Temporal-Scope Grammars for Polyphonic Music Generation

Lukas Eibensteiner
TU Wien
Vienna, Austria
l.eibensteiner@gmail.com

Martin Ilčík
TU Wien
Vienna, Austria
ilcik@cg.tuwien.ac.at

Michael Wimmer
TU Wien
Vienna, Austria
wimmer@cg.tuwien.ac.at

Abstract

We present temporal-scope grammars for automatic composition of polyphonic music. In the context of this work, polyphony can refer to any arrangement of musical entities (notes, chords, measures, etc.) that is not purely sequential in the time dimension. Given that the natural output of a grammar is a sequence, the generation of sequential structures, such as melodies, harmonic progressions, and rhythmic patterns, follows intuitively. By contrast, we associate each musical entity with an independent temporal scope, allowing the representation of arbitrary note arrangements on every level of the grammar. With overlapping entities we can model chords, drum patterns, and parallel voices – polyphony on small and large scales. We further propose the propagation of sub-grammar results through the derivation tree for synchronizing independently generated voices. For example, we can synchronize the notes of a melody and bass line by reading from a shared harmonic progression.

CCS Concepts: • **Applied computing** → **Sound and music computing**; • **Theory of computation** → *Grammars and context-free languages*; • **Software and its engineering** → *Domain specific languages*; Functional languages.

Keywords: algorithmic composition, music, domain specific language

ACM Reference Format:
Lukas Eibensteiner, Martin Ilčík, and Michael Wimmer. 2021. Temporal-Scope Grammars for Polyphonic Music Generation. In *Proceedings of the 9th ACM SIGPLAN International Workshop on Functional Art, Music, Modelling, and Design (FARM '21), August 27, 2021, Virtual Event, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3471872.3472971

FARM '21, August 27, 2021, Virtual Event, Republic of Korea

ACM ISBN 978-1-4503-8613-5/21/08…$15.00
https://doi.org/10.1145/3471872.3472971

1 Introduction

Automation of musical composition certainly has interesting implications. We can see its successful application in computer games, where motifs, themes, and sound effects are triggered by player actions. 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 reacts to the listener's mood and plays variations of their favorite melodies. Today, countless musicians and composers rely on automation in their workflow. *Grammars* 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.

The application of formal grammar theory for music analysis and composition has a long history. The use of abstractions in musical notation definitely predates the first formalization of grammar theory by Chomsky [2]. Schenker's work [26] is an early example of a generative approach, where music is reduced to the *Ursatz*, a hidden structure beyond the concrete musical surface. Another popular work is *A Generative Theory of Tonal Music* by Lerdahl and Jackendoff [10], who model music as four hierarchical aspects governed by various types of rules.

Since the output of a grammar is a sequence, modelling sequential structures such as melodies, harmonic progressions, or rhythms is very intuitive. Yet, within a sequential model of time the representation of polyphonic aspects such as chords, parallel voices, or exotic sound effects is difficult. Existing solutions for generating polyphony with grammars require context-sensitivity in the rules or sentence structure [11, 32] or external processing steps, where sequential harmonic progressions are expanded into chord notes and voices [20].

In this work we build a perspective where polyphony is the norm and tightly integrated into the grammar itself:

- We define *temporal-scope grammars*, where each musical entity is associated with an explicit time-span that can be independently divided, stretched, and moved.
- We propose a time-based query mechanism, which gives entities in different parts of the derivation access to the same local context. This allows us to synchronize any number of parallel voices to a set of abstract musical textures.
- Finally, for defining and evaluating temporal-scope grammars, we developed a functional domain-specific language (DSL) in form of a TypeScript library.

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 [16], who both use a flat classification system, where generative grammars, evolutionary methods, and machine learning are the common clusters. Fernández and Vico [4] propose a detailed hierarchical taxonomy of methods and provide a very useful visualization.

Grammars are a sub-category of rule-based systems, where domain knowledge is encoded explicitly as a system of formal rules. Another sub-category are constraint-based methods, where the composer defines a space of possible pieces and uses logical constraints for selecting candidate solutions. Consider the survey by Anders and Miranda [1]. The definition of the solution space is of particular interest to us. For example, *PWConstraints* by Rueda et al. [25] represents polyphonic music as a set of parallel voices, where each voice is a list of harmonies represented by sets of notes. The music representation framework *MusES* by Pachet et al. [15] uses a point in time and a duration for modelling temporal structures. This representation has been used for constraint programming by Roy and Pachet [24] and also as the basis of this work.

One of our example generators uses basic constraint programming for selecting rhythm and melody fragments. Apart from that, our system is based on generative grammars. It uses a set of context-free replacement rules to develop an abstract symbolic structure into concrete and detailed output. We rely on the definition of a grammar hierarchy by Chomsky [2].

Holtzman [7, 8] 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 [29] 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, making it context-free [28]. Rohrmeier [22, 23] 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 freed of context-sensitivity. De Haas et al. [3] remodelled Rohrmeier's earlier grammar and applied it to automatic parsing of jazz pieces.

Gilbert and Conklin [5] defined *probabilistic context-free grammars* (PCFG) for melody generation, which use pitch intervals as non-terminals. Giraud and Staworko [6] 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. Tanaka and Furukawa [32] model music as a list of voices, where each voice is a list of notes, and notes are replaced in all voices at once. The number of voices is consequently limited on the global level. Rules in this system are not intended to be designed by hand.

