Paul Grünbacher, Rudolf Hanl, and Lukas Linsbauer
Institute of Software Systems Engineering
Johannes Kepler University Linz
Altenberger Str. 69
4040 Linz, Austria
paul.gruenbacher@jku.at, rudi@ruka.at,
lukas.linsbauer@gmail.com

Using Music Features for Managing Revisions and Variants of Musical Scores

Abstract: Music engravers nowadays use music notation software to create scores for musical works. As is common in any creative process, many different versions of digital artifacts are created—e.g., to manage editions of the same musical work, editorial markups, or the history and genesis of compositions. In the field of software engineering, researchers have proposed the use of features, i.e., user-visible aspects of systems, to manage both revisions and variants of source code and other software artifacts. Feature-based version control systems establish and maintain feature-to-code mappings defining which parts of the artifacts realize particular features, e.g., which code implements a specific function. These mappings can then be used to generate new variants based on the artifacts by selecting the desired features. Our article provides an in-depth study on feature-based version control for music notation. Our automated approach adopts the domain-specific language (DSL) LilyPond and the feature-oriented version control system ECCO. Existing studies show that features in musical scores are often fine-grained and affect only small parts of an artifact, are scattered across noncontiguous locations in the artifact, and highly interact with each other. Such properties have strong implications for the usefulness of versioning tools. Our experiment investigates two factors related to the correctness of output from feature-based version control systems when used for symbolic music notation. We demonstrate the incremental refinement of feature-to-artifact mappings when committing DSL code. We further study the impact of the order of feature interactions on the correctness of the automatically generated music artifacts. We find that a larger feature interaction threshold produces only marginally more correct results, but fixing and recommitting incorrect variants has a more powerful effect. Our results further show that considering DSL specifics is important for versioning fine-grained and scattered features.

Music notation is used to visually represent music to be played with instruments or sung by the human voice, i.e., to create "visual analogues of musical sound, either as a record of sound heard or imagined, or as a set of visual instructions for performers" (Bent et al. 2001). Music publishing was carried out since the late 16th century by engraving a mirror image of the music onto a metal plate, applying ink to the grooves, and transferring the music print onto paper. Music engravers used domain-specific tools such as scorers for staves and bar lines, elliptical gravers for crescendos and diminuendos, flat gravers for ties and ledger lines, punches for note heads, clefs, accidentals, and letters, etc. Music engraving nowadays relies on music notation software, which encodes music as digital artifacts using languages such as Cadenza, Music Encoding Initiative (MEI), MusicXML, LilyPond, or Humdrum (Field-Richards 1993; Lemberg, Moser, and Liska 2020). Such domain-specific languages (DSLs) reduce the gap between high-level concepts used by domain experts and low-level abstractions used by software developers (van Deursen, Klint, and Visser 2000; Mernik, Heering, and Sloane 2005; Kosar, Bohra, and Mernik 2016; Borum, Niss, and Sestoft 2021).

As with other digital artifacts, DSL code defining music can evolve (1) in time, leading to enhancements of the code over successive *revisions*, and (2) in space, leading to adaptations of the code to concurrently support different *variants*. This results in challenges for version control and variability management. Specifically, the domain of music notation faces challenges such as differences between variants of the same musical work, variants and revisions at different levels of granularity, different score layouts for the same work, and editorial markups and interventions, as well as understanding and managing the history and genesis of compositions (Teich Geertinger 2021). However, issues of version control obviously exist in many domains. In the field of software engineering, for instance, the term *feature* has been coined to refer to a "prominent or distinctive user-visible aspect, quality, or characteristic of a software system or system" (Kang et al. 1990; Czarnecki et al. 2012; Berger et al. 2015). In software systems, but also in software-intensive

Computer Music Journal, 47:3, pp. 50–68, Fall 2023
doi:10.1162/COMJ_a_00691

systems with software playing a major role, feature-oriented techniques have been proposed to manually or automatically relate features to the artifacts realizing them (Apel et al. 2013a). These methods use features for defining revisions and variants, and they use feature-to-artifact mappings to generate new variants of digital artifacts based on a configuration, i.e., a selection of the available features. However, it has been shown that the properties of features, such as their granularity, scattering, and degree of interaction, strongly impact the process of defining and managing them (Hinterreiter et al. 2020, 2022; Zave 1993; Kästner, Apel, and Kuhlemann 2008).

We investigate whether and how music features can be used for managing revisions and variants in music notation. Our earlier research demonstrated how music scores can be generated automatically using music features, based on mappings of these features to scores managed in a variation control system, which can handle both revisions and variants in a feature-based manner, thereby going beyond conventional version control systems like Git (Grünbacher, Hanl, and Linsbauer 2021). Our results, however, showed that music features are often fine-grained, scattered, and highly interacting, which motivated the research reported in this article. Specifically, we present experiences and lessons learned in the process of developing an approach for managing and composing (i.e., combining) features of music encoded in a DSL. Our article provides the following three contributions:

1. We refine and extend earlier work on music engraving and variability (Grünbacher, Hanl, and Linsbauer 2021) and provide important details of our architecture and implementation. Furthermore, earlier work provided only a preliminary evaluation; it did not investigate the incremental refinement of feature-to-artifact mappings in a feature-oriented workflow.
2. We thus report an experiment investigating the quality of evolving feature-to-artifact mappings by fixing incorrect variants of music artifacts and storing them again in a variation control system, thus assessing its ability to handle fine-grained, scattered, and interacting features (which is our research question #1, hereafter called RQ1). The experiment also studied the impact of different thresholds denoting the number of features considered when computing interactions between them. This is our research question #2, hereafter called RQ2. This is important to understand the trade-off between the quality and performance of the automated analyses.
3. Finally, we provide a discussion of lessons learned in our multidisciplinary research on the intersection of software engineering, music engraving, and digital publishing.

The remainder of the article is organized as follows: We first discuss the background for our interdisciplinary research, and we provide illustrative examples of music features encoded in the DSL LilyPond to motivate our research on automatically tracing features in music. We also briefly describe the background on DSLs for music notation and feature-based version control. We then describe the workflow and architecture of LilyECCO, our DSL-specific extension to the version control system ECCO—in particular, the representations and transformations we used. We present an experiment assessing the correctness of our approach and the impact of using different commit strategies and thresholds for the feature interaction order. We report experiences, lessons learned, and threats to validity. Finally, we discuss our work with respect to related research, and we present conclusions and an outlook on future research plans.

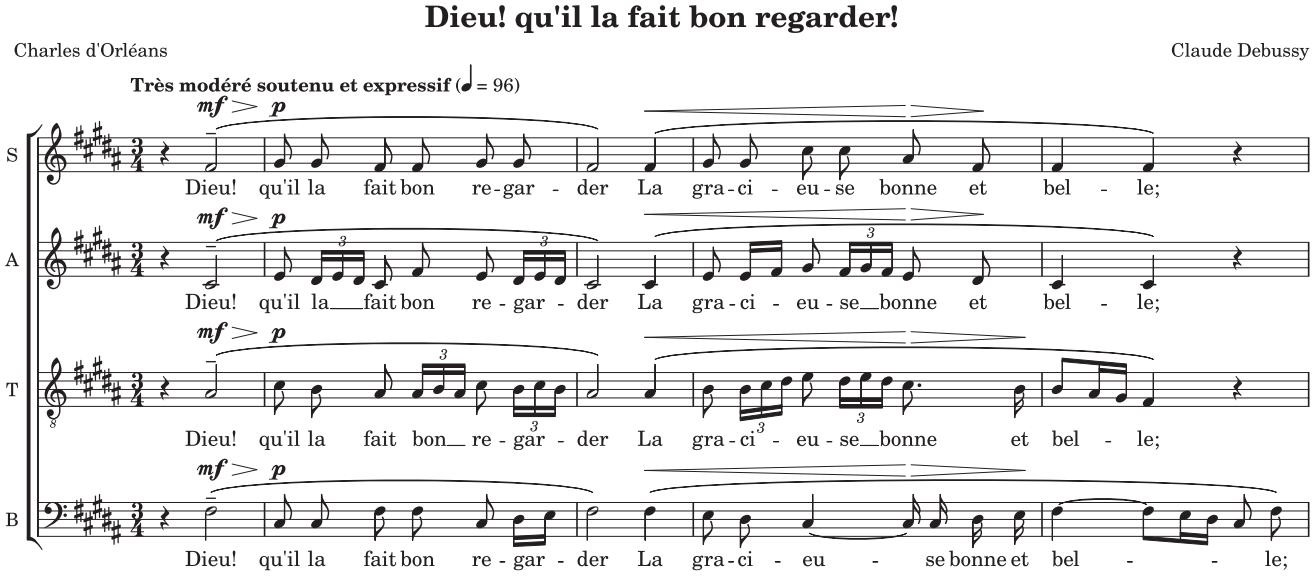
Background

We provide a brief introduction to the main concepts behind our interdisciplinary research: the DSL LilyPond, guidelines for managing revisions and variants in music, the notion of features in music scores, and version control systems. Figure 1 shows a musical excerpt that will be discussed throughout the article.

