Variator: A Creativity Support Tool for Music Composition

Avneesh Sarwate
Princeton University
Department of Computer Science
asarwate@princeton.edu

Rebecca Fiebrink
Princeton University
Department of Computer Science (also Music)
fiebrink@princeton.edu

ABSTRACT

The Variator is a compositional assistance tool that aims to let users quickly produce and experiment with variations on musical objects, such as chords, melodies, and chord progressions. The transformations performed by the Variator can range from standard counterpoint transformations (inversion, retrograde, transposition) to more complicated custom transformations, and the system is built to encourage the writing of custom transformations. This paper explores the design decisions involved in creating a compositional assistance tool, describes the Variator interface and a preliminary set of implemented transformation functions, analyzes the results of the evaluations of a prototype system, and lays out future plans for expanding upon that system, both as a stand-alone application and as the basis for an open source/collaborative community where users can implement and share their own transformation functions.

Keywords

Composition assistance tool, computer-aided composition, social composition.

1. INTRODUCTION

The motivation for Variator is to create a system that can help assist human creativity by attempting to tackle the problem of musical writer's block. Such a system should allow users to quickly insert, tinker with, and play the musical objects that they are working with in a relatively seamless manner. It should also support exploratory search and rich history-keeping, according to the guidelines for "creativity support tools" proposed by Shneiderman [9]. Variator attempts to provide an interface that allows users to quickly and painlessly search through "musical space" and keep track of their progress, so they do not lose good ideas once they find them. Furthermore, Variator aims to incorporate musical intelligence in order to generate new music not explicitly written by the user. The use of musical intelligence within the framework of a creativity support tool differentiates Variator from existing systems.

2. RELATED WORK

Existing popular notation tools and Digital Audio Workstations offer users a fairly quick and easy way to notate, modify, and play back music. However, none of these systems exhibit much "musical intelligence." Notation software packages incorporate some minimal musical knowledge, as they can exploit the basic relationships and rules that govern the diatonic key structure to expedite the process of entering notes into a score. For example, when entering music into a score where a specific key has been chosen, keyboard shortcuts exist to quickly add harmonies to existing notes. The software can also shift segments up or down along the staff, and it automatically groups notes according to the chosen time signature. However, these features mostly amount to minimal arithmetic based on simple and explicitly chosen patterns. The piano roll layout for most DAWs offers even less, simply offering a way to shift selected notes up or down on the piano.

On the other end, there are systems that exhibit a high degree of musical intelligence, but that do not allow a high degree of control over the type of music produced. The most famous example may be David Cope's *Experiments in Musical Intelligence,* which, after learning from a corpus of works, can produce entire pieces [4]. Cope's stated initial motivation for the project—a cure for "composers' block"—is similar to the motivation behind Variator. However, while Cope's system exhibits a very high degree of musical knowledge, it does not provide low-level control over the music produced. In contrast, Variator allows users to control the production of music at the level of individual melodies or chord progressions.

Another related system is Pachet's Continuator (from which Variator borrows its titular theme) [7]. The Continuator lies between the domains of probabilistic music generation systems and interactive music systems. The Continuator's real-time musical output is driven both by a Markov model learned from an offline musical corpus and by the real-time input of a collaborating human musician.

Other systems offering a mix of musical intelligence and user control also exist, but they are not easily usable by the average musician. OpenMusic is a visual interface to the Lisp programming language that functions as a powerful and flexible computer-assisted composition tool [1]. It allows users to build modular "patches" to process, display, and play musical data. It provides representations of musical objects such as chords and melodic "parts," and users can create their own patches to transform or generate these objects. Though very flexible, OpenMusic is quite a departure from the conventional interface of more popular music writing tools, and the logic behind visual programming could prove a deterrent to its use by non-programers. Rather than having to deal with the full abstraction of a programming language or learn to organize a graphical programming language, users need only understand the idea of a "function" and how to call one to use Variator.

