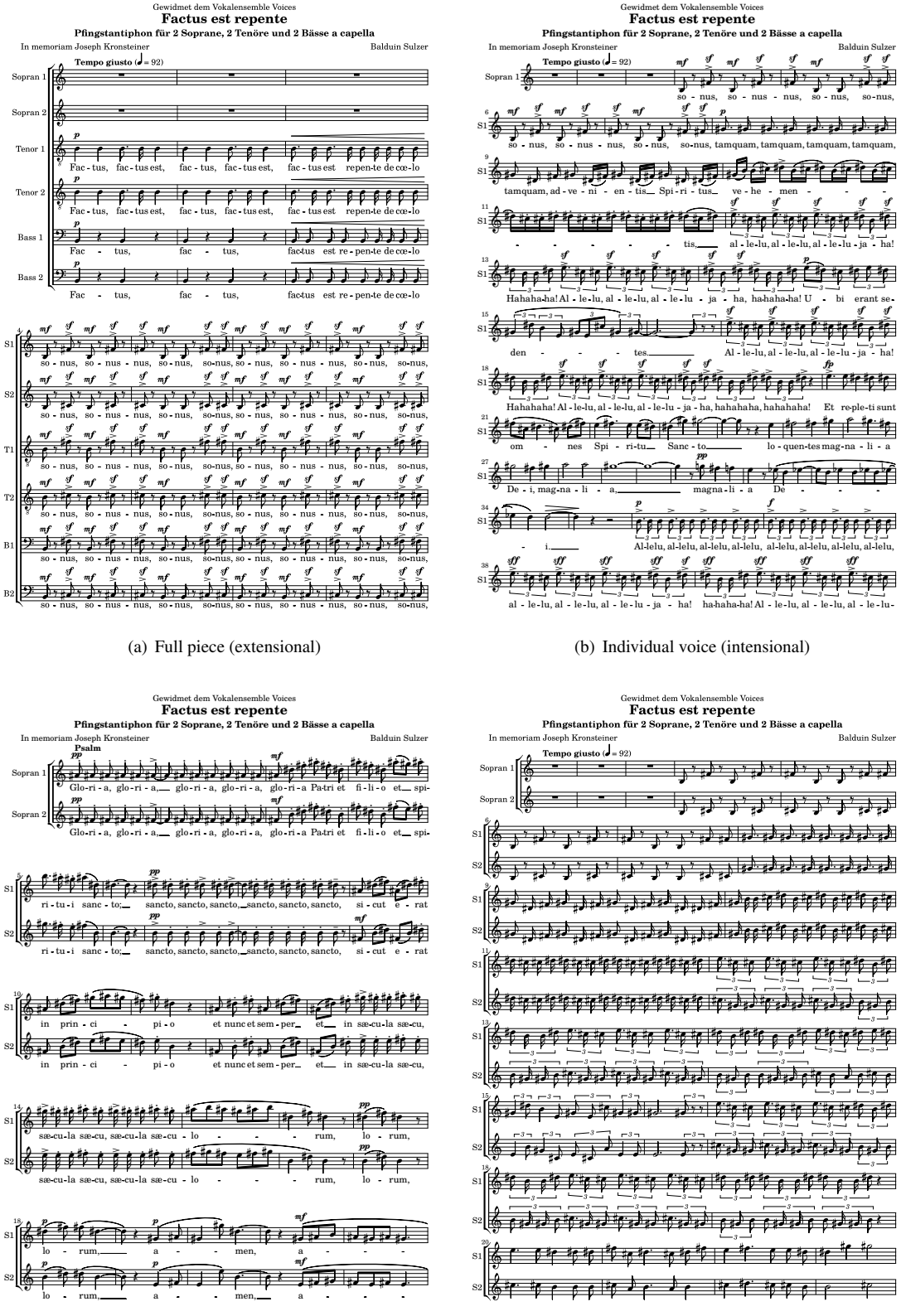
6.3 Results RQ1: Correctness

As outlined above, LilyECCO uses a tree structure to create artifact snippets, which are created and mapped to features based on the feature information in the commit command. In particular, the LilyECCO Reader distinguishes between different contexts (default, string, comment) resulting in tokens at different depths (default, brackets, keywords, lyrics, numbers, pitches and delimiters). More specifically, ECCO generated 11,786 artifacts snippets when replaying the evolution history for the data set we used in our evaluation. About half of the snippets (5,322) depend on the top six mappings to features: partoneSopTwoNotes (1,001), partoneSopOneNotes (992), partoneTenTwoNotes (869), partoneTenOneNotes (839), partoneBasOneNotes (817) and partoneBasTwoNotes (804).