The DSL LilyPond for Music Engraving

High-quality typesetting of music is challenging (Gould 2011), and music engravers nowadays adopt

Figure 1 The first few bars of the piece "Dieu! qu'il la fait bon regarder!" (Debussy 1908). Music features exist for setting up the score; for defining note pitches and durations of the four voices; for handling texts like the lyrics and the header; and for articulations, dynamics, and slurs. The processing of this Debussy excerpt is discussed further in the ensuing figures and text.



numerous DSLs and music notation tools for that purpose (Lemberg, Moser, and Liska 2020). For instance, the DSL LilyPond provides a comprehensive set of commands for engraving music and automatically creating scores (LilyPond 2024). LilyPond reflects traditional rules of manual engraving and computes the details of music layout so that music engravers can focus on the music instead of tweaking the layout, similarly to what LaTeX does for text documents. LilyPond relies on the text-based entry of commands and provides support for engraving classical music, complex notation, early music, modern music, tablature, vocal music, and lead sheets. For instance, Figure 2 shows some simple examples of LilyPond code, including constructs to define musical elements such as key signatures, time signatures, pitches, durations, slurs, dynamics, lyrics, and articulations. The LilyPond compiler processes the DSL code in multiple stages (Sandberg 2006): First, music expressions are converted into expressions in the Scheme programming language that represent the various music events and their relationships. In the second stage, the music events are assigned to contexts in which the music is engraved. Finally, the events are processed to produce graphical or MIDI output.

Guidelines for Revisions and Variants in Music

Our work was also inspired by the MEI Guidelines for managing revisions and variants in music. This open-source effort for encoding musical documents in a machine-readable structure (Teich Geertinger 2021) provides the following requirements and guidelines for creating digital scholarly editions of music:

Encoding differences between multiple exemplars of the same musical work. This means the ability to define one or more alternative encodings (i.e., variants), or to encode a sequence of readings (i.e., revisions).

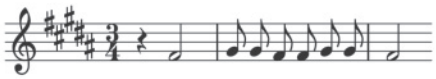
Supporting versions at different granularity. Textual variation can occur at nearly any point in a musical text. For example, it may be used to indicate minor differences such as stem directions. However, versions may have more significant differences, such as extra measures.

Providing different score layouts. Different layouts of scores may be required, even when the musical content itself remains the same. For example, two sources may have the same content, but a different ordering of the staves on which it is written.

Figure 2 Examples of revisions and variants when coding the opening bars of the soprano voice of "Dieu! qu'il la fait bon regarder!" (Debussy 1908). The LilyPond code on the left produces the score shown on the right.

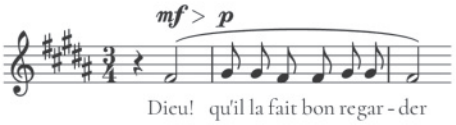
***rev1:* features setup, notes**

```
\key b \major
\clef "treble"
\time 3/4
r4 fis2 gis8 gis fis fis gis gis fis2
```



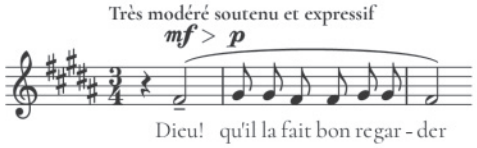
***rev2:* features setup, notes, slurs, dynamics, lyrics**

```
\key b \major
\clef "treble"
\time 3/4
r4 fis2\mf\>\( gis8\p\! gis fis fis gis gis fis2\)

\addlyrics {  Dieu! qu'il la fait bon re -- gar -- der }
```



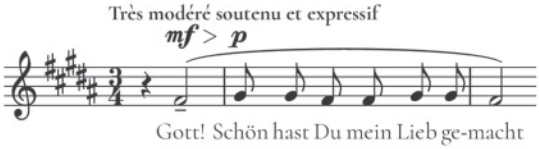
***rev3:* features setup, notes, slurs, dynamics, lyrics, texts, articulations**

```
\key b \major
\clef "treble"
\time 3/4
\tempo "Très modéré soutenu et expressif"
r4 fis2--\mf\>\( gis8\p\! gis fis fis gis gis fis2\)

\addlyrics { Dieu! qu'il la fait bon re -- gar -- der }
```



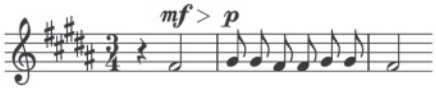
***var1:* features setup, notes, slurs, dynamics, lyrics, texts, articulations, german**

```
\key b \major
\clef "treble"
\time 3/4
\tempo "Très modéré soutenu et expressif"
r4 fis2--\mf\>\( gis8\p\! gis fis fis gis gis fis2\)

\addlyrics { Gott! Schön hast Du mein Lieb ge -- macht }
```



***new version:* features setup, notes, dynamics**

```
\key b \major
\clef "treble"
\time 3/4
r4 fis2\mf\> gis8\p\! gis fis fis gis gis fis2
```



Dealing with editorial markups and interventions. Composers or copyists often mark up different revisions of manuscripts—for example, to correct apparent errors; to indicate the regularization of variants or irregular, nonstandard, or eccentric forms (e.g., the encoder may indicate that the clef has been modernized into a G clef); or to handle editorial additions, suppressions, and omissions.

Understanding the genesis of compositions. Musicologists study the creation of a musical work in all its recorded details—from the first sketches to the complete text—to investigate a composer's working and thinking processes and to understand the gradual elaboration of musical thoughts.

Features in Music Scores

In software engineering, a "feature" has been defined in very general terms, i.e., as "a distinguishable characteristic of a concept (system, component, etc.) that is relevant to some stakeholder of the concept" (Czarnecki and Eisenecker 2000). However, the question of what constitutes a feature depends on the application context and the domain of interest (Berger et al. 2015). So, as we also pointed out in an earlier paper (Grünbacher, Hanl, and Linsbauer 2021), the question in the context of music is: What are the distinguishable characteristics of music relevant to a music engraver? For instance, in our

example in Figure 2 the engraver started with a first revision (rev1) that included features for setting up the key signature, clef, and meter (lines 1–3) as well as a feature for the notes (line 4). The engraver in rev2 further added dynamics to the first two notes (mezzo forte, piano, and a decrescendo indicated by the hairpin) as well as the French lyrics. In rev3, the engraver added a textual indication of tempo and expression, as well as an articulation (tenuto) on the first note. Finally, in var1 the engraver created a variant of the music by replacing the French lyrics with ones in German. (We discuss the "new version" later.)

However, the question of what constitutes a feature also depends on the preferences of individual users. For instance, additional notation information, such as asking LilyPond to use a custom notehead color, could be included with the feature notes, or added as a separate feature, according to the preference of the human engraver.

As our examples show, music features can be used to meaningfully track changes when creating revisions of a digital music artifact (e.g., when adding dynamics to the notes). However, they may also be used for the purpose of defining different variants of an artifact (e.g., when adding a translation of the lyrics for a German edition of a piece). However, we also noticed some particular properties of features that are challenging for feature-based approaches.

Feature granularity. This property refers to the size of individual elements mapped to a particular feature. For instance, the lyrics in our example are a coarse-grained feature, while the articulation represents a fine-grained feature.

Feature scattering. This characteristic refers to the number of different locations of a feature's implementation. For instance, the setup feature defining the key, clef, and time is defined in a single location only, while phrasing slurs or beams require multiple noncontiguous code locations.

Feature interactions. Interactions between features exist, if one feature modifies or influences other features in defining overall behavior (Zave 1993). This usually means that code needs to be available that ensures the joint operation of the interacting features. Such structural interactions manifest at source level whenever code is included in a variant because of a combination of selected (or unselected) features of a variant (Apel et al. 2013b; Fischer et al. 2018). For instance, the score definition of our complete example piece (see Figure 1) contains code (not shown) that defines a staff for both the alto music and the alto lyrics. This code is required obviously only if both features are present in a variant.

Version Control

As is common in an artistic field and typical of any creative process, musical works frequently exist in numerous versions, which often reflect the works' history and genesis during composition, publication, and performance. Managing versions of music scores thus becomes particularly important. As discussed, versions are either revisions, e.g., caused by changes or editorial markups made over time, or variants, e.g., different editions of the same musical work. For instance, Figure 2 shows three revisions and two variants (var1 and new version) of the small excerpt from Claude Debussy's vocal piece.