Pachet's work on description-based design [8] suggests one possible approach for implementing musical transformations. Pachet's system uses machine learning to allow users to train a classifier to recognize a certain musical descriptor; it then generates variations on a melodic phrase that fit "more" or "less" with this learned descriptor. Learning-based generators could be implemented in Variator, but there are many possibilities for transformations based on explicit rules, stochastic decision-making, or iterative design to be implemented as well. Pachet also discusses the possibility of using social tagging systems to let a community of users classify melodies, and then using a learning algorithm to learn descriptors based off of these tags. A longer-term goal of Variator is to leverage such community-derived information in the creation of music transformation modules.

NIME'13, May 27-30, 2013, KAIST, Daejeon, Korea.

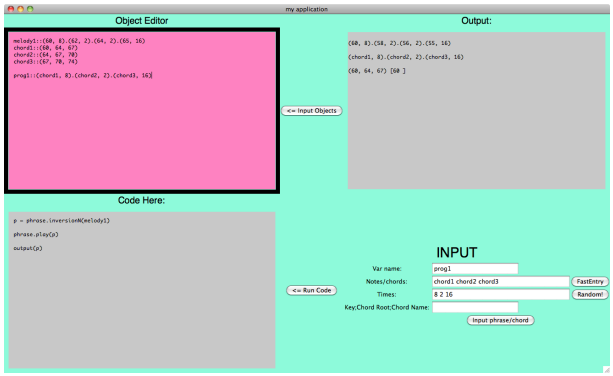

Figure 1. Variator interface: (clockwise from top left) object editor, output window, input, code editor

Music21 is a Python library that allows users to quickly query stores of music data for various types of information. It also implements representations of objects and melodic phrases, and operations on those phrases. It allows users to graph many different musical "quantities" of musical objects or a whole corpus, and it includes a limited number of functions for music generation, many based on Baroque counterpoint rules [2]. However, its implementation as a Python library could preclude non-programmers from quickly adapting it, and it does not include a user interface.

While previous systems have provided either musical intelligence or an interface that is accessible to a wide range of users, none have been particularly successful at combining the two. Variator aims to incorporate both of these features into a tool for computer-aided composition (CAC) that is accessible to non-programmers, and a platform that allows musicians to program their own transformations as easily as possible.

3. THE VARIATOR SYSTEM

Variator is implemented in Python and ChucK [10]. The GUI is built using TKInter, Python's interface to the TK GUI Tool. All of the transformation functions are implemented as functions in Python modules. Variator also allows the user to play back the musical objects created, and ChucK is used as the sound producing back-end. All communication between Python and ChucK is done using Open Sound Control.

3.1 Interface Elements

Variator provides an interface that allows the user to create and manipulate musical objects. Variator currently works with three different types of musical objects: melodic phrases, chords, and progressions. *Phrases* are single-voice melodies, represented as corresponding lists of notes and times. Phrases can also optionally have a string denoting their key. *Chords* are represented as a list of notes.

Chords also specify the note that is the root of the chord, set by default to the lowest note. Chords also have an optional string specifying the name of the chord. *Progressions* are chord progressions, and they are represented by parallel lists of chords and associated times. Currently, progressions can only be created from previously-created chords. The interface, whose whole layout is show in Figure 1, has four main components, described below.

3.1.1 Input

To create an object, users start at the input window, shown in Figure 2. All objects input to Variator must be given variable names. Object types need not be declared explicitly; they are inferred from the input data. Notes will be recognized as a chord, notes and times as a phrase, and chords and times as a progression. Notes are input as integer values corresponding to

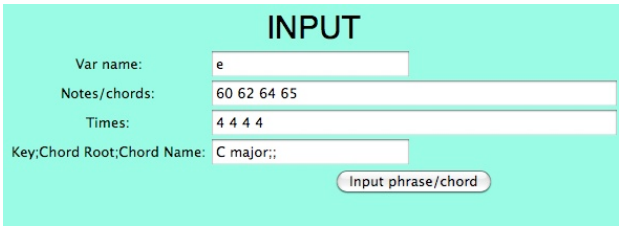

Figure 2. Variator object input area

Figure 3. Variator object editor

```
e = phrase.keyInvert(c, "C minor")

play(e)

output(e)
```

Figure 4. A simple code example. The "phrase" module contains a group of built in transformations. This example takes the existing phrase called "c", transforms it, plays it, and prints the string representation of "e" to the output

MIDI note numbers. Optional attributes, such as associating a key with a phrase, or a non-default root or a name for the chord, can be specified in the bottom input box.