Quick and Hudak used a relative time parameter for splitting entities in their *temporal generative graph grammars* (TGGG) [20] and later *probabilistic temporal graph grammars* (PTGG) [19]. PTGGs are used in the composition tool *Kulitta* [17, 18] to generate harmonic progressions, based on production probabilities that were learned from existing music. PTGGs do not generate polyphonic structures, however, Kulitta expands their output into voices in a subsequent series of processing steps. Melkonian [12] later extended PTGGs to melody and rhythm generation and generalized the harmony generation using a Schenkerian approach. Its capabilities were demonstrated by encoding various grammars from musicologist literature, including the context-free variant of the Steedman grammar [28].

Finally, split grammars from the computer graphics domain, proposed by Wonka et al. [33] and preceded by the work on shape and set grammars by Stiny [30, 31], were a significant inspiration for this work. In graphics, where one usually deals with two or more spatial dimensions, an explicit scope is a prerequisite. The initial split grammar later evolved into the popular *CGA Shape* grammar [13] and its successor *CGA++* [27]. The latter introduces shapes as first-class citizens, allowing operations on generations of sub-grammars, which is a feature we also implemented.

3 Theory

The goal of this work is not to develop a particular program that generates music, but rather to design a theoretic framework that facilitates the development of such programs. We construct this framework with a series of generalizations starting from context-free grammars, including the addition

of abstract features described in this section (i.e., parameters, attributes, nesting, and non-deterministic functions) as well as the domain-specific musical models described in Section 4.

3.1 Context-Free Grammars

Context-free grammars have been used in various ways 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 [21]. 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) is an element of V and the right-hand side (RHS) is 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:

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

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. However, this distinction is not as important for context-free grammars as it is for other types in the Chomsky hierarchy and therefore we ignore it.

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 *derivation tree*, 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 terminals resulting from the *derivation* is also called a *sentence* of the formal language defined by the grammar.

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.

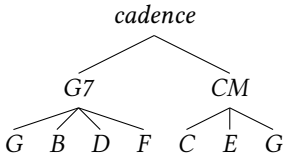

Figure 1. Derivation tree of a context-free grammar

3.2 Parametric Grammars

So far, our grammar generates only a single sequence of seven notes. If we wanted to generate a cadence in another key, we would have to add missing semitones to the vocabulary and transposed copies of $\{p_1, p_2, p_3\}$ to the rule set. 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.

To address these, we first generalize the vocabulary to an n-dimensional space $V = X_1 \times \ldots \times X_n$, which allows us to normalize the information encoded in the 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. We generalize this to the remaining symbols by using the letter s for the starting symbol and t for the terminals:

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

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

Second, we allow replacement sequences to be defined in terms of the input entity, i.e., the RHS can be any function $V \rightarrow V^*$ from an entity to a sequence of entities. The LHS can be any binary function over V that is true when the input entity matches, and false otherwise. Both LHS and RHS accept a single entity as input, 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 key. Figure 2 shows the derivation tree for $S = (G, s)$.

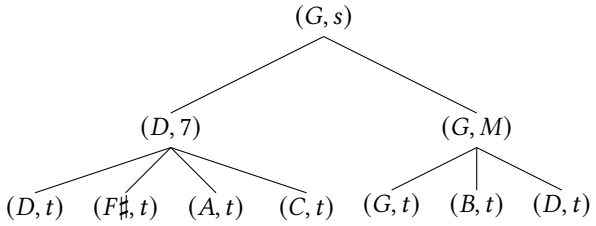

Figure 2. Derivation tree of a parametric grammar.

$$p_1 : x_2 = s \longrightarrow ((x_1 + 7, 7), (x_1, M))$$
$$p_2 : x_2 = 7 \longrightarrow ((x_1, t), (x_1 + 4, t), (x_1 + 7, t), (x_1 + 10, t))$$
$$p_3 : x_2 = M \longrightarrow ((x_1, t), (x_1 + 4, t), (x_1 + 7, t))$$

X_1 represents the notes of the chromatic scale, and we can move between scale degrees by adding or subtracting numeric intervals. Concretely, we can treat each pitch class as equivalent to its zero-based index in the sequence (A, $B\flat$, B, C, $C\sharp$, D,$D\sharp$, E, F, $F\sharp$, G, $G\sharp$) and define a cyclic addition operator. For example, $C + 7 = G$ and $G + 4 = B$.

$$+ : X_1 \times \mathbb{Z} \to X_1 : (x_1, z) \mapsto (x_1 + z) \bmod 12$$

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, the vocabulary will have more 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*. 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 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 RHS's into a single sentence. An LHS can be used to ignore individual operands. For example, if the LHS g does not match the input entity, $[f_1, g \longrightarrow f_2, f_3]$ 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. An LHS can be used to exit the chain early. For example, if the LHS g does not match any of the results of f_1, $\langle f_1, g \longrightarrow f_2, f_3 \rangle$ 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 $\langle f_1, \langle g \longrightarrow f_2 \rangle, f_3 \rangle$, which would result in $\langle f_1, \langle\rangle, f_3 \rangle = \langle f_1, f_3 \rangle$.

