Orchestrating Change: An Artistic Representation of Software Evolution

Shane McIntosh*, Katie Legere†, and Ahmed E. Hassan*
*School of Computing, Queen's University, Canada
{mcintosh, ahmed}@cs.queensu.ca
†School of Music, Queen's University, Canada
legerek@queensu.ca

***Abstract*—Several visualization tools have been proposed to highlight interesting software evolution phenomena. These tools help practitioners to navigate large and complex software systems, and also support researchers in studying software evolution. However, little work has explored the use of sound in the context of software evolution. In this paper, we propose the use of musical interpretation to support exploration of software evolution data. In order to generate music inspired by software evolution, we use parameter-based sonification, i.e., a mapping of dataset characteristics to sound. Our approach yields musical scores that can be played synthetically or by a symphony orchestra. In designing our approach, we address three challenges: (1) the generated music must be aesthetically pleasing, (2) the generated music must accurately reflect the changes that have occurred, and (3) a small group of musicians must be able to impersonate a large development team. We assess the feasibility of our approach using historical data from Eclipse, which yields promising results.**

I. Introduction

Software systems need to react quickly to changes in a competitive, rapidly changing software market. Belady and Lehman first described this tendency in the context of commercial IBM software using *laws of software evolution* [1]. Godfrey and Tu found that Open Source Software (OSS) is not exempt from these laws, but rather that projects like the Linux kernel are subject to an accelerated version of them, growing at superlinear instead of linear rates [2]. Furthermore, Barahona *et al.* find that Debian, a compilation of more than 10,000 OSS projects, doubles in size every two years [3].

Prior work describes several visualization techniques for exploring historical software changes to better understand software evolution. For example, Wu *et al.* expand the sound spectrograph paradigm to a colour-coded visualization of historical code changes [4]. D'Ambros *et al.* propose an evolution radar to visualize co-change information at various levels of abstraction [5]. Wettel *et al.* visualize the complexity of a software system at a particular point in time using a city metaphor [6]. A sequence of city snapshots can be animated to illustrate the evolution process.

In this paper, we propose a novel approach to explore software evolution through musical interpretation of historical changes that are typically recorded in a Version Control System (VCS). Prior studies on the use of sound in software exploration motivate us to explore a musical interpretation of software evolution. For example, Hussein *et al.* show that *sonification*, i.e., the act of transforming data and relations into sounds [7], can be used to aid practitioners in program comprehension [8]. Furthermore, Boccuzzo and Gall use audio properties such as pitch, loudness, and sharpness to complement a visualization in navigating large software systems [9].

We use parameter-based sonification [7] of historical code changes to generate musical scores. We represent three properties of code changes in the generated scores: (1) software components are represented using *motifs*, i.e., short and memorable passages of music, (2) development periods are represented using *measures*, i.e, time-ordered sections of a musical score, and (3) developers are represented using *timbre*, i.e., the quality of sound created by a family of musical instruments.

Our approach generates musical scores that can be played synthetically or by a symphony orchestra. The generated musical scores may be of particular interest for practitioners because they can highlight phenomena that may require attention. For example, we use *dissonance*, i.e., combinations of musical notes that sound harsh or unpleasant, to signify rarely co-changing components.

In addition, sales of recorded performances of the generated music and ticket sales for live performances could generate revenue for OSS projects, who despite highly active volunteer communities [10, 11], need to defray the monetary costs of developing software. For instance, the Mozilla Foundation requires funding for: (1) web hosting, (2) testing infrastructure, and (3) personnel to perform tasks that volunteers are not interested in. The majority of Mozilla Foundation funding is provided by Google, who pays for the privilege of being the default web search option in the Mozilla Firefox web browser [12]. Many OSS projects generate revenue through the sales of project memorabilia, such as mugs and T-shirts.

Our approach takes liberty in its representation of software evolution when it will conflict with aesthetic qualities of the generated score. More specifically, our approach addresses the following three challenges:

(C1) Musicality
Motifs are sampled from well-known classical-era music in order to retain a natural, familiar feel.

(C2) Representativeness
While one can generate music that favours representativeness by treating periods of inactivity as measures of silence in the generated music, this detracts from

the score's musicality. Since we favour musicality over representativeness, we skip periods of inactivity when generating scores, yet we still ensure that the music varies across different periods of activities.

