TU WIEN Informatics

Polyphonic Music Composition with Grammars

Lukas Eibensteiner
Visual Computing

TU Wien Informatics
Institute of Visual Computing and Human-Centered Technology
Research Unit of Computer Graphics
Supervisor: Univ.Prof. Dipl.-Ing. Dipl.-Ing. Dr.techn. Michael Wimmer
Assistance: Mag. Martin Ilčík
Contact: l.eibensteiner@gmail.com

Polyphonic Music

Prior works on grammars for composition focused on the generation of monophonic structures, such as melodies, rhythms, and chord progressions. Polyphony, where notes can overlap on the timeline, was limited. We were able to integrate polyphony deeply into the generation process and address the following problems:

- **representation**: How can we generate polyphonic structures with a grammar?
- **expressiveness**: How can a user describe common musical patterns?
- **synchronization**: How can we make multiple voices fit together?

While we use the term *polyphony* in a broad sense to mean *any musical structure with overlapping notes*, a more specific meaning is *music with multiple leading voices*. The third problem is therefore particularly interesting. We solve synchronization by (1) generating a shared context, for example a chord progression, (2) branching into individual voices, and (3) using the **`query`** operator to read the common context.

Composition with Grammars

Grammars in natural languages tell us how words from a vocabulary can be arranged into sentences. This principle can be applied to music, where the grammar tells us how notes can be arranged into melodies, rhythms, chord progressions, or complex polyphonic compositions. The example below demonstrates this for a simple melody.

A vocabulary contains *`placeholders`* and **`terminals`**.

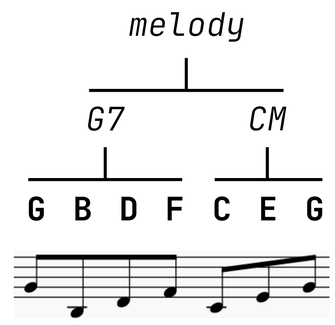
melody G7 CM **A B C D E F G**

One of the placeholders is our starting point.

melody

Each rule replaces a placeholder with a sequence.

melody ⟶ *G7 CM*
G7 ⟶ **G B D F**
CM ⟶ **C E G**

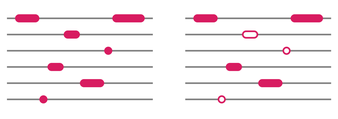


Time Dimension

Similar systems either use no parametric encoding of time, or only note duration, leaving the absolute offset implicit and consequently limiting the possible structures. We associate each entity with two temporal parameters: *time* is the absolute offset of the note on the timeline, and *span* is its duration.

none: sequence with constant durations
simple melodies
rests are explicit

span: sequence with varying durations
rhythmic melodies
rests are explicit

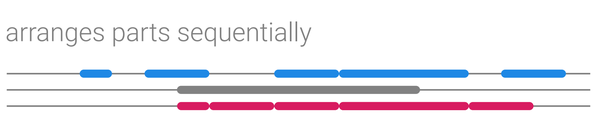


time-span: arbitrary temporal structures
chords, voices
rests are implicit

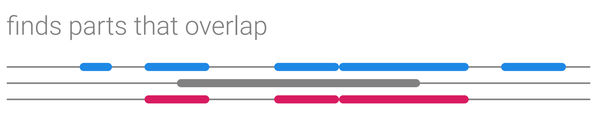
Temporal Operators

Associating each note with a complete time interval means that the composer has to specify at least twice as many temporal parameters than in a purely sequential model. We define convenient operators that simplify working with interval arrangements, for example arranging them sequentially.

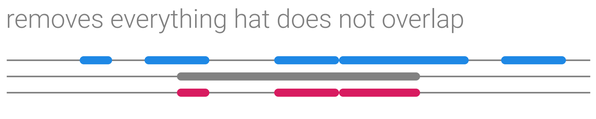
`split``(parts)`
arranges parts sequentially



`query``(parts)`
finds parts that overlap



`trim``(parts)`
removes everything hat does not overlap



Music Programming

Music theorists have invented grammars to model different styles of music, such as jazz, blues, nursery rhymes, and northern Indian drum music. There is probably not one grammar that rules them all. We defined a programming language that can be used to build and evaluate new musical grammars.

- **attributed**: musical objects are sets of key-value pairs, e.g. `chord:7`
- **parametric**: attribute values can be used in logical and arithmetic expressions, e.g. `chord=M`, `note+4`
- **probabilistic**: randomness can be used to cause variation in the output, e.g. **`rand`**, **`choice`**
- **nested**: sentences are first-class citizens and can be stored in attributes or passed to functions, e.g. **`split`**`(`**`notes`**`(4))`
- **modular**: users can define reusable sub-grammars and composable functional modules, e.g. **`notes`**

```
// pseudo-code example
export grammar(
  start ⟶ split(
    <chord:7 note:note+4>
    chord:M
  )
  chord=7 ⟶ choice(
    notes(4)
    split(notes(4))
  )
  chord=M ⟶ notes(3)
  play ⟶ gain:rand(0.5, 1)
)
fun notes: n ⇒ loop(n, i ⇒
  <play:piano note:note+2*i>
)
```

Evaluation

We demonstrated the practical application of our theoretic method by constructing a polyphonic music generator. It generates a single chord progression, a number of parallel voice structures, and finally random arrangements of notes that are synchronized to the chord progression.

Runs in Your Web-Browser

We implemented our system as a TypeScript library. The screenshot below shows our browser-based playback environment. To the left the user can change parameters on the starting entity and thereby influence the outcome. The visualization on the right side shows notes as colored bars. Each color corresponds to one of the voice types.

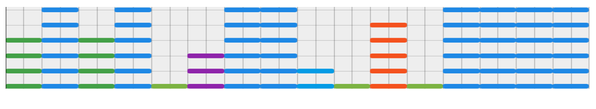
Large-Scale Structure

Once global parameters have been set, the system generates an overall structure for the music. Below we show several examples of abstract parameters we use in our demo. Each line is a possible voice-structure; each colored bar within represents a measure.

diversity: vary structure between voices

monotony: repeat measures within voice

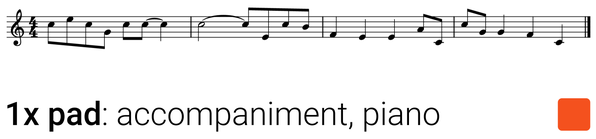
sparsity: remove measures



Small-Scale Motifs

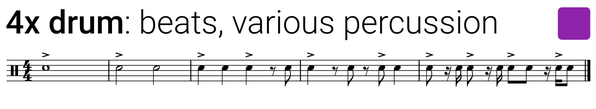
Once an abstract structure exists, the system fills each measure with actual notes. We defined four types of voices, each with their own sub-grammar that generates short, random motifs over a given time-span. Each measure below is one possible result out of hundreds.

2x lead: main melodies, violin / flute



1x pad: accompaniment, piano

1x bass: accompaniment, contrabass



4x drum: beats, various percussion

You can listen to our results and generate your own at: github.com/eibens/thesis-2021