The field of configuration management has developed a wide range of methods and tools (Conradi and Westfechtel 1998), which can also be used to manage versions of artifacts in the domain of music. Tools adopting an *extensional* versioning strategy assume that versions are explicitly enumerated, and the tools then allow retrieving the versions that have been created and committed before. Examples are Git or Subversion, which keep track of the evolution history by assigning revisions to different states. For instance, a music engraver could retrieve ("check out") the versions rev1, rev2, and rev3 of Figure 2 from the configuration management system if they have been provided ("committed") as such to the system before. Since evolution rarely happens just linearly, such tools provide branching mechanisms for dealing with revisions and variants created by multiple engravers for different purposes. For instance, short-term branches might be created for as long as it takes to revise a score and then merge back the changes to the original score. Long-term branches can be used to manage variants of scores

(e.g., to separately maintain the variant var1 seen in Figure 2).

The simple examples in Figure 2 already show two limitations of current extensional systems for the purpose of versioning music artifacts: (1) Using long-term branches to manage versions quickly leads to maintenance problems. For instance, assume that an engraver erroneously committed a note with a wrong pitch. The correction of the pitch then needs to be propagated manually to all relevant branches. (2) Current extensional version control systems rely on computing line-based differences between versions. While this approach is regarded sufficient in many domains and general-purpose programming languages like C or Java, it does not allow one to deal with fine-grained and scattered changes common in DSLs for music. For instance, in rev2 we would only know that line 4 has changed, but would not see that the engraver specifically added a slur and the dynamics to the score.

Version control systems like ECCO or SuperMod (Linsbauer et al. 2021) aim to overcome these limitations by using a feature-based mechanism for managing revisions and variants and an *intensional* versioning strategy. Such tools automatically track fine-grained changes to artifacts at the level of features and thereby also avoid branches. This is achieved by creating an artifact tree and then computing differences between versions by comparing their artifact trees. This allows one to analyze fine-grained properties such as pitches or the durations of notes. This is also important in the case of scattered features, e.g., to determine the start and end of a phrasing slur or hairpin. Intensional systems then allow one to generate arbitrary new versions—versions that have not been explicitly enumerated and committed before—based on features and configurations. For instance, an engraver could specify the configuration setup, notes, dynamics to check out a new version, as shown in our running example in Figure 2.

LilyECCO Workflow and Architecture

The ability to trace features in musical scores depends on the DSL used to encode the music, the notion of music features, and the workflow and discipline of engravers. The study presented in this article uses the feature-based version control system ECCO, a variation control system (Linsbauer et al. 2021) for managing both revisions and variants of digital artifacts. In particular, we used our LilyECCO extension (Grünbacher, Hanl, and Linsbauer 2021), two plug-ins that allow ECCO to handle the DSL LilyPond. We revised and adapted these plug-ins as part of the research for this article.

Workflow

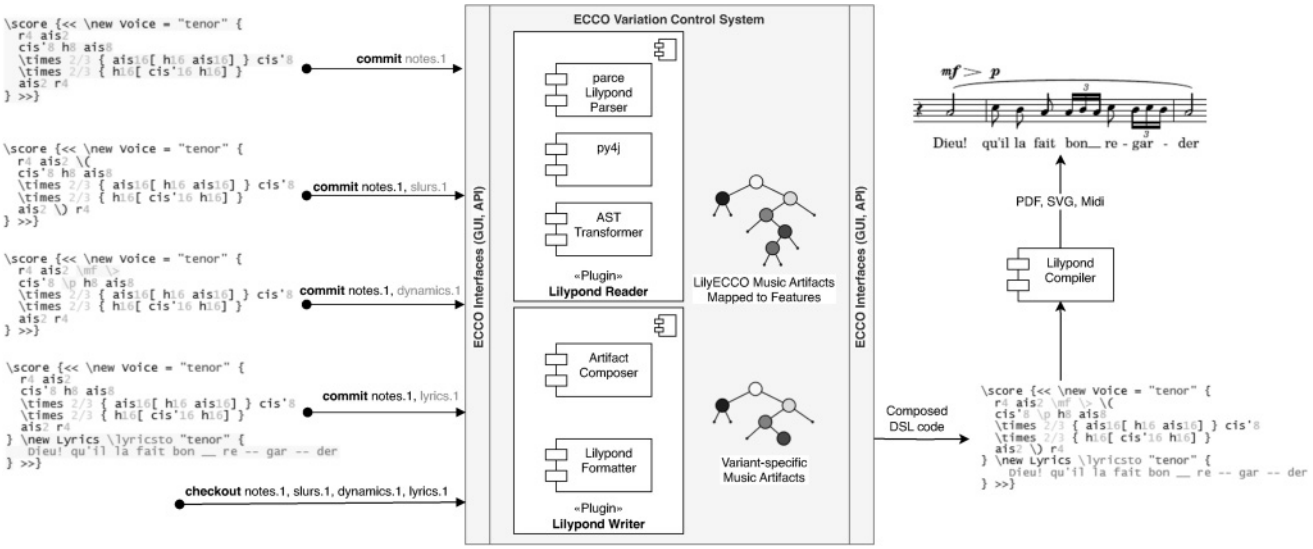
A music engraver working with LilyECCO commits revisions and variants of features, retrieves variants by checking out configurations, and fixes and recommits invalid variants of scores.

Committing Features

The left part of Figure 3 shows how a music engraver commits features incrementally. In the example, the engraver first commits a feature containing the notes, and then adds features for slurs, dynamics, and lyrics. After each step, the engraver commits changes by defining combinations of features, as shown by the commit commands on the arrows. Revisions of features can be indicated by a number after the feature name. Specifically, a commit in the context of LilyECCO means saving changes made to a set of LilyPond files into the repository's history. A commit captures a snapshot of the current state of the files at a specific point in time. While conventional version control systems only ask users to provide a message describing the changes or improvements made in that commit, the engraver in LilyECCO additionally provides names of the feature(s) involved in a commit. LilyECCO does not impose a particular order of committing features, but the order and discipline of the engraver does matter, as we will show in our experiment.

As soon as the engraver commits a change, the newly added or changed LilyPond source code must be mapped to the specified features. In LilyECCO, this is done by automated code analysis extracting the newly added source code artifacts and mapping

Figure 3 LilyECCO workflow and architecture. The left part shows commands an engraver sends to LilyECCO (Grünbacher, Hanl, and Linsbauer 2021) to commit and check out music features. The middle part shows the main components of LilyECCO, i.e., its interfaces, the ECCO version control system, and the plug-ins for the DSL LilyPond. The right part shows LilyPond code produced by LilyECCO. It can be further edited by the engraver and then processed by the LilyPond compiler. The color coding in this figure is more easily distinguishable in the supplementary material at https://doi.org/10.1162/COMJ_a_00691.



them to the committed feature(s) (cf. Figure 3 and Figure 4). For instance, when committing notes.1 and slurs.1 in the second step of Figure 3, LilyECCO determines that the newly added code in the second and sixth code lines (highlighted here with a light gray background for the purposes of illustration, and with yellow background in the color version of the figure)—namely, 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 (Grünbacher, Hanl, and Linsbauer 2021). Specifically, ECCO determines the parts of the DSL code that are common to multiple versions and creates mappings to features that are common to those versions. ECCO also determines the parts of the DSL code that only exist in some versions and creates mappings to features only existing in those versions (Linsbauer et al. 2022). As can be seen in the example, the characteristics of the committed features differ significantly. While the feature lyrics in this example is coarse-grained and located in one position, the features dynamics and slurs are fine-grained and scattered across the DSL code.

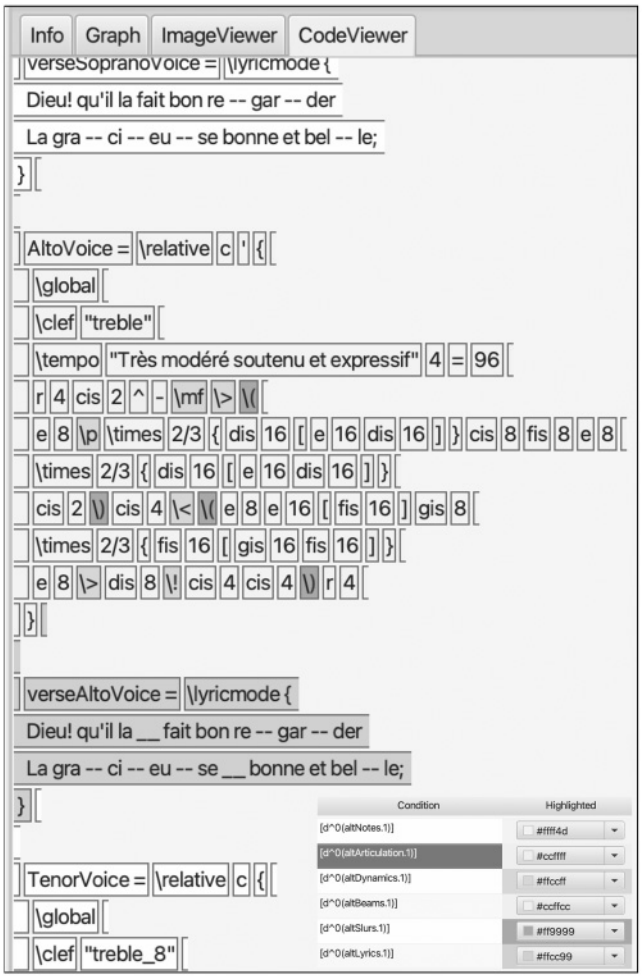
Checking Out Variants

ECCO is an intensional version control system, i.e., one that uses features and configurations to generate versions. This means that ECCO can automatically compose versions even if they have not been explicitly enumerated and committed before, which is not possible with conventional extensional version control systems like Git (Conradi and Westfechtel 1998). In ECCO, an engraver can at any time check out features (or combinations of features) stored in the repository to generate an arbitrary variant of the music artifact. For example, in Figure 3 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 the text on the arrow. LilyECCO automatically generates the code shown on the right based on the feature-to-code mappings stored in ECCO (Grünbacher, Hanl, and Linsbauer 2021).