Treating grammars and sentences as first-class citizens opens up interesting possibilities. Consider Giraud and Staworko [6], who pass motifs as parameters, and Schwarz and Müller [27], who introduced this to shape grammars. For example, one can define a parameter with a value space V^* and propagate grammar results through the derivation tree by using $setx_k$, which stores the result of f in the attribute k.

$$setx_k(f) = (v \in V) \mapsto (set_k(f(v)))(v)$$

The example below defines a super-grammar *piece*, which is composed of three sub-grammars. We store a random result of the *progression* grammar inside the *chords* attribute and then pass the entity to the *pad* and *bass* grammars. We assume that *progression* generates some sequence of entities with harmonic information, while *pad* and *bass* generate notes based on the entities in *chords*.

$$piece : \langle setx_{chords}(progression), [pad, bass] \rangle$$

Instead of propagating sub-grammar results, one could also propagate the sub-grammar itself using the normal set_k function. This is useful for grammars where users can inject custom functionality.

We can now use parameters, attributes, and functional composition to develop complex grammars, yet their results will not 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 [34]. 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 with $type = s$. The derivation algorithm randomly picks one of them.

$$p_1 : type = s \longrightarrow ((note + 7, 7), (note, M))$$
$$p_{1'} : type = s \longrightarrow ((note, M), (note + 5, M))$$

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.

$$\begin{aligned} type = s \longrightarrow choice(& \\ & ((note + 7, 7), (note, M)), \\ & ((note, M), (note + 5, M)) \\ ) & \end{aligned}$$

Parametric variation directly affects the values of parameters and allows non-discrete randomization. A simple mechanism is a non-deterministic function *rand* with a uniform distribution over $[0, 1]$. For example, 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.

As another option, the result of a non-deterministic subgrammar can be calculated 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 Temporal-Scope Grammars

Polyphony in the strictest sense denotes music with multiple independent voices played in parallel. We use it in a broader sense for music that is not *monophonic* (i.e., consisting only of a single voice). This means that notes can overlap, like for example the melody, chords, and bass voice in Figure 3. The score in Figure 3 is actually an example of *homophony*, which lies between monophony and polyphony, but is considered polyphonic under our use of the term. For generating polyphony with a grammar, we have to (1) find a representation of polyphonic structures as a sentence, and (2) handle relationships between entities in the same temporal context.

For (1) we could use a tree-like grouping in the sentence, where child sentences can be arranged either in sequence or in parallel on the timeline. For example, one could use a parallel arrangement of sequential voices like Tanaka and Furukawa [32], or a sequential arrangement of parallel notes for chords like McCormack [11]. For accurately representing

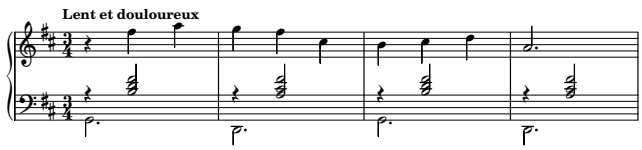

Figure 3. Score representation of measures five to eight of *Gymnopédie No.1* by Eric Satie.

the score in Figure 3, we would need at least two parallel grouping levels, one to model the melody, chords, and bass, and a second one to model the chord notes. A disadvantage of this method is that we need to consider the complete sentence to determine a single entity's position in time.

For (2) we have to consider how shared data propagates through the derivation tree. An entity can only receive information from its parent, so any data that is needed for synchronizing two entities must be provided via a common ancestor. In a polyphonic model two notes that are close in time – and should therefore have common harmonic and metric properties – could be very distant in the tree. This is problematic, as the level of detail in a derivation tree generally increases towards the leaves, but we may already need to determine these details at the root. Tanaka and Furukawa [32] avoid this altogether by using context-sensitive rules that replace notes in multiple places of the tree at once.

Temporal-scope grammars address both points without depending on context-sensitive sentence structure or context-sensitive replacement. First, we associate each entity with an explicit time-span, which decouples its position on the timeline from its position in the sentence. Representing music with time intervals is a well established method and used, for example, in *MusES* [15] or the MIDI format. Second, using sub-grammars and parameters we can propagate complex musical textures downwards in the tree and use the time-span of an entity to locally sample these textures. For example, we can generate a harmonic progression in the root and sample it in the notes of an independently generated melody and bass line.

4.1 Formal Definition

A temporal-scope grammar is context-free and parametric as defined in Sections 3.1 and 3.2. An entity of its vocabulary V is a tuple $(t_0, t_\Delta, x_1, \ldots, x_n)$, where $(t_0, t_\Delta) \in \mathbb{R}^2$ is called the *time-span*, and $(x_1, \ldots, x_n) \in X_1 \times \ldots \times X_n$ are custom parameters, for example pitch information, dynamics, or semantic labels. The time-span encodes the *temporal scope* of an entity as a point in time t_0 and a duration t_Δ. While the use of a duration parameter is already very common in similar systems, the absolute offset t_0 is usually encoded implicitly in the sentence structure. Instead of t_Δ, a second offset $t_1 = t_0 + t_\Delta$ could be used to define the scope, but the former is more convenient in practice.