Regarding *RQ1 (correctness)* we show the first page of the score for four variants created based on the Sulzer data set. Figure 4 depicts four different variants:

(a) Full piece. This variant was created by checking out all music features listed in Section 5.1. The generated code compiles correctly and the score meets the expectations. Creating this variant is an example of extensional versioning [2], i.e., retrieving a previously constructed version from the repository, in the case of LilyECCO based on explicit feature names. This demonstrates that the approach can successfully compose music artifacts based on artifacts snippets stored in the repository for the case of extensional versioning.

(b) Individual voice. This variant represents an interesting case of intensional versioning [2], i.e., to automatically



(a) Full piece (extensional)

(b) Individual voice (intensional)

(c) Soprano voices Part 2 (intensional, small fixes needed)

(d) Soprano voices, notes only (intensional)

Figure 4. The first page of the score for four variants created with extensional or intensional versioning based on our data set during the evaluation: (a) shows the score with all music features; (b) shows a score variant for a single voice; the score variant in (c) includes the two soprano voices of Part two; and variant (d) includes only the notes of the soprano voices. In cases (a), (b), and (d) the variant was created without errors while minor fixes were needed in case (c) to move and remove wrongly mapped code (cf. Section 6.3.).

construct a new combination of features on demand. The intention was to create a score variant with all features needed for a single voice. As in the first case the code of the generated variant compiles correctly and the score meets the expectations.

(c) Soprano voices of Part 2. This score variant extracting the soprano voices for part two is similar to the previous one at first sight. However, in this case of intensional versioning the initial attempt resulted in a syntax error detected by the compiler. The reason is that LilyECCO had never analyzed this specific variant, and there is an interaction of features that never existed before together (cf. our example at the end of Section 3). In particular, LilyECCO could not distinguish some global definitions and score definitions of part 2, as these features had never been committed separately. However, the workflow in LilyECCO in such cases is to simply fix the syntax errors in the checked-out variant and then commit the corrected variant, thereby telling LilyECCO how the specific variant looks like. In this case this was done by moving and removing two lines of code and commit the variant again, thereby allowing LilyECCO to distinguish these two features in future checkouts. Over time, this iterative process improves the mappings of features to music elements and allows the music engravers to correctly perform intensional versioning.

(d) Soprano voices, notes only. This variant is an example of successful intensional versioning. It includes the two soprano voices, but drops all other features. In this case LilyECCO was able to successfully construct the previously unknown variant.

6.4 Results RQ2: Performance

Regarding performance we report the execution times of analyzing the different versions of the evolution history. The experiment has been conducted using a Java 13 HotSpot 64-Bit Server VM on Windows 10 running on a PC with an Intel Core i5 with 3.5 GHz and 16GB DDR3-RAM. In our data set the size of the music artifact ranges from 39 elements (AST nodes) in the first version to 10,286 elements in the 52nd version. The algorithms in ECCO rely on computing tree-based commonalities and differences of code, so both the size of the artifact and the interaction of features have an influence on performance.

However, performance was acceptable for the data set given the complexity of the score, the number of features, and the size of the evolution history. The *time to commit* ranged from 0.1 to 28.2 seconds (mean: 5.8; standard deviation 7.8). User-perceived performance also depends on the time needed to parse the LilyPond code and to create a tree structure that can be handled by ECCO. This *time to parse* ranged from 0.2 to 2.1 seconds (mean: 1.3; standard deviation: 0.6). With respect to overall performance, the maximum time needed for parsing and committing a version was about 32 seconds for version 52, which indicates sufficient performance for practical workflows, given that commits would normally not be made very frequently.

6.5 Discussion

The preliminary evaluation was successful in that it demonstrates the feasibility of the approach to automatically map snippets of music artifacts to features and to compose new variants of music artifacts based on the features for both extensional and intensional cases. These promising results give rise to plenty of opportunities for further research:

The question of *what constitutes a feature* depends on the application context and the eye of the beholder. For instance, features may be used in an ad-hoc fashion to track increments and additions to music artifacts (e.g., adding a new voice). However, they may also be used in a more systematic manner by planning the purpose of the different required variants in advance. This means that features used for the purpose of creating variants for music education would differ from the scenario of a music publishing house creating different scores from the same base.

