Procedural Orchestra: Virtual Music Generation Using Interactive Modularization

Rane Squires
Graduate School of Arts and Sciences
Computer Science Department
College of William and Mary
Williamsburg, Virginia 23187-8795
Email: rsquires@email.wm.edu

Abstract—**In the area of musical composition, computer music and virtual tool-assisted composition has continued to grow in popularity over the last five decades. Nowadays, several completely automatic music generation algorithms exist that allow composers to quickly experiment and interactively compose with software. Utilizing both procedural and machine-learning methodologies, these programs range from extremely simple to rather sophisticated. In this paper, we introduce a novel paradigm for procedural music generation that focuses on the concept of "interactive modularization". This paradigm treats each sequence of notes as an individual virtual musician, which composes its portion of the piece independently from the others. Each module can act cooperatively or antagonistically with the others, allowing for an extremely wide range of possible combinations. This paradigm offers a unique composition space where the human composer has a degree of separation from the generator, by only defining the modules and most broad variables such as piece length. Our method offers a new perspective on interactive composition with many areas for possible growth in the future.**

I. Introduction

Since the 1970s, composers have been experimenting with involving computers in their musical process to various degrees. From early trackers to modern Digital Audio Workstations, commonly referred to as DAWs (Figure 1), computer music composition has become a huge area in music. Not only do DAWs allow composers to iterate quickly on ideas and get immediate audio feedback on their compositions, they have also allowed for entirely new genres and musical forms to emerge. By making computer music accessible to nearly anyone with a computer, DAWs have expanded synthesized and computer music from an esoteric medium dominated by electrical engineering to a widely popular and developed area. However, manual composition using DAWs is not the only area in which virtually-assisted composition has developed. Automated composition has also grown into a unique area of music creation.

Published in 1987, David Cope's "Experiments in Musical Intelligence" [1] is a paper which introduced one of the earliest procedural music generators: EMI, also known as Emmy. Emmy generated music based on a learned "style", which was produced by inputting and analyzing pieces of music by a certain composer. This combination of procedural and machine-learned paradigms produced pieces that were able to cause non-expert listeners to believe that the generated pieces were composed by long-dead classical composers. Emmy caused a tremendous amount of uproar in the musical community, with advocates on both sides suddenly feeling as though they had to defend their medium. As time went on, however, procedural music generators have become normalized and some of today's composers choose to utilize them to assist their own musical process.

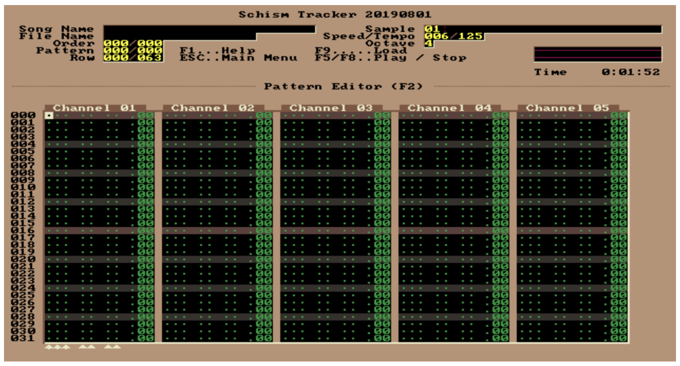
While modern music generation programs range from the extremely simple to extremely complicated, most are designed to serve a specific purpose. Some are designed to have certain effects on players of video games [2], some are designed to be easy to use by anyone [3], and some are designed to approximate or even replicate certain composition styles [4]. As far as we were able to determine, none of these existing programs have followed an interactive, modular paradigm, whereby a user would input parameters for the modules, rather than the piece itself. Our method of "Interactive Modularization" focuses on this paradigm, driving the composition process with the modules designed by the user and the modules' interactions with each other. In this way, the composition process is a cooperative effort between the user and the modules, as though the modules were virtual instrumentalists.

The rest of the paper is organized as follows: in Section III we will discuss the wide range of related work in the area of music generation. In Section IV we will explain the methodology of our developed solution. In Section V we will discuss the limitation of the current implementation of the solution, and the avenues of future work that can be explored. Finally, in Section VI we will conclude the paper.

