Polyphonic Music Composition with Grammars

Lukas Eibensteiner*
Supervised by: Martin Ilčík†

Institute of Visual Computing and Human-Centered Technology
TU Wien
Vienna / Austria

Abstract

Prior works on music composition with grammars tend to focus on the generation of sequential structures, such as melodies, harmonic progressions, and rhythmic patterns. While the natural output of a grammar is a sequence, this does not have to be reflected in the chronology of the composition. Shape grammar researchers have long internalized this perspective and have used grammars to distribute geometric entities across multiple spatial dimensions. We take inspiration from these works and allow free placement of musical entities on the timeline. With overlapping entities we can model chords, preludes, drum patterns, voices—polyphony on small and large scales. Beyond a theoretic discussion of techniques for polyphonic composition with non-deterministic context-free grammars, we present the results of a procedural music generator built on these principles.

Keywords: procedural, music, composition, functional, programming, context-free, non-deterministic, parametric, attribute-based, grammars

1 Introduction

The automation of musical composition certainly has interesting implications. We can see its successful application in computer games, which usually require more flexible soundscapes compared to non-interactive works. Motifs, themes, and sound effects are triggered by player actions and merge into something that has never been heard before. Similar systems could be built for video platforms, which could offer generative soundtracks that adapt to visuals, dialog, cuts, and camera movements. We might someday have procedural radio, that can be configured to produce a lifetime of music in a personalized style, that reacts to the listener's mood, and from time to time plays variations of their favorite melodies. Today, countless musicians and composers rely on automation in their workflow. Generative grammars—the subject of a large body of research, including this work—are one of the tools in the algorithmic and computer-aided composition toolbox.

When we use grammars to analyze music, we break it into its smallest components, then group them into notes, bars, phrases, themes, or whatever higher-level patterns we find. The premise of using grammars generatively is that we can reverse this process of reduction, beginning at an abstract representation and replacing the abstractions until we get something concrete. For music this could mean starting with a particular style or song structure, successively adding themes, phrases, bars, notes, and finally transforming the notes into an audible signal. In order to create something that corresponds to our abstraction, we have to make a series of informed decisions. Some of these decisions will be guided by the model, for example the rules of the musical genre. Other decisions will depend on subjective taste. A composer is aware of these options, yet is forced to make a decision once they write their score. A system, such as the one presented in this work, uses a special notation that allows the composer to state the options, but defer the decision. The system, aware of all options, can automatically collapse this superposition according to user input or random chance. The result is a new song, unheard even by its composer.

While parallels between language and music and their corresponding theories are conspicuous, there is an important difference: we usually see language as something strictly linear. A symbol comes after a symbol, a word comes after a word, and a sentence comes after a sentence. This might explain why research on grammars for composition focuses on the generation of melodies, harmonic progressions, rhythms—all purely sequential structures. Yet, within a sequential model the handling of polyphonic aspects such as chords, preludes, or interdependent voices is difficult and leads to generation pipelines that consist of multiple heterogeneous components. In this work we try to build a perspective where polyphony is the norm, rather than an appendage, and thus tightly integrated into the generation process. It is a perspective where we see musical objects as volumes in a geometric space, objects that can be divided, stretched, and moved. It is a graphical application of generative grammars, and as such finds its roots in shape and split grammar research. The musical interpretation of these graphical principles is something that we have not seen pronounced in similar works, and it is the subject of this paper.

*l.eibensteiner@gmail.com
†ilcik@cg.tuwien.ac.at

2 Related Work

Within the context of algorithmic composition research, grammars are just one model for the composition process. Consider the survey by Nierhaus [14] and the earlier survey by Papadopoulos and Wiggins [15], who both use a flat classification system, where generative grammars, evolutionary methods, and machine learning are the common clusters. Fernández and Vico [3] propose a detailed hierarchical taxonomy of methods and provide a very useful visualization.

With generative grammars we use a set of formal rules to develop an abstract symbolic structure through iterative refinement into concrete and detailed output. We mostly rely on definition of a grammar hierarchy by Chomsky [1]. A prominent and early generative approach was conceived by Schenker [23], who reduced many forms of music to the *Ursatz*, a sort of fundamental, hidden structure. Another highly popular work in the field of generative music theory is *A Generative Theory of Tonal Music* (GTTM) by Lerdahl et al. [10], in which music is modelled as multiple hierarchical aspects.

Holtzman [6, 7] defined the *Generative Grammar Definition Language* (GDDL) which provides interesting meta-level features, such as the selection of alternatives based on prior rule applications. McCormack [11] proposes a musical grammar system with stochastic rule selection, numeric parameters, and nested grammars. While the generations are primarily sequential, limited polyphony can be achieved by marking multiple notes as a chord.

Steedman [26] defined a context-sensitive grammar for generating chord sequences for 12-bar Blues. The replacement entities divide the time interval of the original entity into equal parts, guaranteeing a monophonic and bounded temporal organization. Steedman later revised the grammar and showed that it can be expressed by an equivalent context-free grammar [25]. Rohrmeier [21, 22] derived harmonic substitution rules from various works on harmonic theory. The grammar is context-free, except for a special rule for pivot chords, which was later also reformulated without context-sensitivity. De Haas et al. [2] remodelled Rohrmeier's earlier grammar and applied it to automatic parsing of jazz pieces.