Fixing and Recommitting Variants

The feature-to-artifact mappings are obtained by LilyECCO based on analyzing the incrementally added features. The input may obviously be ambiguous, e.g., if tangled features or only very few versions are committed. The generated DSL code may thus be syntactically incorrect. However, in such cases, the engraver can modify the automatically created code to fix a new revision or variant, and again commit the changes, thus improving the feature-to-artifact

Figure 4 The LilyECCO GUI showing selected music features of the alto voice of an automatically generated variant of the score for the Debussy excerpt. The main part shows the code of this section in the DSL LilyPond. The colors indicate the presence of specific features in the code. The lower right part provides a legend explaining the meaning of the colors. A color version of the figure is available in the supplementary material at https://doi.org/10.1162/COMJ_a_00691.



mappings, as we will show in our experiment. To facilitate this task, ECCO provides hints to identify surplus code mapped to multiple features, or code missing to handle interacting features that have never been committed together before.

Architecture and Implementation

Figure 3 shows the main components of LilyECCO. The tool exploits ECCO's plug-in architecture, which allows one to extend it with components that translate artifacts into its internal tree structure, as well as components that compose artifacts from the internal tree structure. We developed two plug-ins to support the DSL LilyPond: (1) the LilypondReader, which parses the DSL code and maps it to ECCO's feature-aware tree structure; and (2) the LilypondWriter, which generates DSL code for a music variant requested by an engraver in the form of a configuration expression. ECCO can be used with a simple graphical user interface (GUI), or its API can be called by programs. For testing ECCO, we developed a specific GUI to visualize the feature-to-code mappings it computed, whereas for our experiment we used the API.

LilypondReader

This plug-in, which is responsible for mapping LilyPond code to a node structure of artifacts, relies on three modules to generate an abstract syntax tree (AST) suitable for our purpose: (1) We use the Python package parce (https://parce.info) to parse the newly committed LilyPond input into a tree structure based on the LilyPond DSL definition. ECCO is implemented in the programming language Java, while parce is written in Python. (2) We thus use Py4J (https://www.py4j.org) to transfer the Python objects to the Java Virtual Machine. After creating a gateway and setting up an entry point, a small Python script is executed, which starts the parser and returns a stream of tokens representing a code model of the music artifact. (3) Our AST Transformer then optimizes this structure. For instance, while for certain features it is essential to keep fine-grained nodes (e.g., dynamics), a more coarse-grained view is sufficient for other features—e.g., it is convenient to treat the lyrics of a particular voice as a feature. However, since each syllable and hyphen would result in a token (e.g., `re -- gar -- der` consists of five tokens), we instead transform it to a single node in this case. The AST Transformer filters and aggregates strings, lyrics, variable definitions, and Scheme numbers for the purpose of optimization, as reducing the number of nodes improves performance and removing the number of identical tokens improves the composition of variants by avoiding the mixing of characters across features. For instance, the assignment character ("=") occurs many times (e.g., `\new voice = "tenor"` in

Figure 3), but is distinct when merged with the variable declaration artifact.

LilypondWriter

This plug-in creates variant-specific music artifacts to then produce LilyPond DSL code, which can be compiled to PDF, MIDI, or SVG by LilyPond. The Artifact Composer creates a music artifact tree for a specific variant: Checking out a combination of features (or a combination of revisions of features) is achieved by first creating variant-specific music artifacts for the features selected by the music engraver in a configuration expression and then triggering the LilyPond Formatter, generating LilyPond DSL code. The formatter takes care of DSL-specific rules for inserting white spaces in the generated code. Valid LilyPond code requires space characters between all tokens, except for a few cases (e.g., numbers in Scheme code embedded in the DSL code). The LilypondWriter applies several rules to improve the readability of the code, for example, no spaces are inserted between notes and their durations, as is common in LilyPond.

LilyECCO GUI

We adapted the ECCO GUI to highlight music features in the DSL code using different user-defined colors. ECCO stores implementation artifacts as a generic tree structure. Nodes of the tree are labeled with presence conditions, indicating when a specific artifact or part of an artifact shall be included in a specific variant. Users can define colors for presence conditions computed by ECCO. For instance, in Figure 4 (shown in grayscale here but in color in the online supplementary material) yellow is used to mark the notes, blue for articulations, and pink for dynamics. When committing new features or recommitting existing variants, the tree is updated automatically by recomputing the presence conditions of the affected artifacts.

Experiment

An earlier study provided a preliminary evaluation of LilyECCO by replaying an evolution history to study the performance of the commit operations and to assess the correctness of few selected variants (Grünbacher, Hanl, and Linsbauer 2021). However, that study did not investigate the fixing and recommitting of incorrectly generated variants, e.g., those caused by feature interactions not managed in the code. We thus study the incremental refinement of feature-to-artifact mappings via committing DSL code needed to manage feature interactions that lead to incorrect variants. In particular, we investigated two research questions:

RQ1. How do feature-to-artifact mappings improve when fixing and recommitting incorrect variants? Based on our data set, we randomly created variants and checked their correctness. Since visual checks were already done in earlier research to check the semantics, we relied on the LilyPond compiler and regarded a variant as correct if it compiled without errors. We further used the level of ambiguity computed by ECCO as an indicator of the quality of the variants, in order to demonstrate the effect of improving feature-to-artifact mappings over time.

RQ2. How is correctness affected by the threshold for feature interaction order? The variation control system ECCO allows one to set a threshold balancing the quality of the analyses versus their computational efficiency. The threshold defines the maximum size of clauses in presence conditions (cf. Figure 4, bottom right), i.e., the number of feature literals in a conjunction. It corresponds to the number of interacting features and controls the number of clauses, which grows exponentially with the number of features (Linsbauer et al. 2022). The threshold can be freely configured, but for the evaluation in this article it was set to a maximum of three, four, or five interacting features, based on previous empirical research (Fischer et al. 2014, 2016).

Data Set

Our experiment is based on the excerpt of the piece "Dieu! qu'il la fait bon regarder!" by Claude Debussy (1908) shown in Figure 1. As discussed in the examples provided in the Background section, this

musical composition is characterized by fine-grained, scattered, and interacting features. We identified 24 features in the Debussy excerpt: a feature header and—for each of the four voices—features for notes, articulations, dynamics, beams, slurs, and lyrics. (An exception is the soprano voice's lack of any beams.) We created multiple different repositories in our data set by combining different commit strategies with different feature interaction thresholds to investigate the impact of the independent variables on the quality of the feature-to-code mappings and the resulting music variants, as follows.

Considering Different Commit Strategies

As shown in Figure 3, LilyECCO assumes an incremental approach when committing the music features to the repository. This is consistent with Zave's view of a feature as an "increment of functionality, usually with a coherent purpose" (Zave 2004). This means in our case that variants take the form $B + F1 + F2 + F3 \ldots$, where B is some base feature, each F_i represents the artifacts mapped to that feature, and + denotes some composition operation, realized in our case by the ECCO variation control system.

We used two different strategies when creating our data set, to simulate two different levels of discipline of a music engraver working with LilyECCO:

In the first strategy—*adding to full variant (FUV)*—we committed new features on top of all already existing features in the current variant. For our example, applying this strategy resulted in 24 initial commits for the FUV strategy, incrementally adding the features one by one. This simulates the common workflow of a music engraver working with the latest version of a piece when adding a new feature. This strategy may reduce the quality of the feature-to-artifact mappings, as ECCO's strategy for "diffing" (i.e., detecting differences) may lead to ambiguous mappings, i.e., LilyPond code mapped to multiple features, if more features are already present.

In the second strategy—*adding to minimum viable variant (MVV)*—we committed new features on top of a set of only those base features in the current variant that were required for adding the new feature. This strategy simulates an engraver with a high commit discipline working with a minimal variant containing fewer features, thus potentially improving the quality of the feature-to-artifact mappings.