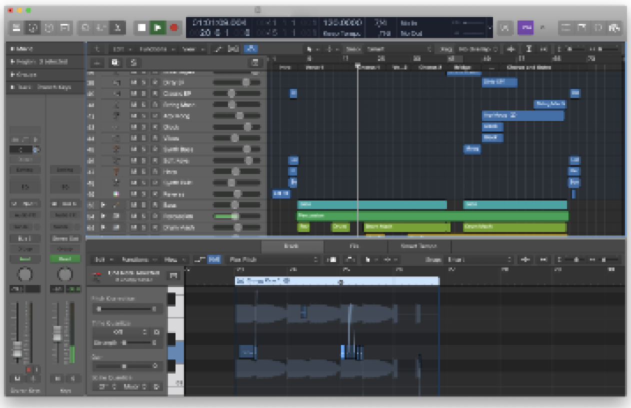
II. Related Work

A. Music Theory

A lot of research has been done on music theory as it relates to the computing domain. Specific research on how to generate and evaluate elements of musical theory with regards to various methodologies is itself a wide area. Researchers have attempted to isolate melodic structure in explicit and mathematical ways using pitch [5], intervals [6], or contours [7].Many publications have also been produced on the generation of harmony, focusing on areas such as chord progressions and strict rule sets like the Bach chorale [8] and rhythm [9].



(a) Tracker



(b) Digital Audio Workstation

Fig. 1: Screenshots of a tracker (left) and a DAW (right). a) Schism Tracker, a reproduction of the MS-DOS Impulse Tracker, b) Logic Pro X, a DAW developed by Apple inc. for their Mac products.

Developing a greater understanding of the underlying principles of music theory from a mathematical perspective continues to be an open problem in computer music. Meta analysis such as that from Herremans et al. 2018 [10] helps us to understand the full scope of the research that has been done so far. While this scope can be extremely large and difficult to grasp fully for either a composer or computer scientist, a basic understanding of each is sufficient to fully understand most music generator implementations.

B. Procedural Music Generators

Many kinds of Procedural Music Generation have been developed over the years. One of the very first methods of procedural music generation created by Pinkerton in 1956 [11] utilized Markov Chains. Markov Chain-based methods continued to be extremely popular for a long time. Methods as recent as Herremans and Chew 2016 [12] iterate on Markov-based optimization solutions. Other methods include those based on generative grammars [13], and genetic algorithms [14]

C. Machine Learning Music Generators

As machine learning has become more popular, so too has music generation utilizing machine learning approaches. As early as the work by Todd and Loy in 1989 [15], recurrent neural networks were used to generate musical pieces. More recent works such as Yang et al. 2017 [16] utilize approaches such as Generative Adversarial Networks. For a comprehensive overview of the machine learning approaches that have been used in music generation, see Briot 2020 [17].