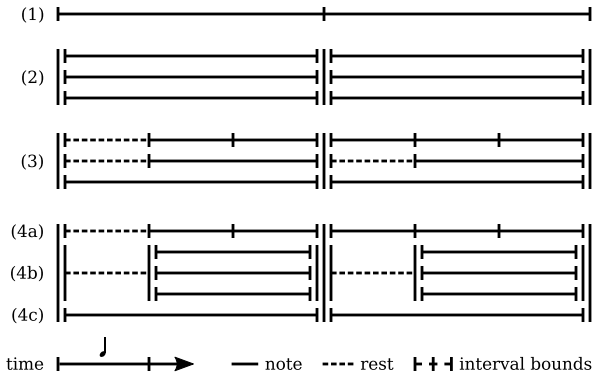

Figure 4. A step by step construction of the temporal structure of measures 5 and 6 of *Gymnopédie No.1* by Eric Satie.

Since the time-span is passed as a parameter, we can generate arbitrary interval arrangements, both absolute and relative, 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 transformations 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. Even though rests are shown in Figure 4, they do not need to be represented in the sentence since entities are completely independent.

A temporal-scope grammar may further have any number of parameters defined over the set of sentences V^*. We can use them to link entities and propagate sub-grammar results through the derivation tree as explained in Section 3.3. Naturally, the grammar must provide a mechanism for retrieving entities from a sentence, for example filtering based on overlap with a time-span. In the next section, we define concrete operators for generating and querying temporal structures.

4.2 Scope Operators

Changing t_0 on the RHS of a rule moves the entity, while changing t_Δ resizes it. Yet, simply arranging notes sequentially is now more work compared to purely monophonic models, as we have to specify both durations and offsets. Similar to the + operator for notes in Section 3.2, we can define higher-level operators for common use-cases:

- The *repeat* operator fills the scope with n time-spans of a fixed duration Δ. We calculate $n = \lfloor t_\Delta / \Delta \rfloor$ to generate zero or more entities that do not exceed the original duration t_Δ. An optional weighting factor distributes the remaining space. Alternatively, we can calculate $n = \lceil t_\Delta / \Delta \rceil$ to generate at least one entity, where the last entity potentially exceeds the original scope.
- The *split* operator divides the scope into n intervals with durations $\Delta_1 \ldots \Delta_n$. Additionally, one can specify

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

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

$\omega_1 \dots \omega_n$ as weights for the distribution of any remaining space. Unlike the repeat operator, split can be used to generate irregular divisions of an entity.

- The *trim* operator removes any part of an entity that exceeds the scope of the 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. 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.

Scope operators accept relative as well as absolute duration notation. The latter requires a synchronized system of units described in the following section.

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 this in our system with three parameters. (1) 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). (2) The beat count parameter $N \in \mathbb{R}_+$ specifies the number of beats per measure. Finally, (3) 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 conversion factors of various temporal units that can be derived from these parameters in Table 1.

Temporal scopes are also well suited for modelling advanced musical phenomena. Polyrhythm and polymeter can be achieved by desynchronizing the respective parameters across the derivation tree. Gradual tempo changes such as *ritard.* or *accel.* work independently of the meter, but require durations and time offsets of the notes to be adjusted based on the desired acceleration curve.

5 Examples

We demonstrate the theoretic constructs by defining multiple polyphonic music generators of increasing complexity. As stated in Section 3, the development of a particular generator is not the goal of this work. Due to spatial constraints, we cannot provide a complete specification of the generators and will instead confine the detailed discussion to a handful of interesting sub-grammars, briefly showcasing possible applications of the theory. The full specification of the generator in Section 5.1 is available in the online data repository linked in Section 7.

The complexity of the presented generators can be compared in Table 2. Individual writing styles produce different structures of both the grammar and the wrapping DSL, so the listed metrics give just rough estimates. For example, the *Waltz* grammar consists of rules with twice as many commands on average as the *Fragments* grammars. At the same time, *Waltz* uses on average almost five times more data per rule. This is due to heavy use of helper structures and native TypeScript functionality. As the *Fragments* grammars are less constrained than *Waltz*, they produce a higher density of notes with more overlaps despite the lower complexity of the grammar definition.

5.1 Generator: Fragments

The first generator produces musical fragments for a small ensemble of voices. There is no higher-level organization in terms of structure, and it is largely randomized with minimal synchronization. It consists of the following components:

- The *piece* grammar serves as the entry point. It first randomizes global parameters such as *tempo*, *beats*, *beatType* and *key*, then uses the grammars listed below to structure and generate a chord progression that is propagated to the voice layers.
- The *layer* grammar generates a binary tree that sets a randomization strategy for voice parts. We can control the degree of repetition within a voice with the *monotony* attribute and the degree of synchronization between voices with the *diversity* attribute. The *depth* of the binary tree dictates the total length of the song, as a leaf equals one measure.
- The *progression* grammar assigns a random chord from a chord pool to the leaves of a layer. We only evaluate it once per piece and pass the result as a shared parameter to each voice.
- The *voices* grammar defines the voice layers: two *lead* voices (violin and flute), one *pad* voice (piano), one

bass voice (double bass), and a variable number of *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. They frequently use the repeat, split, and trim operators to fill measures with melodic, rhythmic, and harmonic patterns.