The MVV case had one commit for the header, six commits for the soprano voice, seven commits for each of the other three voices, and one final commit for all 24 features, resulting in a total of 29 commits. Each voice was committed in following order: header + notes; header + notes + articulation; header + notes + dynamics; header + notes + beams (omitted for soprano voice); header + notes + slurs; header + notes + beams + slurs + lyrics; header + notes + articulation + dynamics + beams + slurs + lyrics. This reflects feature dependencies of LilyPond code for valid compositions, as the header and notes of any voice need to exist as a base. Lyrics depend on the features beams and slurs and were therefore committed together with them.

Considering Thresholds for Feature Interaction Order

We further created the repositories using different thresholds to control the number of features involved in an interaction. Artifact snippets have the order 0 if they label a single feature and do not interact with any other features. As described in Fischer et al. (2014, 2016) an order n thus represents the interaction of $n + 1$ features. The order threshold thus denotes the maximum number of features to be considered when computing interactions. For the evaluation in this article, the order threshold was set to 2, 3, or 4, which respectively denote a maximum of three, four, or five interacting features.The parameters were set based on earlier studies (Fischer et al. 2014, 2016), to allow comparisons of our results with other languages and artifacts.

We combined the two commit strategies with the three feature interaction thresholds, resulting in six repositories overall for our experiment. For instance, FUV2 refers to a repository created with maximum order 2 by always committing to the full variant, while MVV4 refers to a repository created with maximum order 4 by committing

to a variant with the minimum set of features required. Creating multiple different repositories in our data set allowed us to investigate to what extent the engraver discipline and the threshold for feature interaction order impact the quality of the feature-to-code mappings and the resulting music variants.

Method

In our experiment, we incrementally refined feature-to-artifact mappings by fixing and recommitting variants to each of the repositories in our data set as follows:

1. Creating random variants. We created 50 variants by first generating a random configuration for each variant and then composing it with ECCO's checkout operation. All 50 randomly generated configurations for the evaluation respected the basic feature dependencies described above, in order to prevent incorrect compositions in the first place.

2. Determining correctness and quality of variants. As an obvious check of correctness, we first determined whether each of the variants compiled correctly. In addition, we considered ECCO's additional feedback on the quality of the variants. Specifically, when composing new configurations, ECCO computes hints, which are useful for finding artifacts that may have to be added or removed for completing a product. The hints help one identify potential surplus artifacts (e.g., duplicate code that needs to be removed) as well as missing artifacts (e.g., code that needs to be added to account for feature interactions not yet covered). The hints thus help one understand possible feature interactions or dependencies, in particular when combining features that were never used together in a configuration. This feedback supports the completion of a variant by showing clauses of the presence conditions used to generate a configuration.

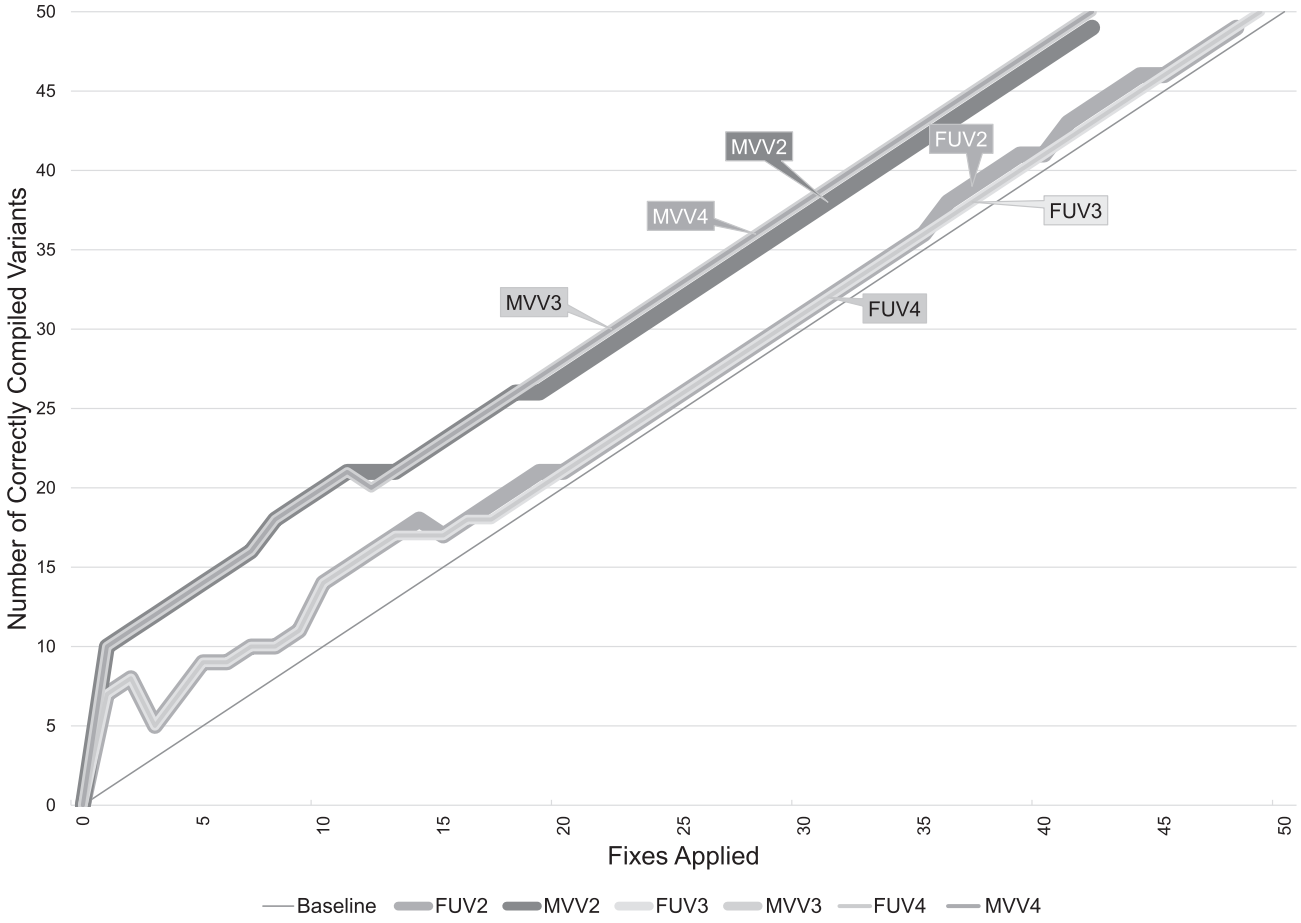
Using the numbers reported by ECCO for missing code and surplus code as an estimate for the quality of a variant, we computed a metric indicating the ambiguity reduction of all variants achieved after a particular fix. In particular, we computed the total ambiguity reduction *TAR* of all n variants after a fix as follows:

$$TAR = 1 - \frac{\sum_{i=1}^{n}(missing_i + surplus_i)}{\max_{i=1,\ldots,n}(missing_i + surplus_i)}$$

3. Fixing an incorrect variant. As pointed out, in the case where a variant never existed before, it is likely that interactions between features are not handled in the DSL code, which often leads to incorrect code. A user would manually fix these problems by removing surplus code (e.g., code mapped to multiple features), or adding missing code (e.g., code handling the joint working of newly combined features). This would not be possible for the almost 20,000 fixes required in our automated experiment. Therefore, we developed a script for our data set that creates correct code for a given configuration expression. Specifically, the script was created by analyzing the feature-to-code mappings existing in the ECCO repository after the final commit (cf. Figure 4 showing the LilyECCO tool, which uses colors to highlight the location of features). The contiguous locations of features were used as a starting point for building the ground truth needed to generate the fixes for the excerpt of Debussy's piece. The script stores the feature mappings of each contiguous location, e.g., AltoVoice = . . . r4 cis2 is mapped to altNotes.1 and the token ^ - is mapped to altArticulations.1. The script then refines these initial mappings, thereby encoding the knowledge of an engraver that is required to eliminate the ambiguities in the ECCO repository. The resulting script preserves the order of the mapped contiguous feature locations and generates correct Lilypond code of a variant for a specific configuration.

4. Recommitting the variant. After fixing the problems and checking for successful compilation, we committed the changes back to ECCO for this variant, thereby also incrementally and automatically updating the feature-to-artifact mappings. We then continued in Step 2 with the next variant, to understand the impact of our fixes on the quality of all the other variants.

Figure 5 Overall variant correctness after fixing and recommitting incorrect variants for the Full Variants (FUV) and Minimum Viable Variants (MVV) commit strategies, as well as for different thresholds for feature interaction order (2–4). The thickness of overlapping lines was adapted to ensure their visibility. A color version of the figure is available in the supplementary material at https://doi.org/10.1162/COMJ_a_00691.