Gilbert and Conklin [4] defined *probabilistic context-free grammars* (PCFG) for melody generation which use pitch intervals as non-terminals. Keller and Morrison [8] defined another PCFG for generating jazz melodies. They use the note type as a parameter to select non-terminals of varying durations. Tanaka and Furukawa [29] model polyphonic music as a list of voices, where every voice is a list of notes. While the generated output is polyphonic, rules still operate on a sequential voice model, replacing one monophonic sequence with another, usually longer one. Giraud and Staworko [5] used context-free parametric grammars to model Bach inventions. The ability to use sequences of notes as parameters is a feature we also implemented for the system presented in this paper.

Quick and Hudak used a relative time parameter for splitting entities in their *temporal generative graph grammars* (TGGG) [19] and later *probabilistic temporal graph grammars* (PTGG) [18]. PTGGs are used in the composition tool *Kulitta* [16, 17] to generate harmonic progressions, based on production probabilities that were learned from existing music. Melkonian [12] later extended PTGGs to melody and rhythm generation and generalized the harmony generation using a Schenkerian approach. Its capabilities are demonstrated by encoding various grammars from musicologist literature, including the context-free variant of the Steedman grammar [25].

Finally, split grammars, proposed by Wonka et al. [30] and preceded by the work on shape and set grammars by Stiny [27, 28], were a significant inspiration for this work. In a split grammar, shapes are generated by recursively dividing spatial volumes, similar to the splitting of the time dimension that is used in many works on algorithmic composition. The initial split grammar later evolved into the popular *CGA Shape* grammar [13] and its successor *CGA++* [24]. The latter introduces shapes as first-class citizens, allowing operations on generations of sub-grammars, which is a feature we partially implemented.

3 Theory

The goal of this work is not to develop a particular program that generates music, but rather the development of a theoretic framework that facilitates the development of such programs. This framework can be divided into the general theoretic models surrounding context-free grammars and the domain-specific musical models. The former will be presented in this section, while the latter can be found in Section 4, as the musical aspect is strongly intertwined with our particular solution to the stated problem of polyphonic music generation.

3.1 Context-Free Grammars

Context-free grammars have been used by various researchers for procedural generation, arguably because they strike a good balance between power and simplicity. A general discussion about different grammar types for music generation can be found in the early survey by Roads and Wieneke [20]. We will assume the use of context-free grammars throughout this paper.

A context-free grammar $G = (V, P, S)$ consists of a vocabulary of symbols V, a set of rules P, and a dedicated starting symbol $S \in V$. A rule in P can be written as $l : \text{LHS} \longrightarrow \text{RHS}$, where the left-hand side (LHS) describes a subset of V and the right-hand side (RHS) describes a replacement string in V^*, which is the set of arbitrary-length sequences over V. l is an optional label that we use to identify the rule. Consider the following example:

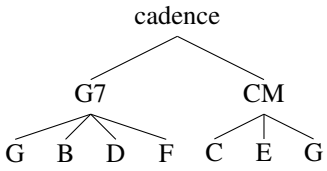

Figure 1: Derivation tree of a context-free grammar

$$
\begin{aligned}
V &= \{cadence, CM, G7, A, B, C, D, E, F, G\} \\
P &= \{p_1, p_2, p_3\} \\
S &= cadence \\
p_1 &: cadence \longrightarrow (G7, CM) \\
p_2 &: G7 \longrightarrow (G, B, D, F) \\
p_3 &: CM \longrightarrow (C, E, G)
\end{aligned}
$$

We can interpret p_1 musically as the fact that *a cadence can be realized by a G seventh chord followed by a C major chord*. Similarly, p_2 (p_3) could mean that *a G seventh chord (C major chord) can be realized by the notes G, B, D, and F (C, E, and G)*. In formal grammar theory we commonly differentiate between terminal symbols, which only appear on the RHS, and non-terminals, that may appear in the LHS and thus are expected to be replaced by other symbols. Albeit, this distinction is not as important for context-free grammars as it is for other types in the Chomsky hierarchy.

The one-to-many relationship between the elements of V expressed by each rule is characteristic for context-free grammars. If we trace the relationships starting at S, we end up with a tree structure, like the one shown in Figure 1, where S is the root, the internal nodes are non-terminals, and the leaves are terminals. The sequence of leaves is also called a *sentence* of the formal language defined by the grammar. The process of generating a sentence from the starting symbol is called *derivation*, and the resulting structure is the *derivation tree*.

Consider the sentence generated by our example grammar, which is (G, B, D, F, C, E, G). There is no indication that these seven symbols represent two chords. For now, we will assume this knowledge is implicit, as the explicit handling of the time dimension is central to our approach and will be discussed Section 4.

3.2 Parametric Grammars

So far, our grammar generates only a single sequence of seven notes, which is not very impressive. If we wanted to generate a cadence in another key, we would have to add three new rules. There are two underlying issues: (1) the vocabulary consists of indivisible, nominal entities, and (2) the rules can only describe replacement with constant sequences of symbols.

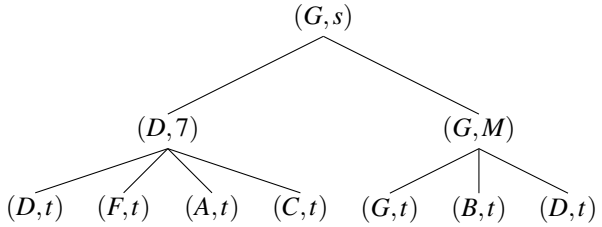

Figure 2: Derivation tree of a parametric grammar.