III. Methodology

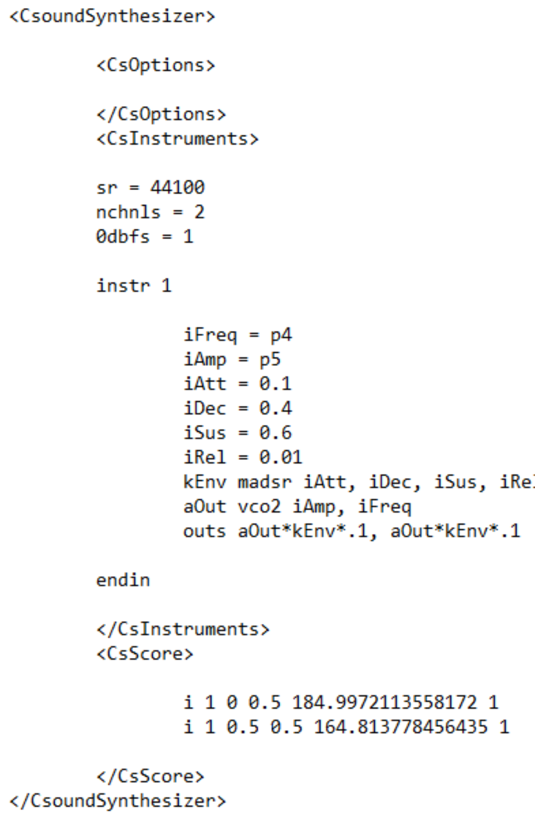
A. CSound

For the base of our solution, we used the open-source sound generation language CSound. CSound is a compiled language designed specifically for the generation of sounds. It has been in development since 1985, and over the many years it has become well featured and efficient. By using CSound, we were able to use a dedicated system that allowed us to bypass much of the complication of generating sounds using other languages' libraries. Additionally, since CSound is an explicit language, we were able to generate compilable CSound code externally and then compile the code into the desired audio file.

We generated our CSound files using custom Python code by directly generating the desired CSound code as a long string and writing the string to a text file. This allowed us to easily convert input to the necessary information needed by CSound, and also provided an easy-to-use platform for developing the procedural note generation algorithm we would use. Our python framework first generates a simple file containing the generated note sequences, then uses that file to generate the final CSound file that would compile into the output audio file. The python script then calls the CSound compiler itself, so that the only file that needs to be run to generate an audio file is the python script itself. After the python script is run, an output audio file in the Waveform Audio File Format (WAV) is generated.

B. Python Framework

In order to generate the audio file in a way that facilitates modular interactions, each module's note sequence is generated both independently and in a linear fashion. Each module is defined by a python class, that is generated by the input to the python script, and by a CSound "instrument", which is generated by the script. During the preparation phase of the script, all the necessary module class instances are generated and then passed to another class's method which handles all of the note sequence generation. This method iterates over each of the passed module instances and generates each individual note sequence, which is then appended to the string to be written to the note sequence file.



```
<CsoundSynthesizer>

        <CsOptions>

        </CsOptions>
        <CsInstruments>

        sr = 44100
        nchnls = 2
        0dbfs = 1

        instr 1

                iFreq = p4
                iAmp = p5
                iAtt = 0.1
                iDec = 0.4
                iSus = 0.6
                iRel = 0.01
                kEnv madsr iAtt, iDec, iSus, iRel
                aOut vco2 iAmp, iFreq
                outs aOut*kEnv*.1, aOut*kEnv*.1

        endin

        </CsInstruments>
        <CsScore>

                i 1 0 0.5 184.9972113558172 1
                i 1 0.5 0.5 164.813778456435 1

        </CsScore>
</CsoundSynthesizer>
```

Fig. 2: A Simple CSound file generated by our program. For our purposes, only the CsInstruments and CsScore headers are used. In the former, only one instrument is defined. In the latter, there is one second of music wherein two notes are played. After 'i', the instrument id, start time, end time, and pitch of the note are shown.

Each module instance is defined by a set of several input parameters, all of which are used in the note sequence generation process, either to select a note independently or to affect the module's interactions with the other modules. The former category of parameters includes the minimum and maximum note value the module is allowed to play, as defined by midi integer note values from 0 to 128, the starting note, the musical mode, and three floating point numbers which define the module's chance to go up in pitch, the module's chance to go down in pitch, and a modifier which influences the module's chance to continue to up or down in a sequence. The parameters which specifically affect modular interactions are two boolean values which determine if the module follows the previously computed module's note durations or follows the notes harmonically. The second of these also has an associated integer that determines the type of harmonization the module performs.

The module class also uses the input mode and starting note to calculate which notes between the minimum and maximum are in key. It uses the starting note as the base note for the key, and by using the distance between successive notes in the given mode, it iterates forward and backward from the starting note to create a list of all the notes which are in key for the module. This list of in-key notes is the set of notes which are available to the note generation algorithm to select. This limitation ensures that the generated sequence of notes sound in-key, which is a basic guarantee of sounding pleasant together. Due to this limitation, any future module that wishes to harmonize with a previous one must be in the same key, and therefore have the same mode and the same starting note value (meaning they must both start on C, for example).