Results

Figure 5 shows the correctness of the variants for data sets created with different commit strategies (Full Variants and MVVs) as well as for different thresholds for feature interaction order (2–4). The baseline shown for comparison indicates the minimum expected effect of fixing and recommitting a single incorrect variant, i.e., that exactly this variant remains operational when subsequent fixes are applied. When looking at the correctness results, we see that this extensional versioning, i.e., retrieving variants committed before, worked for all strategies. However, we see that fixing a single variant occasionally also corrected other variants, meaning that intensional versioning works for variants that were not submitted as such to the repository (examples are places in the figure where the slopes are greater than 45 degrees). This result also holds regardless of the strategy investigated. In other words: any fix applied remains, and certain fixes also correct other variants in our case of a DSL with fine-grained and scattered features. However, in a few cases a fix also caused problems in other variants, as can be seen in occurrences of negative slopes.

Figure 5 further shows that higher discipline (MVV) brings additional benefits, as all MVV results are higher than their FUV counterparts. Regarding feature interaction orders, we see a small improvement, but not enough to justify the much

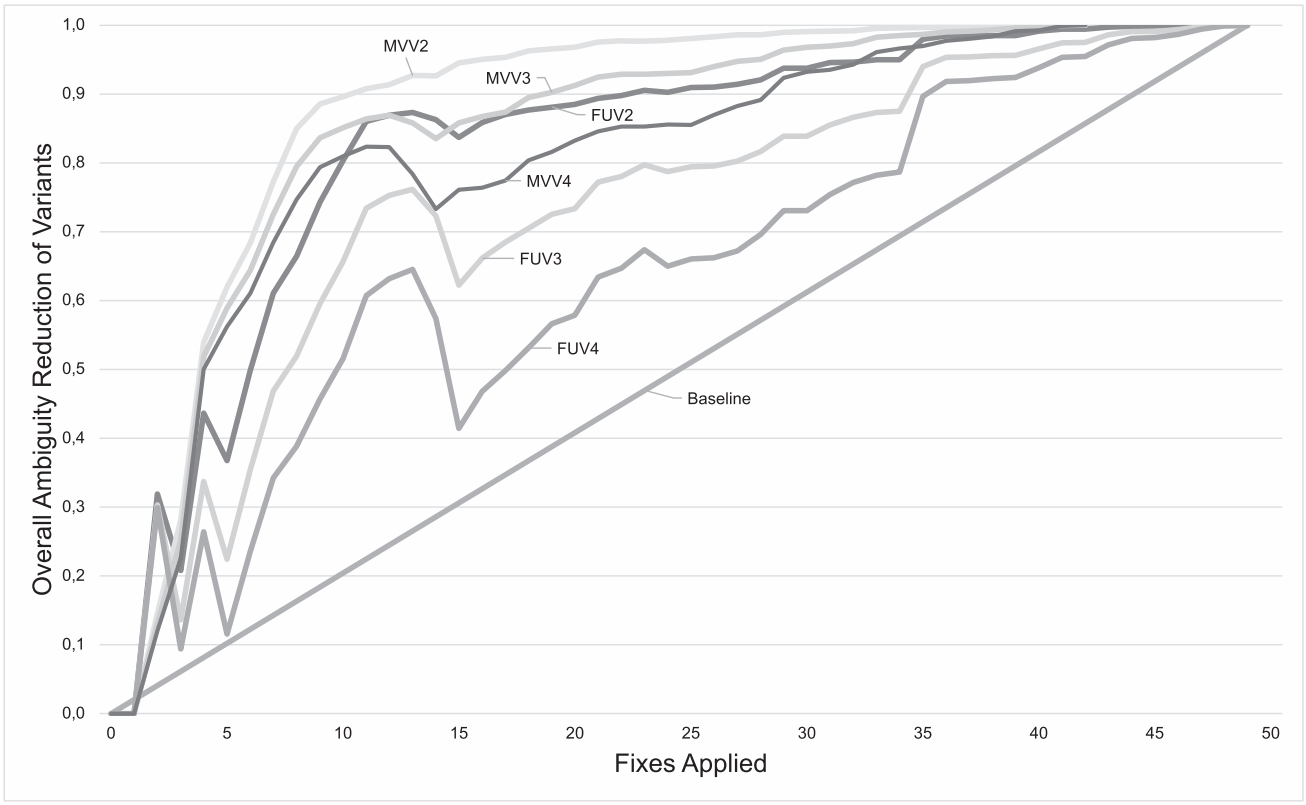


Figure 6 Overall ambiguity reduction effect of fixing and recommitting variants for different commit strategies using Full Variants (FUV) and Minimum Viable Variants (MVV) as well as different levels of feature interaction orders (2–4). A color version of the figure is available in the supplementary material at https://doi.org/10.1162/COMJ_a_00691.

higher computational effort required to compute the feature-to-artifact mappings.

Figure 6 allows one to distinguish the different strategies even further. It compares the ambiguity reduction effect of fixes for all variants (TAR) for different commit strategies, again using FUVs and MVVs, as well as different thresholds for feature interaction order (2–4). We see that even when fine-grained, scattered, and interacting features are present—as is the case for our music artifact—an order threshold of 2 (i.e., a maximum of three interacting features) already had a significant effect of improving other variants. For instance, in the case of MVV2 the ambiguity was reduced by about 90%, showing that relatively few fixes already had a strong impact.

Figure 6 confirms the results obtained regarding the commit strategy, i.e., higher discipline pays off. For example, compare the curves for MVV2 and FUV2.

Regarding performance, a maximum order of 2 obviously gave the best results, but Figure 5 shows that only 49 of 50 configurations were compiled successfully. This was caused by one configuration returning an empty checkout result, even if fixed before. Interestingly, different configurations were affected for the FUV and MVV strategies. Further investigation showed that this was caused by an empty presence condition of an artifact snippet, meaning that no artifact(s) could be found after some additional commits when considering three interacting features only. We thus additionally investigated an adaptive strategy as proposed by Fischer et al. (2016) and temporarily raised the maximum order to 3 for these rare cases, which correctly computed the output of all 50 configurations.

Lessons Learned

We regard the following lessons learned as useful for researchers and practitioners working on similar challenges.

Consider DSL specifics when versioning fine-grained and scattered features. Conventional version control systems lack DSL-specific support as well as support for different characteristics of features. Regarding RQ1, we saw that LilyECCO can manage and generate fine-grained and scattered features with high correctness, even for intensional versioning cases. This was achieved by developing a plug-in that considers language-specific properties, going beyond existing tools' line-based detection of differences. LilyECCO computes differences by comparing ASTs, and thus it is able to deal with different levels of granularity as well as different levels of scattering of music features. In addition, our LilypondReader and LilypondWriter perform additional transformations, thereby considering domain-specific rules for versioning, such as rules regarding the granularity for different kinds of music features. Our plug-ins also include rules for handling lower-level Scheme functions or parameters that may eventually be used by an engraver to control the output.

Commit discipline pays off. Our experiments further showed that when managing fine-grained and scattered features in a DSL, the correctness of feature-to-artifact mappings can be improved by committing new features on top of the minimum set of features required for a change. Interestingly, the average ratio of all fixes of *MVV*2/*FUV*2 was 9%, the average ratio of *MVV*3/*FUV*3 was 24%, and the average ratio of *MVV*4/*FUV*4 was 47%, indicating that a higher discipline becomes even more important when using higher thresholds for feature interaction order. This confirms results by Ratzenböck et al. (2022), who investigated the effects of tangled code changes on correctness when replaying a version history of a conventional version control system to create a repository of a variation control system. It is difficult to estimate whether and to what extent following the MVV-like process to achieve a higher commit discipline would increase the effort for an engraver. In a real-world situation, the fixes would likely be verified by a human, so suggesting an optimal proportion of the number of fixes applied to the total number of variants would be an interesting research question for a future study. However, even given the potential for such a study, the current results already show that the MVV strategy paid off for all cases we investigated.

Balance correctness and computational effort. Regarding RQ2, we saw that a lower threshold for feature interaction order was sufficient even for fine-grained and scattered features in a DSL, as the FUV2 and MVV2 strategies performed very well compared to strategies analyzing a higher feature interaction order. This is a very positive result, as it shows that the computational complexity involved in a variation control system can be be controlled without a strong negative impact for practical use cases. Furthermore, our results confirm the results of earlier studies investigating the feature interaction orders of systems written in general-purpose languages (Fischer et al. 2014, 2016). A maximum of three interacting features was also used for an approach for locating features and their revisions in existing code repositories (Michelon et al. 2020).

Threats to Validity

In our experiment, we assessed correctness by collecting results from the LilyPond compiler and the hints on surplus and missing code computed by the ECCO variation control system. Overall, we created about 20,000 different variants of our piece, making it impossible to visually check all of them. However, we manually inspected a sample of the files to perform additional visual checks. Furthermore, in earlier research Grünbacher, Hanl, and Linsbauer (2021), we already studied the correctness and usefulness on a smaller set of automatically generated variants.