First, we generalize our vocabulary to an n-dimensional space $V = X_1 \times \ldots \times X_n$, which allows us to normalize the information encoded in our symbols. For example, the two-letter chord symbols $G7$ and CM encode the root note of the chord (G or C) and the type of the chord (7 or M). This implies a representation of a chord as a tuple, which we can generalize to the remaining vocabulary by using s for the starting symbol and t for the terminals:

$$
\begin{aligned}
V &= X_1 \times X_2 \\
X_1 &= \{A, B, C, D, E, F, G\} \\
X_2 &= \{7, M, s, t\}
\end{aligned}
$$

Since the term *symbol* implies some degree of indivisibility, we will use the term *entity* from here on when referring to the elements of a multi-dimensional vocabulary.

Second, we allow replacement sequences to be defined in terms of the input entity. From here on the RHS can be any function $V \to V^*$ from an entity to a sequence of entities. An LHS can be any function $V \to \{\top, \bot\}$, which returns $\top$ if the input entity matches the LHS, and returns $\bot$ otherwise. Both LHS and RHS are parametrized by a single entity, and we will assume an implicit binding of the variables $(x_1, \ldots, x_n) \in V$ on either side of the arrow.

We can now redefine the rules (p_1, p_2, p_3) using the two-dimensional vocabulary with the implicitly bound parameters x_1 and x_2. Instead of explicitly specifying notes, we calculate the replacement based on the input parameters. Since all notes are now relative, we may pick any tuple $(x_1 \in X_1, s)$ as the starting entity and generate a cadence in the corresponding diatonic mode. Figure 2 shows the derivation tree for $S = (G, s)$.

$$
\begin{aligned}
p_1 &: x_2 = s \;\; \longrightarrow ((x_1 + 4, 7), (x_1, M)) \\
p_2 &: x_2 = 7 \;\; \longrightarrow ((x_1, t), (x_1 + 2, t), (x_1 + 4, t), (x_1 + 6, t)) \\
p_3 &: x_2 = M \longrightarrow ((x_1, t), (x_1 + 2, t), (x_1 + 4, t))
\end{aligned}
$$

X_1 represents the notes of the C-major scale, and we can move between scale degrees by adding or subtracting numeric intervals. Concretely, we can treat each pitch letter as equivalent to its zero-based index in the sequence (A, B, C, D, E, F, G) and define an addition operator. For example, $C + 4 = G$ and $G + 2 = B$.

$$+ : X_1 \times \mathbb{Z} \to X_1 : (x_1, z) \mapsto (x_1 + z) \bmod 7$$

Compared to simple symbolic replacement, parametric replacement reduces the number of rules needed to express more complex languages. Yet, they suffer from another kind of scaling issue. In practice our vocabulary will have more than two dimensions, including parameters for meter, scales, loudness, playback, and expressing custom semantics. An explicit tuple representation for entities becomes increasingly unwieldy.

3.2.1 Attributes

Attributes solve scaling issues that arise from higher dimensional vocabularies. Their invention has been credited to Peter Wegner by Knuth [9]. The idea is to gradually change entities over the course of the derivation, rather than explicitly replacing them at every step. We accomplish this with a setter function $set_i : X_i \to (V \to V)$ which sets the ith element of the entity tuple, but keeps the other entries.

For example, instead of (x_1, t) in rules p_2 and p_3 we could write $set_2(t)$ to change x_2 but keep x_1. This is hardly an improvement in two dimensions, but in n dimensions this will eliminate $n-1$ redundant terms. We can further use classic function composition to feed the output of one setter to the next, which again allows us to set any subset of parameters.

With a large number of dimensions, numeric tuple indices will increasingly obfuscate the semantics of our rules. We can instead substitute the numbers with explicit attribute names. For example, we can address x_1 with the name *note* and x_2 with *type*. The consistent use of setters instead of the tuple representation allows us to eliminate any dependence on the order of the dimensions in V.

The formalism is now parametric and supports any number of dimensions. Yet, our grammar is still a flat list of rules. With a growing number of rules it becomes increasingly difficult to orchestrate their application, guaranteeing that they are applied in a certain order. We will solve this next by splitting our grammar into multiple nested sub-grammars.

3.3 Nested Grammars

Context-free derivation can be understood as a mapping from a starting entity in V to a sequence of entities in V^*, which is exactly the definition we use for the RHS. Consequently, we can use grammars as the RHS of a rule, which allows us to divide complex grammars into smaller, maintainable sub-grammars. Grammars with sub-grammars are also known as *hierarchical grammars* and have been used for music generation by McCormack [11].

The decomposition of context-free grammars follows from the properties of the derivation tree, where each subtree can be seen as the result of a particular sub-grammar. Still, more interesting to us is grammar composition, where we combine multiple grammars into a super-grammar using higher-order functions. Context-free grammars are just one possible strategy, albeit it is the most general one in our framework.

We further define a square bracket notation $[f_1, \ldots, f_n]$ to concatenate the results of n right-hand sides into a single sentence. A failing LHS can be used to ignore individual operands. For example, $[f_1, \bot \longrightarrow f_2, f_3]$ is equivalent to $[f_1, f_3]$. Note that concatenation does not necessarily affect the temporal arrangement of the entities; it simply combines the sentences for further processing.