(C3) Cost-Effectiveness
We map the most active developers to the most common orchestral instruments. Infrequently contributing developers are allocated to percussive instruments, such that one musician can impersonate several developers – leading to more realistic personnel requirements in a live orchestral setting.

The contributions of the paper are twofold. First, we provide a general approach for sonifying historical code changes that addresses the three challenges listed above. Second, we assess our approach by performing a feasibility study using historical code changes from the Eclipse project. We have made the results of our feasibility study, i.e., the generated scores and synthetically-rendered performances available online [13].

Paper Organization

The remainder of the paper is organized as follows. Section II presents our approach and discusses how it addresses the three challenges listed above. Section III presents our feasibility study on the evolution of the Eclipse project. Section IV surveys related work. Finally, Section V draws conclusions and discusses potential avenues for future work.

II. Our Sonification Approach

Figure 1 provides an overview of our approach to sonifying historical code change data. In this section, we first describe how historical code change data are extracted from a VCS, and then discuss how properties of the code changes are mapped to a musical notation such that the three challenges are addressed.

A. Data Extraction

Software projects evolve through continual change in the source code and other artifacts. These changes are often collected in *patches* that show the differences between revisions of a file. Patches are typically logged in a VCS. File patches that are submitted together make up a *commit*. In addition to recording commits, VCSs record commit metadata, such as the author of the commit and the time when the commit occurred.

Our approach relies on commit metadata that is recorded in the VCS. Hence, prior to sonification, we first extract a listing of historical commits and their properties from the VCS. Next, our approach sonifies three properties of each commit:

(P1) The software component(s) impacted by the commit.
(P2) The date and time when the commit was recorded.
(P3) The developer who produced the commit.

Each commit property is mapped to a musical property in order to address one of the challenges outlined in Section I.

(P1-C1) Musicality: Mapping Components to Motifs

Definition Musicality is concerned with the tastefulness and melodious qualities of music.

Motivation Our approach aims to yield music that is pleasing enough to prompt donations from end users. Hence, the musicality of the generated scores is of paramount importance.

Approach We map software components to motifs. When a given component is modified, the mapped motif is performed. Since VCSs record changes to source code files, we map file changes to the software component that the file belongs to. As described below, the developer who made the modification determines the instrument that performs the motif.

Since a commit may change several components, several motifs may be played simultaneously. To avoid generating music that is full of jarring dissonance, we manually analyze component co-change rates prior to mapping them to motifs. We map consonant motifs to components that change together frequently to give the impression of stability and repose. Conversely, we map dissonant motifs to components that rarely change together to give the impression of unrest. Since we have no quantitative means of measuring consonance or dissonance, we rely on the musical training of the second author when performing this step. We recommend that musicians be consulted when performing this step.

The motifs we select are each one measure long and are sampled from Beethoven's Symphony No. 3, The Eroica. Selecting motifs from a Classical era symphony is advantageous for two reasons. First, orchestral musicians will likely be familiar with the types of motifs found in a Beethoven symphony, and hence, should be able to learn the generated music more easily than an unfamiliar arrangements of notes. Second, those without musical training will likely notice consonance and dissonance in classical motifs.

(P2-C2) Representativeness: Handling Periods of Inactivity

Definition Representativeness is concerned with how accurately the generated music reflects the code change activity of the historical period it is based upon.

Motivation While this is of less concern than musicality, we also aim to generate music that is representative of the studied periods of code change history to a large extent.

Approach In order to preserve the temporal properties of the historical code changes, we map development time periods to musical measures in the generated score. However, there are periods of inactivity that may disrupt the flow of the music if an inappropriate granularity is selected. We address these time period granularity concerns on a case-by-case basis. For example, a short development history of one month could reasonably produce one measure for each hour in the day, while a longer period of development history of one year may only produce one measure for each day.

The tempo of the piece must be carefully selected. A tempo that is too slow can also disrupt the flow of the music by prolonging the silent periods of inactivity. On the other hand, a tempo that is too fast will make it difficult to differentiate between motifs. We rely on the musical training of the second author when selecting a tempo that strikes a balance between silence minimization and motif differentiability.