The LilypondReader and LilypondWriter plug-ins have so far been primarily used and tested for vocal music. The piece used in our experiment is an example from the Western music tradition and associated notation, and the results may not generalize well to

other places and periods. For instance, further transformation rules and enhancements may be required when using the full scope of this DSL. However, as part of testing the two plug-ins, we committed and checked out versions for a range of different music pieces from a data set described in a paper by the first author (Grünbacher 2022) and another data set comprising orchestral music, in order to verify the correct operation of the plug-ins before executing our experiments.

The algorithms in ECCO compute tree-based commonalities and differences of code, and the size of the artifact as well as the interactions of features have an influence on performance. We did not focus on performance measurements in this experiment, as Grünbacher (2022) already demonstrated that performance was acceptable for a data set containing 52 music features and a larger commit history.

Related Work

We now compare our multidisciplinary approach with related work in the domains of both music notation and software engineering. Specifically, our research relates to research on music representation and variability, feature-oriented development, DSLs and variability, and variation control systems. Understanding LilyECCO is important for understanding this section, so we have placed this section only after the presentation of our approach and its evaluation.

Music Representation and Variability

LilyECCO uses tree-based code diffing to relate music features to music elements when committing changes to the repository. Similar approaches have been explored in the domain of music notation and representation: Antila, Treviño, and Weaver (2017) pointed out the limitations of line-based diffing and propose a hierarchical diffing approach for collaboratively editing music artifacts. Herold (2020) presented the MusicDiff tool for comparing two files with encoded music scores, which can also visualize the differences between these encodings. However, these approaches do not consider the use of features to label changes, as done in LilyECCO. Fournier-S'niehotta, Rigaux, and Travers (2016) leveraged a music content model for defining virtual corpora of music notation objects. The idea to perform analyses across diverse digital artifacts is also fundamental to LilyECCO when mapping features to realization artifacts. LilyECCO generates snippets, i.e., partial scores, to create new scores based on a selection of features. The idea to generate new scores based on existing ones has also been proposed by Lepetit-Aimon et al. (2016). In their approach, a score can be composed as an arbitrary graph of score expressions. Grünbacher (2022) described an exploratory study on applying variability management when using LilyPond for multi-device rendering and digital publishing of music sheets. The study shows that further types of variability mechanisms are needed at different stages and for different binding times to create a fully automated workflow. Dannenberg (1993) proposed to provide views on a score. Each view "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 a change in a score to be automatically propagated to the parts (views on the score). LilyECCO's composition of a variant based on features can also be seen as a 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. The issue of discovering common patterns is essential for computational music processing. Conklin and Bergeron (2008) proposed feature set patterns as a mechanism for music data mining to discover similarity in musical material across many pieces.

Feature-oriented Software Development

In the field of software engineering, features are used to distinguish individual products of a product line (Berger et al. 2015), and feature-oriented development (Apel et al. 2013a) has been proposed to map features to their realization. However, establishing such mappings is challenging, as feature

implementations usually span multiple diverse implementation artifacts, and the mappings are difficult to create and maintain (Berger et al. 2015; Czarnecki et al. 2012). A common approach is to use extensional versioning tools such as Git combined with annotation-based mechanisms to manage both revisions and variants of software systems (Schulze et al. 2013; Michelon et al. 2020, 2021). However, this approach lacks the ability to create new variants based on arbitrary feature combinations (Linsbauer et al. 2021). Furthermore, it requires manual editing of feature annotations (Michelon et al. 2021), which could instead be automated by a variation control system, as our approach shows. Techniques have also been proposed for mapping features to models. For instance, Font et al. (2016) presented a feature location approach, based on information retrieval techniques, which uses models as feature realization artifacts. The approach is presented for a DSL and uses the Common Variability Language to formalize the model fragments used as feature candidates. Understanding feature interactions is highly challenging in continuously evolving systems, as reported by Zave (1993). Ferber, Haag, and Savolainen (2002) pointed out that interactions are often difficult to represent in feature models. This gap has been addressed by Feichtinger et al. (2021), who present an approach that visualizes complex code-level dependencies in feature models by combining a variation control system with static code analysis.

DSLs and Variability

Grünbacher (2022) observed that mapping features to DSL concepts is of particular interest for the music engraving context: Specifically, in this thread of research Czarnecki and Antkiewicz (2005) have pointed out that the only way to give features semantics is by mapping them to artifacts. Several authors have addressed this issue: Haugen et al. (2008) proposed to express variability in a standardized language independent of some base modeling language, and they demonstrated this for small DSLs and for the Unified Modeling Language. Czarnecki and Antkiewicz (2005) presented a general template-based approach for mapping feature models to variability representations in other kinds of models. A feature model defines a hierarchy of features together with configuration constraints. A model template contains the union of the model elements in all valid template instances. The elements of the model template can be annotated with expressions defined in terms of features. For instance, a presence condition indicates whether the element should be present in or removed from a template instance.

Variation Control Systems

Variation control systems have been conceived as special types of version control systems, with an emphasis on variant management and intensional versioning. They provide a fine-grained variant mechanism based on individual features and configurations, instead of coarse-grained branches that essentially clone the whole system (Stanciulescu et al. 2016; Schwägerl and Westfechtel 2019; Linsbauer et al. 2022). In a survey, Linsbauer et al. (2021) studied selected variation control systems and analyzed why they have not found widespread adoption. In particular, given their focus on variant management, variation control systems often lack support for revisions. Exceptions are SuperMod and ECCO, which can manage both revisions and variants of different types of product line artifacts. SuperMod integrates temporal and logical versioning, allowing the development of product lines in a single-version workspace in a step-by-step manner by using update and commit operations (Schwägerl and Westfechtel 2019). As discussed above, ECCO can be extended with plug-ins that translate artifacts into its internal tree structure (Linsbauer et al. 2022).

Conclusions and Outlook

Our article studied the use of a feature-based version control system in the domain of symbolic music notation. We reported an experiment investigating the correctness of the generated music artifacts regarding the refinement of feature-to-artifact mappings when incrementally committing DSL code

(RQ1) as well as the impact of the order of feature interactions (RQ2). Most importantly, our results show that higher discipline contributes to the correctness of compositions, that lower thresholds for feature interaction order are sufficient even for cases of fine-grained and scattered features, and that an adaptive strategy helps to further improve correctness. Music engraving for digital publishing is an interesting testbed for research on feature-oriented development and DSLs. However, we believe that our findings are of interest for both practitioners and researchers applying DSLs in different areas—for example, when developing and optimizing tools. Our promising results give rise to plenty of opportunities for further research, as follows.

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 might differ from features in the scenario of a music publishing house creating different scores from the same base. Evaluating the usefulness of music features will thus be necessary—e.g., by conducting user studies with music engravers or by analyzing existing evolution histories.

Our study so far did not consider dealing with obvious *feature dependencies* known to educated musicians. For instance, a publisher may want to prepare an edition for performance based on an urtext score presenting the composer's original intentions. It may be musically obvious that if a performer chooses Variant A in one place, they must choose Variant B in another place, and Variant C later. For this kind of support, LilyECCO would need to be complemented with feature models (Czarnecki et al. 2012) to specify what dependencies and constraints exist between music features. This would be similar to what Feichtinger et al. (2021) have done in the domain of industrial automation systems by combining ECCO with feature models. While we assume that a feature model could be used to define the musical logic at a higher level of abstraction, it is an interesting question to what extent the assumptions made about feature dependencies and constraints also hold in the domain of music.

Another important area to look at is *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 might be cognitively demanding and involve difficult mental operations (Blackwell and Green 2003). Similarly, in terms of possible tool support an interesting capability is to *color features* in music score editors, as shown for source code in programming languages (Kästner, Apel, and Kuhlemann 2008). While the LilyECCO GUI provides preliminary support for visualizing the location of features, a feature-based editor would also ease the systematic study of the granularity of music features in realistic workflows.

Acknowledgments

The authors would like to thank the anonymous reviewers and the Editor for their thoughtful, detailed, and encouraging feedback, which helped to improve our manuscript.

References

Antila, C., J. Treviño, and G. Weaver. 2017. "A Hierarchic Diff Algorithm for Collaborative Music Document Editing." In *Proceedings of the International Conference on Technologies for Music Notation and Representation*, pp. 167–170.

Apel, S., et al. 2013a. *Feature-Oriented Software Product Lines: Concepts and Implementation*. Berlin: Springer.

Apel, S., et al. 2013b. "Exploring Feature Interactions in the Wild: The New Feature-Interaction Challenge." In *Proceedings of the Fifth International Workshop on Feature-Oriented Software Development*, pp. 1–8.

Bent, I. D., et al. 2001. "Notation." In *Grove Music Online*. Oxford University Press. Available online at https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000020114. Accessed August 2023.

Berger, T., et al. 2015. "What Is a Feature? A Qualitative Study of Features in Industrial Software Product Lines." In *Proceedings of the 19th International Software Product Line Conference*, pp. 16–25.