Another strategy is grammar chaining, where the terminals of a grammar are used as starting entities for another grammar. We use an angle bracket notation $\langle f_1, \ldots, f_n \rangle$ to indicate that the input entity should be passed to f_1, the resulting entities to f_2, and so forth. The base case $\langle\rangle$ is equivalent to an identity mapping of the input entity. A failing LHS can be used to break the chain. For example, $\langle f_1, \bot \longrightarrow f_2, f_3 \rangle$ is equivalent to $\langle f_1 \rangle = f_1$. If we only want to skip f_2, we can wrap it in another pair of angle brackets to get $\langle f_1, \langle \bot \longrightarrow f_2 \rangle, f_3 \rangle$, which is equivalent to $\langle f_1, \langle\rangle, f_3 \rangle = \langle f_1, f_3 \rangle$.

Treating grammars and sentences as first-class citizens opens up very interesting possibilities. For example, one can define a parameter with a value space V^* and propagate grammar results through the derivation tree. We will later use this for synchronizing multiple independent voices to a shared harmonic progression. Also consider Giraud and Staworko [5], who pass motifs as parameters, and Schwarz and Müller [24], who introduced the concept to shape grammars.

The example below defines a super-grammar *PIECE* that is composed of three sub-grammars. We store the result of the *CHORDS* grammar inside the *chords* attribute and then pass the entity to the *PAD* and *BASS* grammars. We assume that *CHORDS* generates some form of harmonic progression, while *PAD* and *BASS* generate notes based on the entities in *chords*.

$$PIECE : \langle set_{chords}(CHORDS), [PAD, BASS] \rangle$$

While we can now use parameters, attributes, and functional composition to develop complex grammars, their results will not yet be very surprising. Variation is the missing ingredient that elevates the formalism from a complex tool for music notation to a powerful tool for automatic composition.

3.4 Non-Deterministic Grammars

Some classic composers published musical games that allow the composition of new pieces by randomly selecting from a framework of predefined bars, most notably

Mozart with his minuet generator [31]. The player of the game needed no musical knowledge, only a pair of dice. We will use a similar approach, where the grammar is the framework, and the replacement decisions are delegated to a non-deterministic selection process.

We differentiate between two types of variation. *Structural variation* directly affects the structure of the derivation tree and occurs when there are multiple rules that can replace an entity. For example, our familiar rule p_1 and the new rule $p_{1'}$ both match an entity $type = s$. The derivation algorithm randomly picks one of them.

$$p_1 : type = s \longrightarrow ((note + 4, 7), (note, M))$$
$$p_{1'} : type = s \longrightarrow ((note, M), (note + 3, M))$$

In order to achieve this effect within a single rule, we can define a function *choice* that randomly returns one of its arguments. For example, the two rules above could be expressed with a single LHS and a choice with two options on the RHS.

$$\begin{aligned} type = s \longrightarrow choice(& \\ & ((note + 4, 7), (note, M)), \\ & ((note, M), (note + 3, M)) \\ ) & \end{aligned}$$

Parametric Variation directly affects the values of parameters and allows non-discrete randomization. A simple mechanism is a non-deterministic variable *rand* with a uniform distribution over $[0, 1]$. For example, in order to simulate varying loudness of notes in a real-life performance, we can randomize the value of a numeric *gain* attribute:

$$type = t \wedge gain = 0 \longrightarrow set_{gain}(rand)$$

We can also use random numbers for structural variation. For example, the *pr* function below applies an RHS f with probability $p \in [0, 1]$:

$$pr : (p, f) \mapsto \langle rand \leq p \longrightarrow f \rangle$$

The random numbers are sampled from a pseudo-random number generator (PRNG) with an internal counter. We propagate the PRNG downwards in the derivation tree as a parameter, which means per default the subtrees are all desynchronized from each other. If we want to synchronize two subtrees, we can initialize their PRNGs with the same value, which guarantees that all random processes within both subtrees have the same outcome.

Alternatively, we can calculate the result of a non-deterministic sub-grammar before we branch into the subtrees, as we do with the *chords* attribute in the example in Section 3.3. Either method works by providing shared context to the subtrees. This shared context is especially important for polyphonic composition, as we must guarantee that multiple voices fit together. We have now established the theoretic foundation and will continue with the discussion of our polyphonic composition model.

4 Approach

We define polyphonic music as two or more interrelated voices that play at the same time. They are independent, in the sense that they can define their own movement and rhythmic patterns, but are synchronized to a shared metric and harmonic framework. In music notation individual voices and instruments are often written in separate *staves*, for example, the left and right hand of the piano piece in Figure 3. Each stave spans the whole duration of the piece, and the individual voices within are read sequentially.

To implement this model in a context-free grammar we could derive each voice individually, then combine the results by playing them at the same time. Yet, without further constraints this approach is insufficient as the voices can become desynchronized due to non-determinism in the derivation process. As soon as the algorithm branches into different voices, sharing information between them becomes difficult. Rather, we need to provide a shared metric and harmonic context that allows the voices to actually fit together. Our solution to this is threefold:

1. We allow delaying the point of separation of voices to arbitrary depths in the derivation tree. For this we associate each entity with a time interval, in a way that is reminiscent of the use of spatial volumes in split-grammars, and allow layering, recursive splitting, and space-filling.

2. Using nested grammars we can generate musical textures, which we can use as a common substrate for the voices of a polyphonic piece. For example, we can generate a harmonic progression in an isolated subtree and access it from an independently generated melody and bass line.

3. We define a relative measurement system, expressed as a set of context-dependent temporal units, to align voices in different subtrees. For example, using a common tempo and time signature, we can guarantee that beats of multiple voices coincide.