C. Note Sequence Generation

In order to generate a module's note sequence, the generator performs different operations depending on the presence of defined interactions with the previously generated module. We will begin by describing the process for an entirely independent module, and then we will explain the module interactions. When the outermost loop of the generator reaches a given module, it sets an integer variable representing the current note sequence to zero. This sequence variable describes how many consecutive times the module has performed the same action in a row, either increasing or decreasing in pitch. Once the sequence variable has been prepared, the generator begins to loop until the sequence of notes has the desired duration.

Each time the generator selects a note, it starts by generating a random number between zero and one. This value is then compared to the percent chance of switching its action, decreasing if the sequence variable shows that it has been increasing or increasing if the sequence variable shows that it has been decreasing, modified by the Sequence Modifier variable raised to the power of the length of the sequence. In this way, the more times the module chooses to perform the same action. the less likely it will be to perform that action again unless the sequence variable is chosen to be exactly one. If the random value is greater than the combined sum of both the chance to increase and decrease, it will repeat the previous note. When the generator chooses to switch or repeat actions, it will randomly select the number of notes in the key to increase or decrease. The range of key-note distances is hard set as one to three in our implementation. If the generator repeats the previous action the sequence value will be incremented by one. If the generator switches actions the sequence value will be reset to one and indicate the new previous action. If the new note would be greater than the maximum or less than the minimum allowed value, then the relevant minimum or maximum is instead selected, and the sequence value is set significantly high (15 in our implementation) so that the generator will more than likely choose to move away from the border on the next note.

Once the note has been selected, the generator will pick a duration for the new note measure by the number of beats it will be held. In our implementation, we allowed for the generator to select half a beat (eighth note), one beat (quarter note), two beats (half note), and four beats (whole note). We did not allow the generator to create dotted notes or any kind of cut time to ensure that the resulting piece would sound

Variable	Type	Usable Values
Minimum Note	Int	Between 1 and 128
Maximum Note	Int	Between 1 and 128
Starting Note	Int	Between Minimum Note and Maximum Note
Mode	Enum	Any Modern Western Mode inc. ”Major”, ”Minor”
Chance to Increase	Float	Between 0 and 1
Chance to Decrease	Float	Between 0 and 1
Sequence Modifier	Float	Between 1 and +Inf., Usually Between 1 and 2
Follow Previous Rhythm	Boolean	True, False
Follow Previous Harmonically	Boolean	True, False
Harmonization Mode	Int	0, 1

Fig. 3: The input parameters used to define a module in the python script. Chance to Increase and Chance to Decrease must not add up to greater than 1, as they are percentage chances used together.