Blackwell, A., and T. Green. 2003. "Notational Systems—The Cognitive Dimensions of Notations Framework." In J. M. Carroll, ed. *HCI Models, Theories, and Frameworks: Toward a Multidisciplinary Science*. San Francisco: Morgan Kaufmann, pp. 103–133.

Borum, H. S., H. Niss, and P. Sestoft. 2021. "On Designing Applied DSLs for Non-Programming Experts in Evolving Domains." In *Proceedings of the ACM/IEEE 24th International Conference on Model Driven Engineering Languages and Systems*, pp. 227–238.

Conklin, D., and M. Bergeron. 2008. "Feature Set Patterns in Music." *Computer Music Journal* 32(1):60–70. 10.1162/comj.2008.32.1.60

Conradi, R., and B. Westfechtel. 1998. "Version Models for Software Configuration Management." *ACM Computing Surveys* 30(2):232–282. 10.1145/280277.280280

Czarnecki, K., and M. Antkiewicz. 2005. "Mapping Features to Models: A Template Approach Based on Superimposed Variants." In *Proceedings of the Fourth International Conference on Generative Programming and Component Engineering*, pp. 422–437.

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

Czarnecki, K., et al. 2012. "Cool Features and Tough Decisions: A Comparison of Variability Modeling Approaches." In *Proceedings of the Sixth International Workshop on Variability Modelling of Software-Intensive Systems*, pp. 173–182.

Dannenberg, R. B., 1993. "Music Representation Issues, Techniques, and Systems." *Computer Music Journal* 17(3):20–30. 10.2307/3680940

Debussy, C., 1908. "Dieu qu'il la fait bon regarder. N°1 des 3 Chansons de Charles d'Orléans, à 4 voix mixtes a capella (manuscrit autographe)." Bibliothèque nationale de France. Available online at https://gallica.bnf.fr/ark:/12148/btv1b55007508k. Accessed August 2023.

Feichtinger, K., et al. 2021. "Guiding Feature Model Evolution by Lifting Code-Level Dependencies." *Journal of Computer Languages* 63:101034. 10.1016/j.cola.2021.101034

Ferber, S., J. Haag, and J. Savolainen. 2002. "Feature Interaction and Dependencies: Modeling Features for Reengineering a Legacy Product Line." In G. J. Chastek, ed. *Software Product Lines*. Berlin: Springer, pp. 235–256.

Field-Richards, H. S., 1993. "Cadenza: A Music Description Language." *Computer Music Journal* 17:60. 10.2307/3680545

Fischer, S., et al. 2018. "Predicting Higher Order Structural Feature Interactions in Variable Systems." In *Proceedings of the IEEE International Conference on Software Maintenance and Evolution*, pp. 252–263.

Fischer, S., et al. 2014. "Enhancing Clone-and-Own with Systematic Reuse for Developing Software Variants." In *Proceedings of the 30th IEEE International Conference on Software Maintenance and Evolution*, pp. 391–400.

Fischer, S., et al. 2016. "A Source Level Empirical Study of Features and Their Interactions in Variable Software." In *Proceedings of the IEEE 16th International Working Conference on Source Code Analysis and Manipulation*, pp. 197–206.

Font, J., et al. 2016. "Feature Location in Models through a Genetic Algorithm Driven by Information Retrieval Techniques." In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems*, pp. 272–282.

Fournier-S'niehotta, R., P. Rigaux, and N. Travers. 2016. "Is There a Data Model in Music Notation?" In *Proceedings of the International Conference on Technologies for Music Notation and Representation*, pp. 85–91.

Gould, E., 2011. *Behind Bars: The Definitive Guide to Music Notation*. London: Faber Music.

Grünbacher, P., 2022. "A Study on Variability for Multi-Device Rendering in Digital Music Publishing." In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*, pp. 6:1–6:9.

Grünbacher, P., R. Hanl, and L. Linsbauer. 2021. "Using Music Features for Managing Revisions and Variants in Music Notation Software." In *Proceedings of the International Conference on Technologies for Music Notation and Representation*, pp. 212–220.

Haugen, Ø., et al. 2008. "Adding Standardized Variability to Domain Specific Languages." In *Twelfth International Software Product Line Conference*, pp. 139–148.

Herold, K., 2020. "MusicDiff: A Diff Tool for MEI." In *Music Encoding Conference Proceedings*, pp. 59–66.

Hinterreiter, D., et al. 2020. "Supporting Feature-Oriented Evolution in Industrial Automation Product Lines." *Concurrent Engineering* 28(4): 265–279. 10.1177/1063293X20958930

Hinterreiter, D., et al. 2022. "Feature-Oriented Clone and Pull Operations for Distributed Development and Evolution." *Software Quality Journal* 30(4): 1039–1066. 10.1007/s11219-022-09591-4

Kang, K. C., et al. 1990. "Feature-Oriented Domain Analysis (FODA) Feasibility Study." Technical report CMU/SEI-90-TR-021. Carnegie-Mellon University Software Engineering Institute.
Kästner, C., S. Apel, and M. Kuhlemann. 2008. "Granularity in Software Product Lines." In *Proceedings of the 30th International Conference on Software Engineering*, pp. 311–320.
Kosar, T., S. Bohra, and M. Mernik. 2016. "Domain-Specific Languages: A Systematic Mapping Study." *Information and Software Technology* 71:77–91. 10.1016/j.infsof.2015.11.001
Lemberg, W., L. F. Moser, and U. Liska. 2020. "Music Engraving Conference 2020, Music University Mozarteum, Salzburg." Available online at https://gitlab.com/MusicEngravingConference/2020. Accessed January 2021.
Lepetit-Aimon, G., et al. 2016. "INScore Expressions to Compose Symbolic Scores." In *Proceedings of the International Conference on Technologies for Music Notation and Representation*, pp. 137–143.
LilyPond, 2024. "LilyPond: Notation Reference." Available online at https://lilypond.org/doc/v2.24/Documentation/notation.pdf. Accessed January 2024.
Linsbauer, L., et al. 2022. "Systematic Software Reuse with Automated Extraction and Composition for Clone-and-Own." In R. E. Lopez-Herrejon, J. Martinez, W. K. G. Assunção, T. Ziadi, M. Acher, and S. Vergilio, eds. *Handbook of Re-Engineering Software Intensive Systems into Software Product Lines*. Cham, Switzerland: Springer Nature, pp. 379–404.
Linsbauer, L., et al. 2021. "Concepts of Variation Control Systems." *Journal of Systems and Software* 171:110796. 10.1016/j.jss.2020.110796
Mernik, M., J. Heering, and A. M. Sloane. 2005. "When and How to Develop Domain-Specific Languages." *ACM Computing Surveys* 37(4):316–344. 10.1145/1118890.1118892
Michelon, G. K., et al. 2021. "The Life Cycle of Features in Highly-Configurable Software Systems Evolving in Space and Time." In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, pp. 2–15.
Michelon, G. K., et al. 2020. "Locating Feature Revisions in Software Systems Evolving in Space and Time." In *Proceedings of the Systems and Software Product Line Conference*, pp. 1–11.
Ratzenböck, M., et al. 2022. "Refactoring Product Lines by Replaying Version Histories." In *Proceedings of the 16th International Working Conference on Variability Modelling of Software-Intensive Systems*, pp. 8:1–8:10.
Sandberg, E., 2006. "Separating Input Language and Formatter in GNU LilyPond." Master's thesis, Uppsala University, Department of Information Technology.
Schulze, S., et al. 2013. "Does the Discipline of Preprocessor Annotations Matter?: A Controlled Experiment." In *Proceedings of the International Conference on Generative Programming: Concepts and Experiences*, pp. 65–74.
Schwägerl, F., and B. Westfechtel. 2019. "Integrated Revision and Variation Control for Evolving Model-Driven Software Product Lines." *Software and Systems Modeling* 18(6):3373–3420.
Stanciulescu, S., et al. 2016. "Concepts, Operations, and Feasibility of a Projection-Based Variation Control System." In *Proceedings of the International Conference on Software Maintenance and Evolution*, pp. 323–333.
Teich Geertinger, A., 2021. "Digital Encoding of Music Notation with MEI." In M. S. Bue and A. Rockenberger, eds. *Notated Music in the Digital Sphere. Possibilities and Limitations*, *Nota bene—Studies from the National Library of Norway*, vol. 15. Oslo: National Library of Norway, pp. 35–56.
van Deursen, A., P. Klint, and J. Visser. 2000. "Domain-Specific Languages: An Annotated Bibliography." *ACM SIGPLAN Notices* 35(6):26–36. 10.1145/352029.352035
Zave, P., 1993. "Feature Interactions and Formal Specifications in Telecommunications." *Computer* 26(8):20–28. 10.1109/2.223539
Zave, P., 2004. "FAQ Sheet on Feature Interaction." Available online at http://www.pamelazave.com/faq.html. Accessed 4 May 2022.