Polyphony is a phenomenon that relates strongly to the time dimension. Since pitch and harmony are not central to this work, we do not discuss these aspects in detail. For

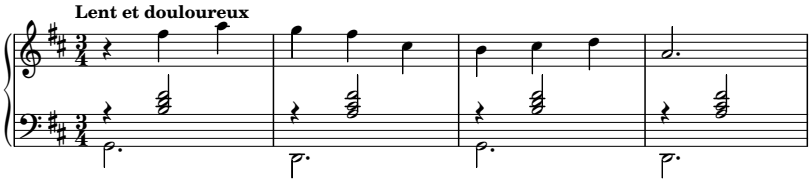

Figure 3: Score representation of measures five to eight of *Gymnopédie No.1* by Eric Satie.

the practical evaluation we used a relative system based on the key and diatonic mode for converting from scale degrees to absolute pitches.

4.1 Temporal Scope

We can interpret an entity's position in the sentence as its position on the timeline, which is a simple and common approach [11, 22, 25]. Adding a duration parameter allows us to generate entities of different lengths and model arbitrary rhythms, which has been demonstrated by Quick and Hudak [19] amongst others. Yet, for polyphonic structures, the terminals must be allowed to overlap, which is not possible when their absolute temporal offset is implicitly derived from their position in the sentence.

We borrow the concept of scope from split-grammars, which makes the time interval of each entity explicit and independent of the sentence. In graphics the scope is a 3D transformation of space, whereas in music it is a 1D transformation of time. We encode this transformation as a number pair $(t_0, t_1) \in \mathbb{R}^2$ and further define its duration $\Delta = t_1 - t_0$. The scope is passed as a parameter, which means we can generate arbitrary interval arrangements on any level of the derivation tree.

For example, we can interpret the temporal structure of the score in Figure 3 as four levels of scope operations visualized in Figure 4. In (1) we split the piece into a sequence of whole measure intervals. (2) replaces each measure with three parallel intervals for the voices, which we split into notes and rests in (3). In (4b) we once again use parallel placement to stack the chord notes. An alternative interpretation would swap step (1) and (2) and generate the voices on the first level and measures on the second.

4.2 Scope Operators

The scope is treated like any other parameter, which means we can freely modify it on the RHS. When we do not change the scope, the replacement entities cover the same time interval, so polyphony can be considered the default in our model. Similar to the $+$ operator for notes in Section 3.2, we can define operators that modify the scope, for example simple linear transformations that stretch or move the entity. We take further inspiration from split-grammars and implement the following geometric operators:

- The *repeat* operator fills the scope with n intervals of a fixed duration Δ'. We calculate $n = \lfloor \Delta/\Delta' \rfloor$ to generate zero or more entities that do not exceed the original duration Δ. An optional weighting factor can be used to distribute the remaining space.

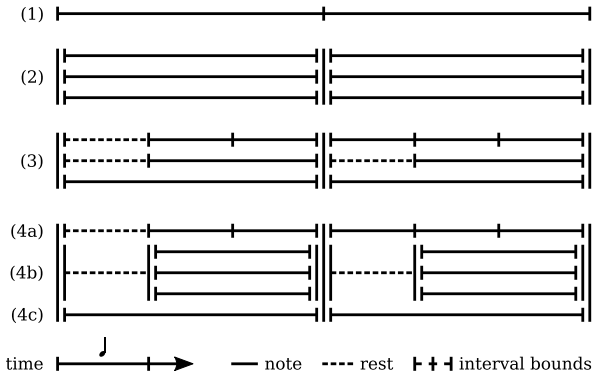

Figure 4: A step by step construction of the temporal structure of the fifth and sixth measure of *Gymnopédie No.1* by Eric Satie.

- The *repeat cover* operator works similar to the *repeat* operator, with the difference that it calculates $n = \lceil \Delta/\Delta' \rceil$. It generates at least one entity, where the last entity potentially exceed the original scope.

- The *split* operator divides the scope into n intervals with durations $(\Delta_1, \ldots, \Delta_n)$. Additionally, one can specify numbers $(\omega_1, \ldots, \omega_n)$ which are used as weights to distribute any remaining space. Unlike the repeat operator, it can generate irregular divisions of an entity.

- The *trim* operator removes any part of an entity that exceeds the scope of the current input entity. Entities that have an empty intersection with the current scope are removed entirely. We can pass the result of a repeat or split operation to the trim operator to guarantee that the entities fit into the current scope.

- The *query* operator finds entities in a sentence that overlap with the current scope. It is similar to *trim*, but has different use cases. Queries allow us to treat the results of sub-grammars as functions over time, which is very useful when we need to share local information between voices. For example, we would use *query* to synchronize multiple voices to the same harmonic progression.

unit	factor	base
second	1	second
minute	60	second
beat	$1/T$	minute
measure	N	beat
whole	B	beat
half	$1/2$	whole
quarter	$1/4$	whole
eighth	$1/8$	whole
sixteenth	$1/16$	whole

Table 1: Conversion table for temporal units. Reading: *A minute is equal to 60 seconds.*

4.3 Temporal Units

Time in music is usually not described in purely relative and hierarchical terms, but rather through a temporal grid established by the piece's tempo and meter. We implement meter in our system with three parameters. The tempo parameter $T \in \mathbb{R}_+$ is used for the conversion between beats and physical time in seconds. We measure T in beats per minute (BPM). The beat count parameter $N \in \mathbb{R}_+$ specifies the number of beats per measure. Finally, the beat type parameter $B \in \mathbb{R}_+$ allows us to calculate the length of a whole note as B beats.