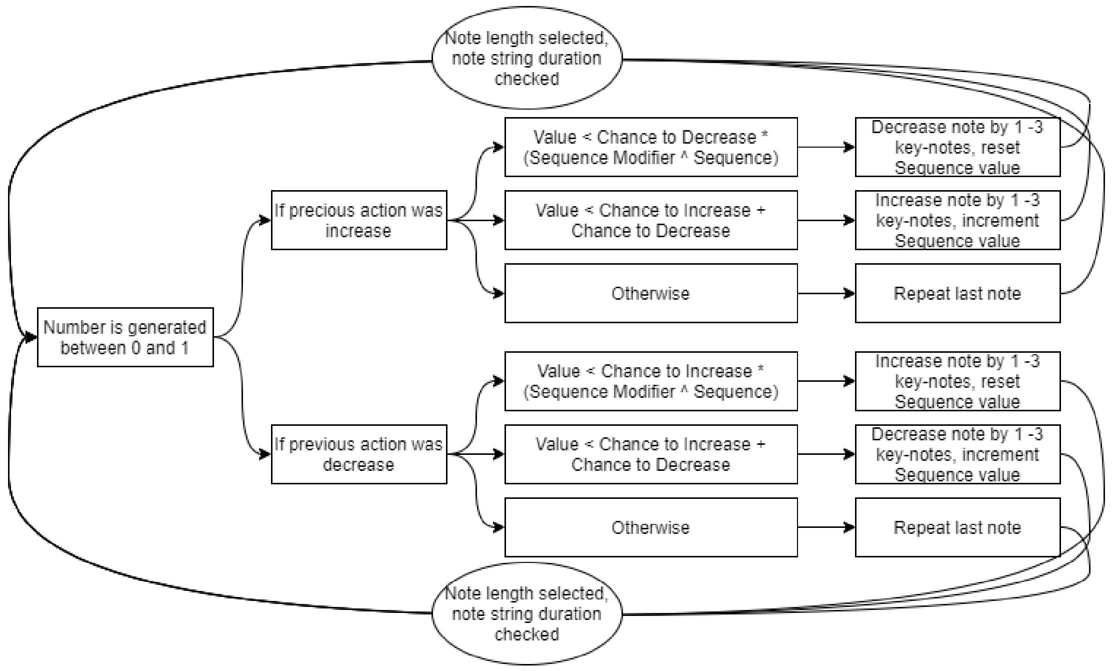

Note length selected, note string duration checked
Number is generated between 0 and 1
If precious action was increase
Value < Chance to Decrease * (Sequence Modifier ^ Sequence)
Value < Chance to Increase + Chance to Decrease
Otherwise
Decrease note by 1 -3 key-notes, reset Sequence value
Increase note by 1 -3 key-notes, increment Sequence value
Repeat last note
If previous action was decrease
Value < Chance to Increase * (Sequence Modifier ^ Sequence)
Value < Chance to Increase + Chance to Decrease
Otherwise
Increase note by 1 -3 key-notes, reset Sequence value
Decrease note by 1 -3 key-notes, increment Sequence value
Repeat last note
Note length selected, note string duration checked

Fig. 4: A flowchart showing the note selection process for an independent module.

“in-time” with a basic 4/4 time signature. For the independent module, these durations are randomly selected from an equally weighted distribution. If the selected duration would cause the total length of the note sequence to be greater than the desired duration, a new duration is randomly selected until the new total duration would be less than or equal to the desired length for the piece. Once the duration and note have been determined they are added to the string in order to be written to the note sequence file, and the selected note, duration, and cumulative duration are appended to lists to be used by the next module if it has specified interactions.

D. Module Interactions

In our implementation, there are two major kinds of module interactions: rhythmic following and harmonization. For rhythmic following, the note selection process occurs as normal for an independent module, but the duration selection process is changed. Instead of selecting the duration randomly, the duration is simply copied from the sequential list of note durations from the previous module. Because of this change, the new module plays notes selected separately from the previous module, but they will be played at the same time and for the same duration. Since the sum of all the durations must always be equal to the specified piece length, there will always be just enough durations for the new module and no more. The new module will play simultaneously with the previous module for the entire piece.

The harmonization process is significantly more complicated than rhythmic following. In our implementation, we allow for strictly-lower chordal harmonization. In this method, the second module will always generate a note that is lower in pitch than the note selected by the previous module, and it will select a note that has the same value as a note a third, fifth, octave, or unison away. To accomplish this, the generator creates a merged key for both modules, in order to allow it to know which notes can be played by either module and to calculate the distance in key-notes between selected notes. This merged key is simply a union between the keys for both of the modules. It then finds the distance between the note previously selected by the generator and the note that the previous module will be playing at the same time, using the list of selected notes and cumulative durations saved by the previous module.

Once the generator has determined the distance between the previous note of the current module and the current note of the previous module, it will choose whether to move toward the previous module's note (chance to increase) or away from it (chance to decrease). It does this identically to the independent module's method for choosing whether to increase or decrease in pitch. Once it has selected an action, it will move in the desired direction until it reaches a note with a permitted distance from the previous module's note. In order to ensure the note actually moves closer or farther to the previous module's note and not just up or down in pitch, the generator compares the distance to the one used in the previous note's generation and ensures that it has increased or decreased appropriately. If the desired action would cause the note to exceed the defined bound for the module, the generator selects the legal value closest to the bound, and sets the sequence value as an independent module would in order to encourage the generator to switch actions on the next iteration.