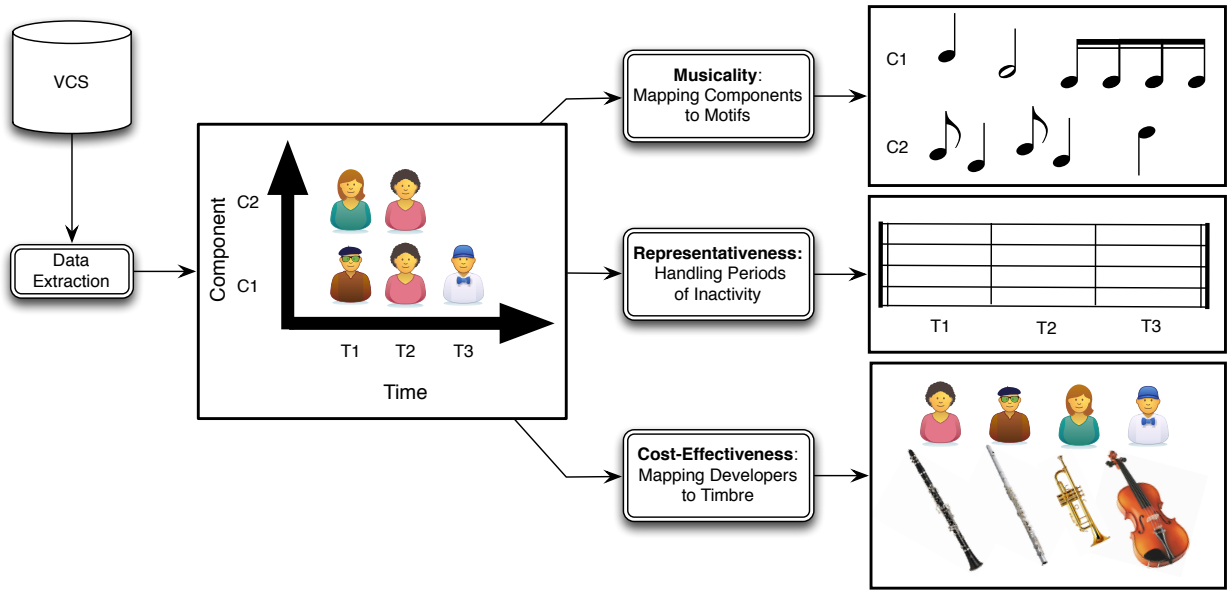

Fig. 1: Overview of our approach to sonify software evolution data.

(P3-C3) Cost-Effectiveness: Mapping Developers to Timbre

Definition Cost-effectiveness is concerned with the practicality of the generated music.

Motivation Since the number of available musicians may vary depending on the location of the recording studio or performance hall, the code change property that is mapped to timbre must adapt to fit the set of available musicians. In areas with few available musicians, a small number of musicians will need to impersonate a large number of developers.

Approach Several developers coordinate their efforts when working on large-scale projects. We consider each developer as a performer in a software development orchestra. As shown in Figure 1, each developer is mapped to a timbre. As concrete examples of timbre, we use instruments typically found in an orchestra, e.g., clarinet, flute, trumpet, and violin. Whenever a commit made by a particular developer is encountered in the history, they are considered to be performing and their mapped instrument is sounded. The motif that is played is determined by the components that have been modified as described above.

In order to optimize the mapping of developers to instruments, we first extract the list of unique developers and order them by contribution rate, i.e., the rate at which they produce source code changes. The instruments are also ordered from most to least common with percussion instruments appearing at the bottom of the list. We then map the most prolific developers with the most common musical instruments.

We perform this rank-based mapping of developers to musical instruments for two reasons. First, it is more likely that one can find a musician who plays the flute than one who plays the zither, so mapping developers with the highest contribution rates to the most commonly played musical instruments ensures that the final product requires a palette of commonly available musicians. Second, there are likely more developers participating in a software project than available musicians. In a modern orchestra, it is common for a small group of musicians to play several percussive instruments during a performance. Thus, mapping several infrequently contributing developers to percussion instruments helps to reduce the number of required musicians.

III. Eclipse Feasibility Study

We assess the feasibility of our approach by applying it to two periods of Eclipse development history. Table I provides an overview of the studied periods. We set the granularity of our approach to one measure per day, i.e., all of the development activity that occurs in one day is compressed into a single measure in the generated music. While the granularity level of days may seem too coarse, we find that it yields reasonable musical scores based on the length of the studied periods of development history.