The following examples discuss some of the sub-grammars in more detail.

5.2 Grammar: Piece

The *piece* grammar generates a complete piece with multiple voices. The initialization of global parameters such as *tempo* is not shown here. First, we define a set of attributes H that carry harmonic information. Next, we define the *progression* grammar, which uses *layer* to generate a sequence of measures, each with a random chord from a chord pool $\{c_1, c_2, \ldots\}$. The *piece* grammar uses angle brackets to apply commands $f_1 \ldots f_6$ sequentially (see Section 3.3).

$$
\begin{aligned}
&H = \{harmony, key, mode, chord\}\\
&progression : \langle layer, choice(c_1, c_2, \ldots)\rangle\\
&piece : \langle\\
&\quad f_1 : set_{span}(2^{depth} * measure),\\
&\quad f_2 : setx_{chords}(progression),\\
&\quad f_3 : [\langle set_{motif}(pad), set_{play}('piano')\rangle,\\
&\qquad \langle set_{motif}(bass), set_{play}('doublebass')\rangle,\\
&\qquad \langle set_{motif}(lead), set_{play}('violin')\rangle],\\
&\quad f_4 : layer,\\
&\quad f_5 : append(\langle query(chords), select(H)\rangle),\\
&\quad f_6 : motif\\
&\rangle
\end{aligned}
$$

In f_1 the total duration of the piece is determined based on the *depth* attribute. A harmonic progression is generated and stored in the *chords* attribute in f_2. f_3 uses the square bracket notation to branch into three voices. Each has a *motif* attribute referencing a sub-grammar and a *play* attribute with the instrument name. All voices inherit the chord progression from f_2.

Step f_4 generates measures for each voice, again using the *layer* grammar. Before generating the actual notes, harmonic information from the chord progression must be retrieved for the respective measure. A *query* retrieves entities for the current time-span from the *chords* attribute. Note that we know that f_2 produces a single entity per measure. Then, the *select* function removes all attributes from the entity that are not in H. The *append* function merges the remaining attributes into the input entity. This pattern allows us to retrieve the harmonic attributes from the shared progression while preserving the voice-specific attributes. Finally, f_6 generates the actual notes.

5.3 Grammar: Layer

The *layer* grammar generates a sequence of empty measures with different seeds. The measures are generated as the leaves of a binary tree of a specified *depth*. f_3 starts the recursion, g_1 and h_1 ensure the termination criterion, and h_3 generates the two children for the current node. In order to introduce repetition of measures or phrases, we can use the *seed* function, which creates a new PRNG for the respective subtree based on the specified seed value.

We structurally synchronize layers by using the same *depth* and use them as a harmonic and metric basis for the voices. Seeds of the generated measures are controlled by a pair of attributes. *monotony* is the probability that two siblings are synchronized with the same seed. For example, if the monotony is 1, all measures will end up with the same seed. If the value is somewhere between 0 and 1, repeating patterns will emerge, as shown in Figure 5. This is implemented in steps h_2, h_4, and h_5. *diversity* is used to control how similar layers are to each other. We retain a unique value for each layer in f_1, then synchronize all layers with the global *pieceSeed* attribute. h_6 then randomly reintroduces the entropy retained in f_1, causing the node to diverge from other layers in the piece, as shown in Figure 6.

$$
\begin{aligned}
&layer : \langle\\
&\quad f_1 : set_{layerSeed}(rand()),\\
&\quad f_2 : seed(pieceSeed),\\
&\quad f_3 : derive(\\
&\qquad g_1 : depth > 0 \longrightarrow \langle\\
&\qquad\quad h_1 : set_{depth}(depth - 1),\\
&\qquad\quad h_2 : set_{sync}(rand()),\\
&\qquad\quad h_3 : split([set_{\omega}(1), set_{\omega}(1)]),\\
&\qquad\quad h_4 : \langle monotony < sync \longrightarrow set_{sync}(rand())\rangle,\\
&\qquad\quad h_5 : seed(sync),\\
&\qquad\quad h_6 : pr(diversity, seed(layerSeed + rand()))\\
&\rangle)\rangle
\end{aligned}
$$

5.4 Grammar: Pad

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

Figure 5. Six random results of the *layer* grammar, where the monotony m decreases from top ($m = 1$) to bottom ($m = 0$) in steps of 0.2. The visualization is limited to 16 colors, which causes the last row to show random repetitions.

Figure 6. Six random results of the *layer* grammar with *diversity* = 0.25, demonstrating its effect on the similarity between layers.

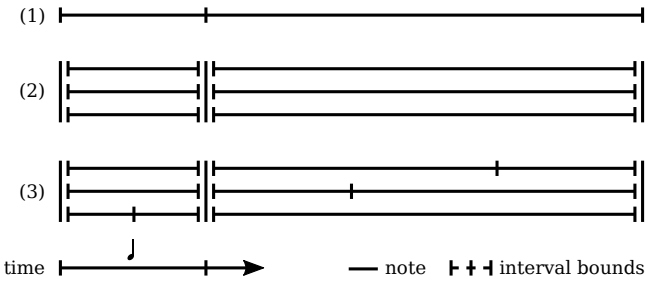

Figure 7. A step by step construction of a possible result of the *pad* grammar.