It is possible to combine rhythmic following and harmonization in the same module. In this case, the second module would play a note that forms a basic chord with the note played by the first module at the same time and duration. This creates a constant "chordal instrument" effect. It is also possible to chain rhythmic following or harmonization any number of times. For rhythmic following, each successive module would have the same note durations as all the previous modules with rhythmic following and the module immediately before this group. For harmonization, each successive module harmonizes with the module directly before it. However, due to the key-note distance we selected for the harmonization, this will result in selecting notes that are also legal harmonizations with the previous modules with harmonization and the module immediately before that group.

E. Final CSound File Generation

Once all of the module's note sequences have been generated and saved to the note sequence file, the generator can create the final CSound file that will be compiled into the resulting audio file. First it creates the CsInstruments section by iterating over each of the modules again, generating the necessary code for each module as a CSound instrument. It will then generate the CsScore section using the note sequence file. Since CSound requires the score to represent each note in terms of its starting time, ending time, and frequency, this requires some computation. To determine the start tiem and end time, it uses the generated duration in beats and a provided piece speed in Beats-Per-Minute (BPM). By dividing the BPM by sixty and multiplying by the selected number of beats, the length of the note in seconds can be obtained. The generator tracks the current time of the piece for the start time, and simply adds the length on the note in seconds to it to obtain the end time.

To provide the frequency on the desired note, it is necessary to convert the note number to its associated frequency. Fortunately, this is a relatively simple numerical process that can be performed quickly using the Hertz value of A4 (440) as a reference point:

$$Freq = (440/32) * 2^{((notenumber-9)/12)} \tag{1}$$

Once the frequency has been obtained, it can be easily inserted into the string in order to be written to the final CSound file. Once the generator has iterated over the entirety of the note sequence file, it can add the closing lines of the CSound file to the string and write it to the final file. Finally, the python script calls the CSound compiler with this file and an audio file is generated.

IV. Results

Our solution is able to generate audio files of any length with any number of input modules. Each module provides a sequence of notes where one note is played at a time for the entirety of the piece. These sequences of notes share the same tempo, time signature, and volume, but may differ entirely in rhythm and selected pitches. Within each module the notes are guaranteed to be within a key defined by the modules starting notes and a given mode, and are bound above and below by given note values. It is possible to both match modules rhythmically, and use one method of harmonization to match modules harmonically. In this way, it is possible to generate unique pieces of music that have the potential to sound extremely musical and sometimes pleasant.

Due to our focus on the generative algorithm, only one type of sound is used by the generator for the pieces, which is a Voltage Controlled Oscillator (VCO) Sawtooth wave. This means that all the modules have the same quality of sound, and the use of the sawtooth wave can result in a very "buzzy" sound to the pieces. This selection is hard-coded into our implementation. It can be manually changed, but the implementation does not allow for the waveform to be selected by input. Despite this limitation, the use of limited music theory has allowed us to generate interesting pieces using the generator and even pieces with modules defined at random by a human have interesting and somewhat musical qualities to them. This demonstrates the enormous potential of the Interactive Modularization system to both allow for a unique interactive composition experience and to create pieces with a wide variety of compositional qualities.

A. IM1 Rhythmic Following Demonstration

In this piece, which can be found attached to this paper or at https://github.com/KennedyQ/proceduralorchestra, we demonstrate the effect of rhythmic following using two modules. By listening to this piece, it should be apparent that both modules play all of their notes in sync and for the same duration. The modules in this piece have the same key, but do not have harmonization. The selected notes should sound somewhat separate even though they are playing at the same time. This piece also illustrates the basic cadencing our implementation has, where the modules return to their starting notes at the end of the piece. This will cause each module to play the base of its key's I chord, which is the most common way to end a phrase or piece.