The pair (N,B) has the same semantics as the two numbers of the time signature in musical notation. For example, if $N = 3$ and $B = 4$, there are three beats per measure, and the duration of each beat is equivalent to $1/4$ of a whole note. We define the conversion factors of various temporal units that can be derived from these parameters in Table 1.

4.4 Example

Based on the presented theoretic constructs, we implemented a simple polyphonic music generator. As stated at the beginning of Section 3, the development of a particular generator is not the goal of this work. The following example is just one possible application of the theory. Our generator consists of the following sub-grammars:

- The *piece* grammar serves as the entry point. It first randomizes global parameters such as *tempo*, *beats*, *beatType* and *key*, then generates a chord progression that is propagated to the voice layers.
- The *layer* grammar generates a binary tree that serves as the skeleton for a chord progression or voice layer. We can control the degree of synchronization within a layer with the *monotony* attribute and the degree of synchronization between layers with the *diversity* attribute. The *depth* of the binary tree dictates the total length of the song, as a leaf equals one measure.

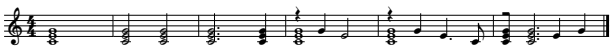

Figure 5: Each measure shows a possible result of the *pad* grammar (File: pad.mp3) using four quarter notes per measure and a C major scale.

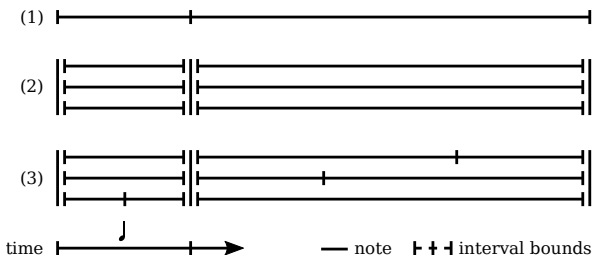

Figure 6: A step by step construction of a possible result of the *pad* grammar.

- The *progression* grammar assigns a random chord from a chord pool to the leaves of a layer. We only evaluate it once and pass the result as a parameter to the voice layers.
- The *orchestra* grammar defines the voice layers: two *lead* voices (violin and flute), one *pad* voice (piano), one *bass* voice (contrabass), and four *drum* voices. The two lead voices are constrained in such a way that they either play an individual motif alone, or a common motif in unison.
- The *motif* grammar queries the chord progression layer and assigns the harmonic information to the current entity. It then selects a dedicated sub-grammar based on the voice type.
- The *lead*, *pad*, *bass*, and *drum* grammars generate the actual notes within a measure. In these grammars we frequently use the repeat, split, and trim operators to fill measures with melodic, rhythmic, and harmonic patterns.

Due to spatial constraints we cannot provide a complete formal specification of the presented generator and will instead confine our detailed discussion to one exemplary component: the *pad* motif grammar.

4.4.1 Example: Pad Grammar

The *pad* grammar develops notes for a piano accompaniment. It adapts to the time interval, metric grid, and key of the current entity. We first define a reusable helper function *cut* which splits an entity into two parts: a fixed part with duration Δ' and a flexible part that receives all of the remaining space by setting the weight parameter $\omega = 1$. We use *choice* to randomize the order of these two parts. The outer *trim* is useful when Δ' exceeds the Δ of the input entity:

$$
\begin{aligned}
&cut : \Delta' \mapsto trim(split(choice(\\
&\quad [set_\Delta(\Delta'), set_\omega(1)],\\
&\quad [set_\omega(1), set_\Delta(\Delta')]\\
&)))
\end{aligned}
$$

We define the *pad* grammar using our angle bracket notation from Section 3.3 to chain three right-hand sides: f_1 applies *cut* with a probability of 0.5. We use *choice* to pick a random duration for the fixed part. f_2 generates three parallel notes for every input entity, similar to the RHS of p_3 in Section 3.2. Finally, we apply f_3 to each of the entities generated by f_2. f_3 is equivalent to f_1, only this time it is applied to three or six notes, depending on whether *cut* was applied in f_1:

$$
\begin{aligned}
&pad : \langle\\
&\quad f_1 : pr(0.5, cut(choice(half, quarter, eighth))),\\
&\quad f_2 : [\langle\rangle, set_{note}(note+2), set_{note}(note+4)],\\
&\quad f_3 : pr(0.5, cut(choice(half, quarter, eighth)))\\
&\rangle
\end{aligned}
$$

This particular grammar does not use recursion or complex matching criteria. Nevertheless, it demonstrates both sequential and parallel note placement and the use of temporal units. Figure 5 shows a selection of results. Figure 6 shows the intermediate results for one possible derivation path. In (1) a quarter note is cut off from the beginning of the input entity. (2) expands the two parts into chords of three notes each. (3) applies the *cut* function to three of the six notes. Either part of the three splits could have been the flexible one in this example.

5 Implementation

We implemented the theoretic framework and our practical example using functional programming in TypeScript. The type system is capable of statically checking most of our higher-order functional constructs. For playback we use the Web Audio API, which provides accurate scheduling of sounds on a timeline. Since the grammar generates a symbolic score we rely on prerecorded instrument samples. Grammar authors can develop new grammars in their editor of choice, while a provided script compiles and reloads the graphical interface in the web browser after every change. Figure 7 visualizes 'song1.mp3' from Table 2 in our browser-based interface.

file	key	mode	bpm	meter
song1.mp3	C#	minor	120	4/4
song2.mp3	A	major	136	6/8
song3.mp3	E	major	114	4/4
song4.mp3	E	minor	93	4/4
song5.mp3	A#	minor	108	2/8
song6.mp3	G#	major	87	3/8
song7.mp3	A	major	126	6/8

Table 2: Seven pieces automatically composed by our polyphonic music generator. The audio files are attached to this document and available online:
`https://github.com/eibens/cescg-2021`

6 Results

We implemented the example from Section 4.4 within our browser-based framework and generated various pieces. A selection of these pieces, their audio files, and initial parameters are listed in Table 2. We will briefly contrast our polyphonic model with the most relevant related works and then summarize our subjective observations about its practical applicability.