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 t_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 the angle bracket notation to chain three right-hand sides: f_1 applies *cut* with a probability of 0.5. The *choice* picks a random duration for the fixed part. The square bracket notation in f_2 generates three parallel notes forming a triad 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:

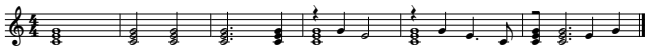

Figure 8. Each measure shows a possible result of the *pad* grammar (File: pad.mp3) using four quarter notes per measure and the C major scale.

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

The *pad* grammar does not use recursion or complex matching criteria. Nevertheless, it demonstrates both sequential and parallel note arrangements and the use of temporal units. Figure 7 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. Figure 8 shows a selection of possible results.

5.5 Generator: Extended Fragments

The second generator builds directly on the previous one. It uses the *layer* grammar on two levels, first to divide the piece into sections, then to divide sections into measures. Both sections and measures can be dropped randomly in particular voices to generate more interesting piece structures. Instruments and motif variety can be configured by the user.

5.6 Generator: Waltz

The third generator is focused on short waltzes. It is aligned with the strategy of the previous ones, but it adds synchronization of various score aspects based on local continuities and global patterns. For example, it features a duo of complementary melodies. Variability is decreased in favor of constraints that limit the selection of rhythms, melodies and harmonies. Efficient stochastic selection under the given constraints is performed by additional utility components that list all possible sequences of:

- waltz *rhythms* for a single measure with a granularity up to a 1/16 note. Filtering of rhythms is accelerated by precomputing additional features like: *energy*, *diversity*, *regularity*. The generator selects a pair of rhythms with contrasting energies that repeat in each musical sentence. Users can introduce further constraints.
- key-agnostic *melodies* with derived features like: *range*, *leaps*, *monotony*, *diversity* for better control of desired melody types. Melodies can also be selected to respect

harmony on strong beats for a given rhythm. The generator also gives preference to melodies ending one diatonic step away from the following harmony to support harmonic development.
- *harmonies* build upon the database of melodies with the degrees interpreted as harmonic functions. We include constraints which select only progressions ending with a certain cadence.

The waltz generator further allows manually overriding some of the random choices, for example harmony progressions. Other parameters, like the rhythm energy, can be limited to intervals or single values.

6 Implementation

We implemented the theoretic framework as a DSL in TypeScript 4.1.2. It is roughly equivalent to the abstract notation introduced in the previous sections. Some extra syntax is required, for example, generic type parameters and type constraints should be used to decouple components from a specific entity definition. Entities are generally immutable. Setting an attribute creates a new descendant entity with the changed value. We copy unchanged attributes, but one could also use a reference to the ancestor if memory is limited.

Since the DSL is embedded in TypeScript, a wide range of functionality like global variables, classes, etc. can be used for efficient specification of grammar instances. Note that while the instances of our formalism, like the generator examples from Section 5, are implemented as TypeScript programs, we still refer to them as *grammars* throughout this work, since the DSL only provides an interface for the underlying grammar-based theory. The TypeScript code below implements the *pad* grammar from Section 5:

```
function pad<V extends Entity>(): Fun<V[], V> {
  return pipe(
    pr(0.5,
       cut(choice<number, V>(half, quarter, eighth))),
    x => fork(
      pipe(),
      set({ note: x.note + 2 }),
      set({ note: x.note + 4 }),
    ),
    pr(0.5,
       cut(choice<number, V>(half, quarter, eighth))),
  );
}
```

For playback we use the Web Audio API, which means that grammars developed with our system work in any modern web-browser. Figure 9 shows our browser-based graphical interface. While the capabilities of a browser-based playback environment are limited to a certain degree, the benefits are significant. Composers can immediately deploy their grammars to a wide range of devices, and users can generate new pieces and listen to them with zero setup.

The generation process can be controlled on two levels. Experts can work directly with the TypeScript code and design grammars, functional components, and define controls for the parameters on the starting entity. Users without deep knowledge of the framework can use these controls in the web-interface to adjust the generated music without touching the code. Parameters exposed in the web-interface can affect the whole piece, like the initial *seed* parameter, but also fully isolated characteristics like the melody of a voice or accents in some beats, while keeping all the rest of the generated piece unchanged.

7 Results

The implementation of the generators described in Section 5 serves to demonstrate the capabilities of our theoretic contribution. Live versions of the generators for try-out and a selection of generated music can be found at:
https://github.com/eibens/farm-2021

We have further conducted an informal experts-study with five participants with backgrounds in musicology or composition. They were instructed to interact with the generators on their own and judge the musical quality of the results. Note that two specialists on classical music only reviewed the Waltz Generator:

Evaluation: Fragments Generator (extended).

- **P1** noted the high variability and ambient style of the results. They found the structure, voicing, and effects to be interesting, but also concluded that the results are obviously generated by a program.
- **P2** praised the interesting structure and contrasts, as well as the use of pauses and repetition – only the melody could be richer. They also suggested to improve the distribution of cadences.
- **P3** thought that the distribution of the drums was sometimes bad and that the harmonic progression sounded arbitrary. But the structure and repetition were received positively.

Evaluation: Waltz Generator.