3.1.2 Object Editor

Once an object has been input, it will be displayed in the object editor (Figure 3). The object editor allows users both to see all of the objects currently defined in Variator and to edit them. The editor window displays each object succinctly in text format, using a concise syntax (`varName:KeyString: objectRep-resentation`) for each object. Users can directly edit the text to modify objects. Users can save and reload objects as text.

3.1.3 Coding Area

Users execute transformations on objects in the coding area (Figure 4). All objects in Variator are implemented as Python objects, and all transformations are Python functions that take Python/Variator objects as arguments and return other Python/Variator objects. Transformations are executed in standard Python in the coding area as seen in Figure 3. New code can be immediately executed by pressing the "Run Code" button to the right of the coding area.

3.1.4 Output window

The output window is a simple output terminal. The `output()` command prints its argument(s) in the Variator output window in a text format that allows the user to save them for later use by copying them directly to the object editor window.

3.2 Transformations

The core novelty of Variator lies in its transformative approach to music generation. The transformations allow users to create variations of their own objects extremely rapidly, thus letting users traverse "musical space" very quickly. Each

transformation is implemented as a Python function that takes in a Variator/Python object as one of its arguments, and then produces a different Variator/Python object. Variator comes with several built-in transformations, and users can easily define their own transformations through code. Users can implement transformations as Python functions and add their code to the Variator source.

3.2.1 Built-In Transformations

In addition to standard counterpoint transformations (transposition, retrograde, inversion), Variator includes the following built-in transformations:

- Interpolations between two phrases: The transformation function takes two phrases and a floating-point value (0, 1) as arguments. The floating-point value determines whether the output will be more like the first phrase argument (values closer to 0), or the second (values closer to 1). It is implemented using the Levenshtein distance algorithm.
- Melodicizing a chord progression: The function takes a chord progression and a melody as arguments and inverts each chord in the progression to produce a new progression where the top notes of each chord follow the melody as closely as possible.
- Giving progressions a direction: This function takes a chord progression and re-voices the chords so that all the top notes either descend or ascend by the smallest possible intervals.
- Reevaluating rhythm: This function takes a phrase object and an importance ranking over the scale of the phrase. Then, based on the importance ranking, it redistributes the time values of the phrase, assigning the longer time values to the more "important" notes.
- Key-preserving transposition: This function transposes the notes in a phrase by scale degree rather than by absolute intervals, thus preserving the key of the original phrase.
- Key-preserving inversion: Rather than reversing the intervals between each pair of notes, this function reverses the change in scale degree between each pair of notes, thus preserving the key of the original phrase.
- Fitting a melody to a chord progression: This function takes a phrase, a chord progression, and a mix factor [0, 1]. For the fraction of notes specified by the mix factor, the function changes them to chord tones from the chord in the progression over which the note will be played.
- Generating random "noise" in a phrase: This function selects a random note in the scale and does one of several things: shifts the note up or down by 0–5 semitones, changes the duration of the note, adds a note within 5 semitones, removes the note, or changes the duration of the note. The relative probability with which it does these things is a parameter of the function.

3.2.2 Helper Objects

Variator comes with several "helper" objects and functions to help users build their own transformations. It contains:

- A function to change the key of a melody
- A function to change the mode of a melody (but preserve the tonic of the scale)
- A function for detecting the key of a given melodic sequence
- Intervallic definitions of the modes of the major scale
- An "importance ranking" for notes of the major scale
- A function that, given a melodic sequence and a chord progression, returns the segments of the sequence over each chord in the progression
- A function to determine the best triad over a given melodic sequence
- A function that fits a phrase to a given key

The helper objects are subjectively defined in terms of musical choices, and users may have different criteria for determining them (e.g., a user might choose to fit a chord to melody in a different way). They are included as templates for users' own definitions.