A. Periods of Inactivity

We apply our approach to a foundational Eclipse development period (May 2001 - Oct 2002), as well as a later maintenance period (Jan 2005 - Dec 2005). There are intervals of rapid activity as well as intervals of inactivity in both of the studied periods. We apply two different approaches when handling intervals of inactivity:

(1) Treat inactivity as musical silence
In this more literal data interpretation, we map intervals of inactivity to rests in the score in order to preserve the temporal characteristics of the data.

(2) Ignore inactivity
Contrary to the first approach, we discard intervals of inactivity to produce a more contiguous piece of music.

In our opinion, the second approach yields more aesthetically pleasing results. However, we have produced musical scores using both approaches. We adopt approach 1 when

TABLE I: Overview of the feasibility study data

Period	Timeframe	Commits	Developers	Components
Foundational	May 2001 Oct 2002	261	14	9
Maintenance	Jan 2005 Dec 2005	315	24	27

generating the foundational period score, and approach 2 when generating the maintenance period score. The scores generated for the two periods, as well as synthetically rendered performances of them are available online [13].

B. Foundational Period Results

We apply our approach to the foundational Eclipse development period between May 2001 and October 2002. We set the tempo of the piece to 240 beats per minute. Since each measure is comprised of four beats, each measure lasts one second. Thus, the entire generated piece can be played in just over four minutes. Although the individual motifs are played quickly, they are still distinct enough for an astute listener to notice trends and gain insight into the evolution of the Eclipse project. In addition, since the longest period of inactivity is 15 days, the selected tempo ensures that 15 seconds is the maximum duration of silence in the generated score.

When listening to the synthesized music, the long periods of silence quickly become apparent. One can easily detect periods of little change (where few instruments are sounded), as well as periods of complete stagnation (where there is silence in the piece). Many of the periods of silence coincide with common holidays, e.g., there was little activity during the winter holiday season of 2001.

C. Maintenance Period Results

The lengthy periods of silence in the foundational development period results detract from the listening experience. Even after accelerating the tempo of the piece, the periods of silence tend to detract from the listening experience without adding much value. Hence, we produce a second piece using a highly active maintenance period with few periods of inactivity.

The synthesized music for this period is more intricate than that of the foundational period. Indeed, the two-note motif that sounds repeatedly beginning at 0:04 indicates that a single component is changed frequently. The fact that the pattern is sounded often by the clarinet indicates that a single developer is responsible for much of the change.

Consonance and dissonance also play a much bigger role in the maintenance piece than the foundational one, since co-change occurs more frequently. For example, the measures played between 1:40 and 1:51 are highly consonant, indicating the co-change that occurred represents a large proportion of the co-change for the components involved. On the other hand, the motifs sounded between 0:14 and 0:16 are dissonant, indicating that combination of co-changed components is rare.

IV. Related Work

Sonification has received some software engineering research attention as of late. Vickers and Alty built CAITLIN, a sonification system that maps Pascal programming constructs to motifs and can be used to "listen" to a program [14]. Vickers and Alty also show that CAITLIN can be useful for Pascal debugging tasks [15].

Prior work, such as that of Hussein *et al.* suggest that sonification can aid practitioners in program comprehension [8]. Indeed, Berman and Gallagher build and evaluate a sonification tool for assisting developers in understanding the architecture of Java applications [16]. Boccuzzo and Gall augment their `CocoViz` tool with auditory icons [9] and ambient sounds [17] to help practitioners to understand programs and how they evolve. Rather than providing software development tools, we focus on the artistic qualities of the generated music.

V. Conclusions & Future Work

Software tends to grow in terms of size and complexity as it ages [1]. Much research effort has been invested in visualization tools and techniques that aid in the software evolution process [4–6]. In this paper, we explore the use of parameter-based sonification as a means of producing a musical interpretation of software evolution data. Through a feasibility study on the evolution history of the Eclipse project, we note that:

- The rate of activity during time intervals can be noticed when inactive periods are mapped to musical rests, however the cohesiveness of the musical score suffers.
- Consonance and dissonance can highlight interesting phenomena (e.g., rarely co-changing components).