- **P1** suggested additional form structures, pauses, and dissonance for the melody. They recommended more repetition to make the melody memorable.
- **P2** praised the rhythm and harmonic progression as very fitting to the musical style. They also suggested increasing repetition in the melody and adding options for solos.
- **P3** found the quality of the harmonic progression surprising, and overall judged the results to be good in an almost uncanny way. They criticised the melody, which sometimes leaps in ways that seem atypical for the genre.

Generator	Files size	Rules	Commands	Instruments	Notes	Overlap	Length
Fragments	7 KiB	23	172	4	232	5.5	32 s
Fragments (ext.)	29 KiB	50	413	4	971	8.0	66 s
Waltz	61 KiB	22	360	2	665	3.4	72 s

Table 2. Statistics for the example grammars: definition complexity (middle) and average results complexity (right).

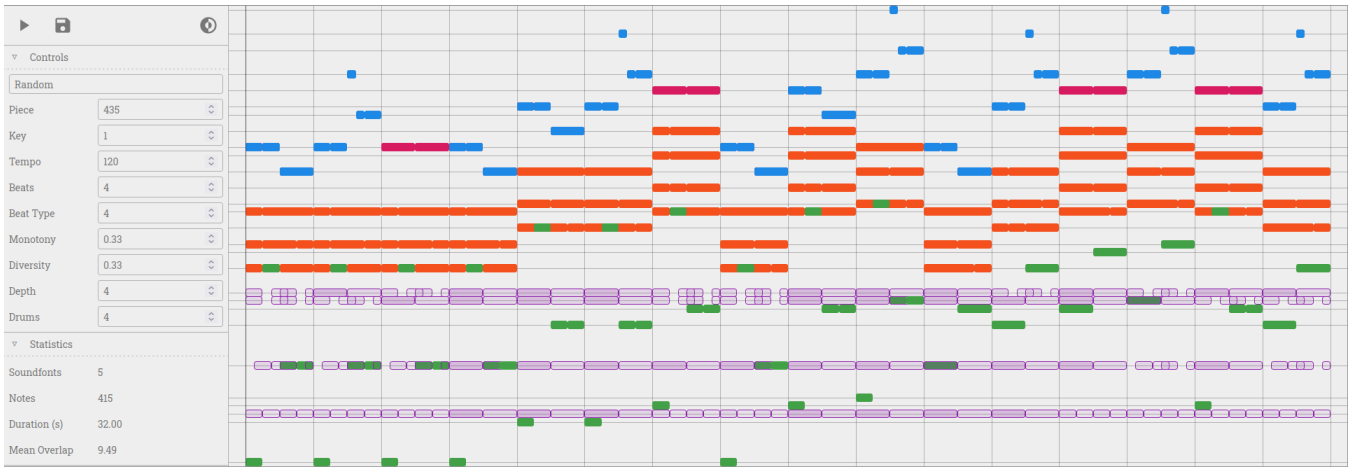

Figure 9. The controls to the left can be used to manipulate parameters on the starting entity. The score visualization to the right shows time on the X axis, logarithmic pitch on the Y axis, and the scope of the terminal entities as colored bars. Notes of the same color belong to the same voice, and the opacity encodes the loudness.

- **P4** praised the harmony, but criticised the melodic movement, which contained too many leaps, as well as the overall structure, which they did not find interesting. They also called for more emotional expression, yet concluding that the generator is a good source of inspiration for beginning composers.
- **P5** approved of the harmony, but noted that sometimes subsequent dissonance occurs without a specific purpose. They liked the subtle use of dynamics and encouraged to further increase local dynamic contrasts. Fine-grained control with parameters uncommon in standard music theory and analysis was very interesting for them, along with the possibility to adjust features of an already generated piece.

All impressions were positive, praising mainly the harmonic soundness and rich rhythms. Suggestions for improvements were mostly targeted towards specific features of the generators – better structuring of Waltzes, less randomness for Fragments. Suggestions for better handling of melodies point to the underlying framework with its strict tree-like data flow. To get information about the surrounding entities, it must be pre-computed and stored in attributes early in the derivation. For complex pieces, direct horizontal and vertical linking of neighboring entities would provide even more efficient means for temporal smoothing of melodies.

A coherent seed in combination with parametric adjustments enabled all experts to tweak the generated song on different levels, but at the same time most of them felt overwhelmed by the large number of parameters. As user interface design and UX was not subject of our research, we are sure that these aspects can be greatly improved.

8 Conclusion

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

Our model of time appears to be capable of expressing a wide range of temporal structures. 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 simply discard an entity without affecting its surrounding.

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. However, after moving an entity one should consider querying its new context.

Finally, synchronizing multiple voices to one or more musical textures 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 polyphonic structures are only expanded in a post-processing step.

Future Work. The selection of scope-based operators that the system provides out of the box is still limited. Additional operations on entity sequences could be used for more powerful effects. As an alternative to random number generators, one could integrate user input directly into the derivation process. This would allow users to assume manual control over any aspect of the generation.