B. IM2 Harmonization Demonstration

In this piece, we demonstrate the effect of harmonization without rhythmic following. We again use only two modules, which due to the limitations of our harmonization method are in the same key. While some of the combinations of notes sound more discordant than others due to the nature of the mode and key, the modules sound like they are working together rather than independently. The sounds produced by harmonization tend to sound more like they are creating one complete sound rather than two separate sounds being played at the same time, even when the modules are playing notes at different times. The modules do play some notes simultaneously and for the same duration at a few points in the piece, but this is by coincidence and not by design. A more robust rhythmic following system would allow the module to choose when and when not to do this.

C. IM3 Complete Piece

In this piece we demonstrate some of the potential of our solution. We constructed a piece which is performed by four modules, which follow many of the standards of the "four voice" style. In this way, we are able to construct a composition which has many of the elements of a classical choral piece. All of the modules sound as though they are forming a complex sound, while having some element of independence afforded by the lack of rhythmic following. The module which has the role of "leading voice" in the sound is determined by the randomly selected series of notes and changes as the piece progresses, despite the fact the module with the highest pitch range is the base for the harmonization. This piece is an excellent example for our implementation, as it demonstrates both the capabilities it has now and the limitations it could be designed to overcome.

V. Limitations and Future Work

A. Sound Limitations

As mentioned in the previous section, only one sound can be selected for all of the modules. In a more developed piece, the ability to select a different sound for each module would allow for more diverse sounding pieces. Additionally, the sounds that we can select are all simple waveforms. Modern virtual instruments are much more complex than simple waveforms and often have many variables. Modifying our CSound code and generator to handle these levels of sound variation would have involved a significant departure from the focus of our work, which was the note generation process. However, allowing for complex and highly defined sounds for each module would not only provide the generator with the ability to generate much more complex sounding pieces, but it would also open new avenues for modular interactions.

B. Music Theory Limitations

Our note generation algorithm follows a very simple key-based approach to harmonic note selection. Using only the notes in one key guarantees a level of tonicity, but a great many compositions utilize notes outside of the key in various ways. Many compositions utilize note "jumps" of a much greater size than we have allowed in our generation algorithm. Limiting the notes that each module can select and the distance between successive notes allows us to guarantee that the resulting piece has properties that are known to sound pleasant from a music theory perspective. However, it does somewhat limit the range of pieces that can be generated.

Additionally, our solution only offers one harmonization method at this time. While we can provide guaranteed harmonic interaction, our lower-pitch standard chordal harmonization method puts some restrictions on both the supplied input and the resulting pieces. For the input, it requires that for a module to follow the previous module harmonically it must have the same key. It also requires that the second module be able to be lower in pitch than the first module. If the second module has a higher maximum or minimum note value than the first, the range of allowed values will be considerably restricted. For the resulting piece, this method only allows one kind of chord, a standard I-III-V chord. There are many other types of chords, which are more complex, that are also often used in compositions.

C. Module Limitations

Currently, our solution only provides two methods for modules to interact with each other. While these interaction methods provide an excellent example of the potential of Interactive Modularization, they are simple. These interactions allow for modules to make decisions based on the selected note values or durations of previous modules, but they do not allow modules to make decisions based on the existence or properties of other modules. Additionally, both of these interactions are absolute and cooperative. Because they are absolute, once a module has an interactive property, they are guaranteed to follow that property and to follow it for the entire piece. Also, while cooperative interactions offer more guarantees for the musicality of a resulting piece, antagonistic interactions are implied by the attempt to give modules "personality" and could result in a wide variety of interesting compositions.

D. Future Directions

In the future we would like to continue to develop Interactive Modularization to allow for the possibility of more fully-featured and complex compositions. Integrating more complex sounds and Virtual Studio Technologies (VSTs) into the CSound and module definitions would open up an extremely wide variety of pieces and possible modular interactions. Sound variables like envelope, oscillators, and effects, such as reverb or delay, could be set or reacted to by the various modules. We would also like to add the ability for modules to have conditional, variable, and antagonistic interactions to give them more variety and more of a sense of independence while interacting with each other. Developing a user interface for the generator that would allow users to more easily create pieces and modules, and save modules for mix-and-match type scenarios would allow easier use of the generator, and would encourage experimentation with various modules and modular properties in a rapid-iteration paradigm.