Future Work Several of the steps in our approach require manual intervention. For example, we manually analyze component co-change rates in order to map consonant and dissonant motifs appropriately. We plan to explore approaches to automate these sorts of steps in future work.

From a more general perspective, we are concerned with representing the collaborative software development experience artistically. We have primarily focused on music, however the visual arts may also provide an interesting avenue for future work. The field of visualization has produced several useful and representative renderings of software evolution data. However, little work has focused on the aesthetic qualities of these representations. Inspired by procedural generation techniques from computer graphics, we plan to generate software evolution landscapes algorithmically. Ultimately, we will pursue a hybrid approach that combines both visual art and music to represent software evolution data and create an immersive experience similar to that of a film.

Acknowledgments

This research was supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

References

[1] L. A. Belady and M. M. Lehman, "A model of large program development," *IBM Systems Journal*, vol. 15, no. 3, pp. 225–252, 1976.

[2] M. W. Godfrey and Q. Tu, "Evolution in Open Source Software: A Case Study," in *Proc. of the 8th Int'l Conf. on Software Maintenance (ICSM)*, 2000, pp. 131–142.

[3] J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. Amor, and D. German, "Macro-level software evolution: a case study of a large software compilation," *Empirical Software Engineering*, vol. 14, no. 3, pp. 262–285, 2009.

[4] J. Wu, R. C. Holt, and A. E. Hassan, "Exploring Software Evolution Using Spectrographs," in *Proc. of the 11th Working Conf. on Reverse Engineering (WCRE)*, 2004, pp. 80–89.

[5] M. D'Ambros, M. Lanza, and M. Lungu, "Visualizing Co-Change Information with the Evolution Radar," *Transactions on Software Engineering (TSE)*, vol. 35, no. 5, pp. 720–735, 2009.

[6] R. Wettel, M. Lanza, and R. Robbes, "Software Systems as Cities: A Controlled Experiment," in *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*, 2011, pp. 551–560.

[7] T. Hermann, "Taxonomy and definitions for sonification and auditory display," in *Proc. of the 14th Int'l Conf. on Auditory Display (ICAD)*, 2008, pp. 1–8.

[8] K. Hussein, E. Tilevich, I. I. Bukvic, and S. Kim, "Sonification Design Guidelines to Enhance Program Comprehension," in *Proc. of the 17th Int'l Conf. on Program Comprehension (ICPC)*, 2009, pp. 120–129.

[9] S. Boccuzzo and H. C. Gall, "Software Visualization with Audio Supported Cognitive Glyphs," in *Proc. of the 24th Int'l Conf. on Software Maintenance (ICSM)*, 2008, pp. 366–375.

[10] G. Robles, S. Dueñas, and J. M. Gonzalez-Barahona, "Corporate Involvement of Libre Software: Study of Presence in Debian Code over Time," in *Open Source Development, Adoption and Innovation*, ser. The International Federation for Information Processing (IFIP). Springer, 2007, vol. 234, pp. 121–132.

[11] D. Riehle, P. Riemer, C. Kolassa, and M. Schmidt, "Paid vs. Volunteer Work in Open Source," in *Proc. of the 47th Hawaii Int'l Conf. on System Science (HICSS)*, 2014, to appear.

[12] "Mozilla and Google Sign New Agreement for Default Search in Firefox," https://blog.mozilla.org/blog/2011/12/20/mozilla-and-google-sign-new-agreement-for-default...

[13] "Supplementary materials," http://sailhome.cs.queensu.ca/replication/Orchestrating_Change/.

[14] P. Vickers and J. L. Alty, "Using music to communicate computing information," *Interacting with Computers*, vol. 14, pp. 435–456, 2002.

[15] ——, "When bugs sing," *Interacting with Computers*, vol. 14, pp. 793–819, 2002.

[16] L. I. Berman and K. B. Gallagher, "Using Sound to Understand Software Architecture," in *Proc. of the 27th Int'l Conf. on Design of Communication (SIGDOC)*, 2009, pp. 127–134.

[17] S. Boccuzzo and H. C. Gall, "CocoViz with Ambient Audio Software Exploration," in *Proc. of the 31st Int'l Conf. on Software Engineering (ICSE)*, 2009, pp. 571–574.