McCormack [11] achieves polyphony with both parallel and sequential placement, but only for discrete symbols. Since there is no explicit temporal parameters, moving, resizing, or arbitrary splits are likely not possible. While our model of time is more expressive, McCormack's system is more powerful in theory, as it allows context-sensitive matching.

Tanaka and Furukawa [29] use rules that replace notes in all voices at once. There is no split operator and the piece grows in length with every replacement. While this insertion of entities can be emulated with split operations on the whole voice, it is arguably counter-intuitive to do so within our temporal model. The system is further limited to a fixed set of voices, while our model can add voices on every level of the derivation tree.

Quick and Hudak's PTGG [18] and its later extension by Melkonian [12] are based on split operations with relative units and note durations, which are both available in our framework. Polyphony is not integrated into the grammar itself, and the presented scores do not suggest that time intervals may freely overlap after the polyphonic post-processing pass.

Our own model of time appears to be capable of expressing the temporal operations used in prior works. Its recursive parallelism allows one to define pieces with an unbounded number of entities within a temporal slice. We believe this is an important feature since many types of music require polyphony on at least two levels: voices and chords.

Artifacts that arise from implicit temporal representations are eliminated. For example, a rest has no explicit representation because we can discard an entity by replacing it with the empty sequence.

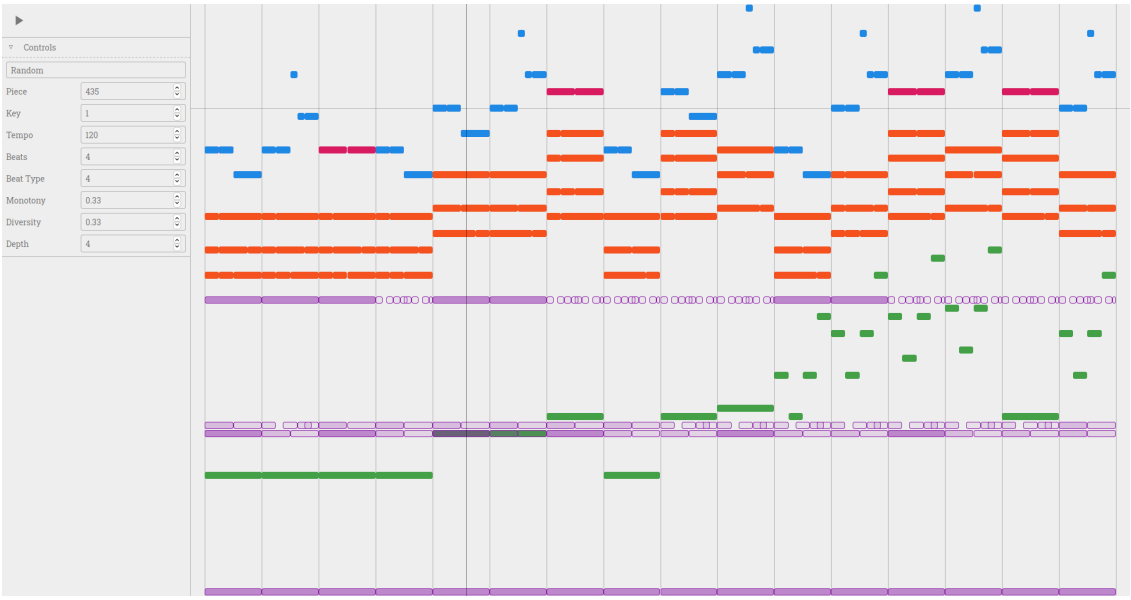

Figure 7: The controls to the left can be used to manipulate parameters on the starting entity. The score visualization to the right shows time on the X axis, logarithmic pitch on the Y axis, and the scope of the terminal entities as colored bars. Notes of the same color belong to the same voice, and the opacity encodes the loudness.

Temporal hierarchies, where the replacement entities do not exceed the original scope, are common in music and our model is well capable of describing them with the split, repeat, and trim operators. At the same time, we can easily break the hierarchy by moving or resizing entities. For example, we can randomly offset notes in a drum beat, or add a prelude to a measure. Although, after moving an entity one should consider querying its new context.

Finally, synchronizing multiple voices to one or more hidden layers intuitively mirrors how a human composer may initially define a metric grid and harmonic progressions, and develop the notes for the instruments in a second pass. The integrated generation of parallel structures such as chords allows us to use this information while the derivation is still in progress. For example, we can query the highest note in a voice and generate a second voice that always stays above it. This is more difficult when structures such as chords are only expanded in a post-processing step.

7 Conclusion

In this work we presented a formal grammar approach for generating polyphonic music. Each musical entity is associated with an independent temporal scope, and replacement entities can be freely arranged on the timeline. Placement of entities is facilitated by multiple scope-based operators. Parallel voices can be supplied with common context by retaining terminal strings of sub-grammars and reading their entities with a time-based query mechanism.