Evaluating the usefulness of music features will be necessary, e.g., by conducting user studies with music engravers or by analysing existing evolution histories such as the Mutopia archives. Such studies will also help to better understand the practical advantages of LilyECCO compared to a more traditional approach using Git or a similar tool. In our evaluation we chose a rather fine-grained definition of features for the purpose of evaluating both the correctness and performance of our approach. We plan to study the impact of different levels of feature granularity on both correctness and performance.

Another important area is to look at *usability*, in particular the cognitive complexity of specifying configuration expressions. Variation control systems like ECCO use logical expressions to manage variants with features. Depending on the number of versions and the interactions of features this task may become cognitively demanding. The Cognitive Dimensions of Notations framework [12] refers to such tasks as hard mental operations. For instance, creating configuration expressions for checking out variants is difficult for engravers who think in terms of music code but not in terms of features. It may help to let users point to artifact snippets that should be included in the variant rather than having them to consider logical expressions. Also, feature models may help to reduce the cognitive load by providing a higher-level and hierarchically-organized graphical perspective, as SuperMod [13] or FORCE^2 [5, 14] show.

In terms of possible tool support an interesting capability is to *color features* in music score editors, as shown for source code of programming languages [15]. Such a feature would also ease to systematically study the granularity of music features in realistic workflows. For instance, such studies have been conducted in the domain of software engineering to better understand how complex software artifacts can be decomposed into features of different granularity [15].

Our current LilyPond Reader and LilyPond Writer plugins have so far been primarily used and tested for vocal music. Further *implementation enhancements* are required to support the full scope of the LilyPond language. This will then allow a more comprehensive evaluation of using

music features for different kinds of music. We also plan to improve the performance of our approach by executing the Java and Python code of LilyECCO in a single virtual machine.

The LilyECCO extension for ECCO is based on the LilyPond system. However, our approach can also be applied to *other music notation software packages*, if they allow parsing their music artifacts to create a tree structure, which can then be analyzed by ECCO. ECCO also supports XML files. In case no API is available, an alternative thus would be to use MusicXML as an intermediate artifact format. However, this approach seems to be risky and cumbersome, given the often unpredictable results of current MusicXML exporters and importers.

Our current evaluation focused on the case of a single engraver committing music features and composing different music variants. We can extend this to support *collaborative scenarios involving multiple engravers* based on the distributed operations of the variation control system ECCO, which allows to clone repositories and to pull music features from one repository to another as shown in [16].

7. RELATED WORK

LilyECCO uses tree-based code diffing to relate music features to music elements when committing changes to the repository. Antila et al. [17] discuss the limitations of line-based diffing approaches and also propose a hierarchical diffing approach for collaboratively editing music artifacts. Similarly, Herold et al. [18] present the MusicDiff tool for comparing two files with encoded music scores, which can also visualize the differences between these encodings. However, both approaches do not use features to label changes and to support music composition as in LilyECCO.

Fournier-S'niehotta et al. [19] propose an approach that leverages a music content model for defining virtual corpora of music notation objects that allow the development of search and analysis functions across music artifacts encoded in different formats. While the approach does not consider evolution, the idea to perform analyses across diverse digital artifacts is also fundamental to LilyECCO, which can work with different kinds of artifacts.

Dannenberg has proposed to provide views on a score [20], which "contains a subset of the information in the data structure and sometimes provides alternate or additional data to that in the data structure". This would allow that a change in a score can automatically be propagated to the parts (views on the score). The composition of a variant with LilyECCO based on features can also be seen as mechanism to create views on a score, and changes committed to the shared repository could be made available to other views (variants) via committing feature revisions and again checking out variants.

LilyECCO composes snippets, i.e., partial scores, to create new scores based on a selection of features. The idea to compose new scores based on existing ones has also been proposed by Lepetit-Aimon et al. [21]. In their approach a score can be composed as an arbitrary graph of score expressions.

8. CONCLUSIONS

We presented the LilyECCO approach for managing both revisions and variants of digital music artifacts created with music notation software. Our approach uses music features and a variation control system to manage the evolution of digital music artifacts using the LilyPond language. Our preliminary evaluation investigated both the correctness and the performance of LilyECCO, overall demonstrating its feasibility. The experiences gained in the experiment further allowed us to identify a number of interesting research directions for using music features in music notation workflows.

9. REFERENCES

[1] W. Lemberg, L. F. Moser, and U. Liska, "Music Engraving Conference, Music University Mozarteum, Salzburg," 2020. [Online]. Available: https://gitlab.com/MusicEngravingConference/2020

[2] R. Conradi and B. Westfechtel, "Version models for software configuration management," *ACM Computing Surveys*, vol. 30, no. 2, pp. 232–282, 1998.