References

[1] Torsten Anders and Eduardo R Miranda. 2011. Constraint programming systems for modeling music theories and composition. *Comput. Surveys* 43, 4 (2011), 1–38. https://doi.org/10.1145/1978802.1978809

[2] Noam Chomsky. 1956. Three models for the description of language. *IRE Trans. on Inf. Theory* 2, 3 (1956), 113–124. https://doi.org/10.1109/tit.1956.1056813

[3] W Bas De Haas, Martin Rohrmeier, Remco C Veltkamp, and Frans Wiering. 2009. Modeling harmonic similarity using a generative grammar of tonal harmony. In *Proc. of the* 10^{th} *Int. Conf. on Music Inf. Retrieval.*

[4] Jose D Fernández and Francisco Vico. 2013. AI methods in algorithmic composition: A comprehensive survey. *Journal of Artificial Intelligence Research* 48 (2013), 513–582. https://doi.org/10.1613/jair.3908

[5] Édouard Gilbert and Darrell Conklin. 2007. A probabilistic context-free grammar for melodic reduction. In *Proc. of the Int. Workshop on Artificial Intelligence and Music,* 20^{th} *Int. Joint Conf. on Artificial Intelligence (IJCAI), Hyderabad, India.* 83–94.

[6] Mathieu Giraud and Slawek Staworko. 2015. Modeling musical structure with parametric grammars. In *Mathematics and Computation in Music.* Springer, 85–96. https://doi.org/10.1007/978-3-319-20603-5_8

[7] SR Holtzman. 1981. Using generative grammars for music composition. *Computer Music J.* 5, 1 (1981), 51–64. https://doi.org/10.2307/3679694

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

[9] Donald E Knuth. 1990. The genesis of attribute grammars. In *Attribute Grammars and Their Applications.* Springer, 1–12. https://doi.org/10.1007/3-540-53101-7_1

[10] Fred Lerdahl, Ray Jackendoff, et al. 1983. *A generative theory of tonal music.* Vol. 1996. The MIT Press.

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

[12] Orestis Melkonian. 2019. Music as language: putting probabilistic temporal graph grammars to good use. In *Proc. of the* 7^{th} *ACM SIGPLAN Int. Workshop.* 1–10. https://doi.org/10.1145/3331543.3342576

[13] Pascal Müller, Peter Wonka, Simon Haegler, Andreas Ulmer, and Luc Van Gool. 2006. Procedural modeling of buildings. *ACM Trans. on Graph.* 25, 3 (2006), 614–623. https://doi.org/10.1145/1179352.1141931

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

[15] François Pachet, Geber Ramalho, and Jean Carrive. 1996. Representing temporal musical objects and reasoning in the MusES system. *Journal of new music research* 25, 3 (1996), 252–275. https://doi.org/10.1080/09298219608570707

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

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

[18] Donya Quick. 2015. Composing with kulitta. In *Proc. of the Int. Computer Music Conf.*

[19] Donya Quick and Paul Hudak. 2013. Grammar-based automated music composition in Haskell. In *Proc. of the* 1^{st} *ACM SIGPLAN workshop on Functional art, music, modeling & design.* ACM, 59–70. https://doi.org/10.1145/2505341.2505345

[20] Donya Quick and Paul Hudak. 2013. A temporal generative graph grammar for harmonic and metrical structure. In *Proc. of the Int. Computer Music Conf.*

[21] Curtis Roads and Paul Wieneke. 1979. Grammars as representations for music. *Computer Music J.* (1979), 48–55. https://doi.org/10.2307/3679756

[22] Martin Rohrmeier. 2007. A generative grammar approach to diatonic harmonic structure. In *Proc. of the* 4^{th} *Sound and music computing conf.* 97–100.

[23] Martin Rohrmeier. 2011. Towards a generative syntax of tonal harmony. *Journal of Mathematics and Music* 5, 1 (2011), 35–53. https://doi.org/10.1080/17459737.2011.573676

[24] Pierre Roy and François Pachet. 1997. Reifying constraint satisfaction in Smalltalk. *Journal of Object-Oriented Prog.* 10, 4 (1997), 43–51.

[25] Camilo Rueda, Magnus Lindberg, Mikael Laurson, Georges Bloch, and Gerard Assayag. 1998. Integrating constraint programming in visual musical composition languages. In *Proc. of the Workshop on Constraints for Artistic Applications (ECAI'98).*

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

[27] Michael Schwarz and Pascal Müller. 2015. Advanced procedural modeling of architecture. *ACM Trans. on Graph.* 34, 4 (2015), 1–12.

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

[29] Mark J Steedman. 1984. A generative grammar for jazz chord sequences. *Music Perception* 2, 1 (1984), 52–77. https://doi.org/10.2307/40285282

[30] George Stiny. 1980. Introduction to shape and shape grammars. *Environment and Planning B: Planning and Design* 7, 3 (1980), 343–351. https://doi.org/10.1068/b070343

[31] George Stiny. 1982. Spatial relations and grammars. *Environment and Planning B: Planning and Design* 9, 1 (1982), 113–114. https://doi.org/10.1068/b090113

[32] Tsubasa Tanaka and Kiyoshi Furukawa. 2012. Automatic Melodic Grammar Generation for Polyphonic Music Using a Classifier System. In *Proc. of the* 9^{th} *Sound and Music Computing Conf.*

[33] Peter Wonka, Michael Wimmer, François Sillion, and William Ribarsky. 2003. Instant architecture. *ACM Trans. on Graph.* 22, 3 (2003), 669–677. https://doi.org/10.1145/1201775.882324

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