While we can apply locally context-sensitive operations by passing the result of a sub-grammar to a function, the derivation process itself is still fundamentally context-free. For example, we cannot specify an LHS that finds a pair of chords that form a cadence, as this would require us to match on entity pairs.

Further, the selection of scope-based operators that our system provides out of the box is still limited. Additional operations on entity sequences could be used for more powerful effects, such as inferring a chord progression from a generated melody and generating a corresponding accompaniment.

As an alternative to random number generators, one could integrate user input directly into the derivation process. When an external input is required, the derivation of the sub-tree could be suspended until the user makes a decision. This would allow users to assume manual control over any aspect of the generation.

References

[1] Noam Chomsky. Three models for the description of language. *IRE Transactions on Information Theory*, 2(3):113–124, 1956.

[2] W Bas De Haas, Martin Rohrmeier, Remco C Veltkamp, and Frans Wiering. Modeling harmonic similarity using a generative grammar of tonal harmony. In *Proceedings of the 10th International Conference on Music Information Retrieval (ISMIR)*, 2009.

[3] Jose D Fernández and Francisco Vico. AI methods in algorithmic composition: A comprehensive survey.

Journal of Artificial Intelligence Research, 48:513–582, 2013.

[4] Édouard Gilbert and Darrell Conklin. A probabilistic context-free grammar for melodic reduction. In *Proceedings of the International Workshop on Artificial Intelligence and Music, 20th International Joint Conference on Artificial Intelligence (IJCAI), Hyderabad, India*, pages 83–94, 2007.

[5] Mathieu Giraud and Slawek Staworko. Modeling musical structure with parametric grammars. In *Mathematics and Computation in Music*, pages 85–96. Springer, 2015.

[6] SR Holtzman. Using generative grammars for music composition. *Computer Music Journal*, 5(1):51–64, 1981.

[7] Steven R Holtzman. A generative grammar definition language for music. *Journal of New Music Research*, 9(1):1–48, 1980.

[8] Robert M Keller and David R Morrison. A grammatical approach to automatic improvisation. In *Fourth Sound and Music Conference*, 2007.

[9] Donald E Knuth. The genesis of attribute grammars. In *Attribute Grammars and Their Applications*, pages 1–12. Springer, 1990.

[10] Fred Lerdahl, Ray Jackendoff, et al. *A generative theory of tonal music*. The MIT Press, 1983.

[11] Jon McCormack. Grammar based music composition. *Complex systems*, 96:321–336, 1996.

[12] Orestis Melkonian. Music as language: putting probabilistic temporal graph grammars to good use. In *Proceedings of the 7th ACM SIGPLAN International Workshop*, pages 1–10, 2019.

[13] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. Procedural modeling of buildings. *ACM Transactions on Graphics*, 25(3):614–623, 2006.

[14] Gerhard Nierhaus. *Algorithmic composition: paradigms of automated music generation*. Springer, 2009.

[15] George Papadopoulos and Geraint Wiggins. AI methods for algorithmic composition: A survey, a critical view and future prospects. In *Proceedings of the AISB symposium on musical creativity*, volume 124, pages 110–117, 1999.

[16] Donya Quick. *Kulitta: A Framework for Automated Music Composition*. Yale University, 2014.

[17] Donya Quick. Composing with kulitta. In *Proceedings of the International Computer Music Conference*, 2015.

[18] Donya Quick and Paul Hudak. Grammar-based automated music composition in Haskell. In *Proceedings of the 1st ACM SIGPLAN workshop on Functional art, music, modeling & design*, pages 59–70. Association for Computing Machinery, 2013.

[19] Donya Quick and Paul Hudak. A temporal generative graph grammar for harmonic and metrical structure. In *Proceedings of the International Computer Music Conference*, 2013.

[20] Curtis Roads and Paul Wieneke. Grammars as representations for music. *Computer Music Journal*, pages 48–55, 1979.

[21] Martin Rohrmeier. A generative grammar approach to diatonic harmonic structure. In *Proceedings of the 4th sound and music computing conference*, pages 97–100, 2007.

[22] Martin Rohrmeier. Towards a generative syntax of tonal harmony. *Journal of Mathematics and Music*, 5(1):35–53, 2011.

[23] Heinrich Schenker. *Der Freie Satz*. Universal Edition, 1935.

[24] Michael Schwarz and Pascal Müller. Advanced procedural modeling of architecture. *ACM Transactions on Graphics*, 34(4):1–12, 2015.

[25] Mark Steedman. The blues and the abstract truth: Music and mental models. *Mental models in cognitive science*, pages 305–318, 1996.

[26] Mark J Steedman. A generative grammar for jazz chord sequences. *Music Perception*, 2(1):52–77, 1984.

[27] George Stiny. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design*, 7(3):343–351, 1980.

[28] George Stiny. Spatial relations and grammars. *Environment and Planning B: Planning and Design*, 9(1):113–114, 1982.

[29] Tsubasa Tanaka and Kiyoshi Furukawa. Automatic melodic grammar generation for polyphonic music using a classifier system. In *Proceedings of the 9th Sound and Music Computing Conference*, 2012.

[30] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. Instant architecture. *ACM Transactions on Graphics*, 22(3):669–677, 2003.

[31] Neal Zaslaw. *Essays in Honor of László Somfai on His 70th Birthday. Studies in the Sources and the Interpretation of Music*, chapter Mozart's Modular Minuet Machine, pages 219–235. Scarecrow Press, 01 2005.