[3] L. Linsbauer, F. Schwägerl, T. Berger, and P. Grünbacher, "Concepts of variation control systems," *Journal of Systems and Software*, vol. 171, p. 110796, 2021.

[4] T. Berger, D. Lettner, J. Rubin, P. Grünbacher, A. Silva, M. Becker, M. Chechik, and K. Czarnecki, "What is a feature? a qualitative study of features in industrial software product lines," in *Proceedings 19th International Software Product Line Conference*, ser. SPLC'15. Nashville, USA: ACM, 2015, pp. 16–25.

[5] D. Hinterreiter, L. Linsbauer, K. Feichtinger, H. Prähofer, and P. Grünbacher, "Supporting feature-oriented evolution in industrial automation product lines," *Concurrent Engineering*, vol. 28, no. 4, pp. 265–279, 2020.

[6] S. Fischer, L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Enhancing clone-and-own with systematic reuse for developing software variants," in *30th IEEE International Conference on Software Maintenance and Evolution, Victoria, BC, Canada, September 29 - October 3, 2014*. IEEE Computer Society, 2014, pp. 391–400.

[7] L. Linsbauer, R. E. Lopez-Herrejon, and A. Egyed, "Variability extraction and modeling for product variants," *Software & Systems Modeling*, vol. 16, pp. 1179–1199, Jan 2017.

[8] "Lilypond – notation reference," 2020. [Online]. Available: https://lilypond.org/doc/v2.20/Documentation/notation-big-page.html

[9] K. Czarnecki and U. Eisenecker, *Generative Programming: Methods, Tools, and Applications*. Boston, MA: Addison-Wesley, 2000.

[10] P. Zave, "Feature interactions and formal specifications in telecommunications," *Computer*, vol. 26, no. 8, pp. 20–28, Aug 1993.

[11] W. Berendsen, "The parce module," 2020. [Online]. Available: https://parce.info

[12] A. Blackwell and T. Green, "Notational systems–the cognitive dimensions of notations framework," in *HCI Models, Theories, and Frameworks*, ser. Interactive Technologies, J. M. Carroll, Ed. San Francisco: Morgan Kaufmann, 2003, pp. 103–133.

[13] F. Schwägerl and B. Westfechtel, "Integrated revision and variation control for evolving model-driven software product lines," *Software & Systems Modeling*, vol. 18, no. 6, pp. 3373–3420, 2019.

[14] K. Feichtinger, D. Hinterreiter, L. Linsbauer, H. Prähofer, and P. Grünbacher, "Guiding feature model evolution by lifting code-level dependencies," *Journal of Computer Languages*, p. 101034, 2021.

[15] C. Kästner, S. Apel, and M. Kuhlemann, "Granularity in software product lines," in *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008, pp. 311–320.

[16] D. Hinterreiter, L. Linsbauer, F. Reisinger, H. Prähofer, P. Grünbacher, and A. Egyed, "Feature-oriented evolution of automation software systems in industrial software ecosystems," in *23rd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2018)*, Torino, Italy, 2018, pp. 107–114.

[17] C. Antila, J. Treviño, and G. Weaver, "A hierarchic diff algorithm for collaborative music document editing," in *Proceedings of the International Conference on Technologies for Music Notation and Representation – TENOR'17*, H. L. Palma, M. Solomon, E. Tucci, and C. Lage, Eds. A Coruña, Spain: Universidade da Coruña, 2017, pp. 167–170.

[18] K. Herold, J. Kepper, R. Mo, and A. Seipelt, "MusicDiff – A Diff Tool for MEI," in *Music Encoding Conference Proceedings*, E. De Luca and J. Flanders, Eds. Humanities Commons, 2020, pp. 59–66.

[19] R. Fournier-S'niehotta, P. Rigaux, and N. Travers, "Is there a data model in music notation?" in *Proceedings of the International Conference on Technologies for Music Notation and Representation – TENOR'16*, R. Hoadley, C. Nash, and D. Fober, Eds. Cambridge, UK: Anglia Ruskin University, 2016, pp. 85–91.

[20] R. B. Dannenberg, "Music representation issues, techniques, and systems," *Computer Music Journal*, vol. 17, no. 3, pp. 20–30, 1993.

[21] G. Lepetit-Aimon, D. Fober, Y. Orlarey, and S. Letz, "Inscore expressions to compose symbolic scores," in *Proceedings of the International Conference on Technologies for Music Notation and Representation – TENOR'16*, R. Hoadley, C. Nash, and D. Fober, Eds. Cambridge, UK: Anglia Ruskin University, 2016, pp. 137–143.