4. EVALUATION

We conducted a preliminary evaluation of Variator to assess the potential usefulness of the overall system, and to identify room for improvement in the user interface and the selection of built-in transformations. We evaluated the system with a group of four undergraduate students. All had some music experience, with three being active in songwriting or composition. Of the four, one had no coding experience, one had minimal coding experience (basic HTML familiarity), and two had moderate coding experience (introductory computer science classes).

The group was given a 20-minute tutorial demonstrating the system and some of its functions. Participants were then given the code to run on their own machines. They had one hour to experiment with the system, and they were encouraged to write music and to tinker with the code and write original functions.

Afterwards, participants filled out a written survey. A first set of questions asked them to about their overall likelihood to use the system, and their likelihood to use CAC tools in general. Participants were then asked to qualitatively describe their songwriting process, and more specifically, what "parts" of the song they generally tended to write before others (e.g., rhythms before melodies before chords, etc). A second component of the survey assessed the user interface using metrics proposed in the Creativity Support Index [3], asking for users' assessments of the flexibility and intuitiveness of the interface, as well as their perceived efficiency while using it. They were also asked for suggestions for features and improvements. Finally, participants were asked about their knowledge of music theory, experience with writing music, and programming experience.

After individual responses were collected, all participants were gathered for a group discussion on Variator's strengths, weaknesses, and potential directions for the future.

5. DISCUSSION

Overall, the evaluation showed two things: that users found the core concept intriguing and the interface somewhat inefficient. The user reports suggested that Shneiderman's two key features of Creativity Support Tools, rich history keeping and exploratory search support, are weak in Variator. Although history keeping exists, it requires that users actively manage history keeping, and the visual representation of the history does not quickly provide the desired information. Also, the programming approach to executing transformations renders the system all but unusable to those without coding experience.

The interface of Variator provides an improvement over that of the Continuator (at least in terms of music composition), by providing a mechanism for history keeping at all. Continuator, being a live-performance extender, has no such mechanism [at least, none that the authors of this paper could find in documentation]. Variator also offers the advantage that learning it does not require one to deal with a full programming language, as is necessary with OpenMusic and Music21. For Variator, a user must simply understand the idea of variables, functions, and function arguments to use all of the core features. One of the users taking part in the evaluation did not even have trivial experience with programming, but was able to pick up the previously mentioned concepts and begin using Variator in less than 10 minutes.

It was noted that the one user with no background in music theory seemed lost while experimenting in the system, and he remarked that he didn't know how to proceed without knowing how the transformations worked. With this in mind, it may

provide useful to divide further development of functions by demographics. Functions for users with little music experience could involve a high degree of intelligence, taking more of the decision making away from the user, but providing "normal" sounding music more often. Also, functions that randomly generated "melodically consistent" objects, or even a set of built-in phrases, chords and progressions, could allow novice musicians a chance to explore the transformations without the burden of creating the source objects. Functions aimed toward more serious composers could incorporate more randomness, as suggested, or involve transformations that are more systematic and combinable into intricate systems. Two of the users with more music theory experience suggested the inclusion of additional random generator functions, suggesting that this randomness could help overcome the initial problem of writer's block. Among the functions that users found particularly useful was the random "noise" function, which allowed the users to produce variations without having to think about how they would guide the variations produced. Users were also satisfied with the key-fitting function, which allowed them to experiment freely without worrying about having to manually make their new objects share keys with the rest.

Users also brought up the question of how non-coders would use the interface. Changes to the interface could allow non-programmers to select transformations and arguments via menus, but as a result, some of the flexibility of programming would be lost. The choice to have a coding interface was made to maximize flexibility during the development of Variator. However, having multiple, selectable interfaces could solve this problem. Users found the ability to directly edit saved objects in the object editor helpful, but they suggested that the ability to view objects in standard musical notation would greatly improve efficiency. There is also the problem that non-coders would not be able to design their own transformations. One evaluation participant suggested that, if a community of transformation makers were to form, non-coding users of the system could post requests of transformations to be made by those who can program. All users said that they would be interested in using Variator's musical intelligence to experiment in their own composition.