VI. Conclusion

In this paper we introduced the Virtual Orchestra, a procedural music generation algorithm that utilizes Interactive Modularization to create a unique cooperative composition experience. We explained the generation algorithm for each module, as well as the interactions the modules can have with each other. We discussed the kind of pieces that can be created with the Virtual Orchestra and their limitations, and we explored the surface of the future directions that can be taken with modular music generation. In the future we hope that Interactive Modularization can become a highly-featured composition experience that is both enjoyable and creatively diverse.

Acknowledgment

The authors would like to thank Dr. Pieter Peers, for his mentorship and guidance in developing this project.

References

[1] D. Cope, "Experiments in music intelligence (EMI)," in *Proceedings of the 1987 International Computer Music Conference, ICMC 1987, Champaign/Urbana, Illinois, USA, August 23-26, 1987*. Michigan Publishing, 1987. [Online]. Available: http://hdl.handle.net/2027/spo.bbp2372.1987.025

[2] K. Collins, "An introduction to procedural music in video games," *Contemporary Music Review*, vol. 28, pp. 5–15, 02 2009.

[3] D. Temperley, *The cognition of basic musical structures*. MIT, 2004.

[4] G. Hadjeres, F. Pachet, and F. Nielsen, "Deepbach: A steerable model for bach chorales generation," in *Proceedings of the 34th International Conference on Machine Learning - Volume 70*, ser. ICML'17. JMLR.org, 2017, p. 1362–1371.

[5] D. Conklin, "Music generation from statistical models," *Journal of New Music Research*, vol. 45, 06 2003.

[6] D. Herremans, K. Sörensen, and D. Martens, "Classification and Generation of Composer-Specific Music Using Global Feature Models and Variable Neighborhood Search," *Computer Music Journal*, vol. 39, no. 3, pp. 71–91, 09 2015. [Online]. Available: https://doi.org/10.1162/COMJ_a_00316

[7] A. Alpern, "Techniques for algorithmic composition of music," *Hampshire College*, 11 1995. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.23.9364&rep=rep1&type=pdf

[8] K. Ebcioğlu, "An expert system for harmonizing chorales in the style of j.s. bach," *The Journal of Logic Programming*, vol. 8, no. 1, pp. 145–185, 1990, special Issue: Logic Programming Applications. [Online]. Available: https://www.sciencedirect.com/science/article/pii/074310669090055A

[9] N. Tokui and H. Iba, "Music composition with interactive evolutionary computation," in *Proceedings of the Third Internation Conference on Generative Art*, vol. 17, no. 2, 2000.

[10] D. Herremans, C. Chuan, and E. Chew, "A functional taxonomy of music generation systems," *CoRR*, vol. abs/1812.04186, 2018. [Online]. Available: http://arxiv.org/abs/1812.04186

[11] R. C. Pinkerton, "Information theory and melody," *Scientific American*, vol. 194, no. 2, pp. 77–87, 1956. [Online]. Available: http://www.jstor.org/stable/26171737

[12] D. Herremans and E. Chew, "Morpheus: automatic music generation with recurrent pattern constraints and tension profiles," in *2016 IEEE Region 10 Conference (TENCON)*, 2016, pp. 282–285.

[13] J. Mccormack, "Grammar based music composition," in *In R. Stocker et al. (Eds.), From Local Interactions to Global Phenomena, Complex Systems 96*. ISO Press, 1996.

[14] A. Horner and D. Goldberg, "Genetic algorithms and computer-assisted music composition," *Urbana*, vol. 51, pp. 437–441, 01 1991.

[15] P. Todd and G. Loy, "A connectionist approach to algorithmic composition," *Computer Music Journal*, vol. 13, pp. 173–194, 1989.

[16] L. Yang, S. Chou, and Y. Yang, "Midinet: A convolutional generative adversarial network for symbolic-domain music generation using 1d and 2d conditions," *CoRR*, vol. abs/1703.10847, 2017. [Online]. Available: http://arxiv.org/abs/1703.10847

[17] J.-P. Briot, "From artificial neural networks to deep learning for music generation – history, concepts and trends," 2020.