6. FUTURE WORK

6.1 Interface Extensions

We plan to explore ways to allow users to define functions within the graphical interface itself. This would both give users the ability to prototype faster during function development, and make the interface more expressive by allowing for immediate customization of the tools available to the user.

More long-term plans for Variator include integration with notation software or DAWs. Variator could be much more closely tied to the compositional process if it were integrated with the tools musicians use most often to compose. In particular, allowing copy/pasting of musical objects between other software and Variator could virtually eliminate a huge source of time overhead in its use.

6.2 Improving Transformation Tools

In order to facilitate the writing of functions by users, it would be worthwhile to include more "low-level" objects and functions based on more rigorous music theory research. Functions that measured various definitions of distance between different types of objects, as well as different measures of "affinity" between objects would serve as useful building blocks. Also, intervallic definitions for keys not generally found in western music, as well as ways to "convert" melodies to keys with different numbers of notes (e.g., hexatonic to pentatonic) could provide users with a means of exploring new soundscapes with less time spent learning the requisite theory.

6.3 Sharing Transformations

Creating a forum where users could share their transformation functions could greatly expand the use of the system by allowing users access to tools that they could not produce themselves. A similar community exists based around the products of guitar software manufacturer Line 6 [5]. Line 6 specializes in the digital modeling of guitar amplifiers and effects, and has created a large online community where users can share their presets with others. A community for Variator "patches," however, could prove even more useful. Allowing users to share patches could allow those users with minimal music theory knowledge to utilize transformations created by more proficient theorists. Also, allowing users to share patches could greatly increase the rate at which users generate their own transformations, as they could more easily learn the different ways to attack problems from reading others' code, and could modify existing patches to reach their specific goals instead of starting from scratch.

Creating a means to share functions could also create a new model for collaborative composition. Teams of users could work together on more ambitious projects, such as fitness functions for hard-to-define adjectives like "funkiness" or "jazziness".

7. CONCLUSION

On the continuum of "interactivity" to "control" as defined by Lippe [6], the Variator seems to fall very far to the side of control. The lack of published research on systems offering the same level of explicit low-level control suggests that there may be a niche among composers for such a tool. Though users in our evaluation found some elements of the Variator interface inefficient, the overwhelming interest in the core concept shows the Variator project to be a promising line of research.

8. REFERENCES

[1] Agon, C., Assayag, G., Bresson, J. 2005. OpenMusic 5: A cross-platform release of the computer-assisted composition environment. *Proc. 10th Brazilian Symposium on Computer Music.*

[2] Ariza, C., Cuthbert, M. 2000. Music21: A toolkit for computer-aided musicology and symbolic music data. *Proc. ISMIR.*

[3] Carroll, E., Latulipe, C., Fung R., and Terry, M. 2009. Creativity factor evaluation: Towards a standardized survey metric for creativity support. *Proc. ACM Creativity & Cognition*, 127–136.

[4] Cope, David. Experiments in musical intelligence. http://artsites.ucsc.edu/faculty/cope/experiments.htm

[5] Line 6 CustomTone. http://line6.com/customtone/

[6] Lippe, C. 2002. Real-time interaction among composers, performers, and computer systems. *Notes of the Information Processing Society of Japan, 123,* 1–6.

[7] Pachet. Francois. 2003. The Continuator: Musical interaction with style. *Journal of New Music Research*, 32(3): 333–341.

[8] Pachet, Francois. 2009. Description-based design of melodies. *Computer Music Journal* 33(4): 56–68.

[9] Shneiderman, B. 2007. Creativity support tools: Accelerating discovery and innovation. *Communications of the ACM* 50(12): 20–32.

[10] Wang, Ge. 2008. The ChucK audio programming language: A strongly-timed and on-the-fly environ/mentality. PhD thesis, Princeton University, Princeton, NJ, USA.