CONSTRAINED MUSIC GENERATION USING MODEL-CHECKING

Théis Bazin, Shlomo Dubnov

CONSTRAINED MUSIC GENERATION USING MODEL-CHECKING

Théis Bazin[1,2] and Shlomo Dubnov[3]

[1]*Département Informatique*, ENS Cachan
[2]IRCAM, UMR STMS 9912 CNRS, Sorbonne Universités UPMC
[3]CREL, University of California San Diego

ABSTRACT

Numerous works have tackled the problem of style modeling: learn some implicit notion of style from an ensemble of musical sequences and generate new content resembling this set of examples. This paper deals with the additional problem of introducing high-level constraints into such applications, that is, structured generation: generate new music in the style of existing musical data, satisfying a given temporal structure.

We propose an original approach to solve this issue through the use of model-checking. It is applied here to the Factor Oracle, an automaton that has already been used by composers for music generation, but currently lacks high-level control capacities and therefore structure. FO captures the sequential structure of symbolic or signal-level musical inputs. Following an abstraction of the automaton structure, the specifications are converted to temporal-logic formulae and solutions are efficiently searched for by means of external model-checking tools. This search is done in a backward manner, allowing for the retrieval of optimal solutions.

A Python implementation of the system and musical examples, e.g. for the extraction of chord sequences from a piece, are provided.

1. INTRODUCTION

In constraint systems, it sometimes turns out that a little dose of creativity can go a long way, providing an ability to come up with unexpected answers to the problem at hand.

Such concerns arise for instance in the field of robotics, when automatically generating movement schemes for a surveillance robot. Adding partial improvisation to the generation mechanism instead of only following the programmed patterns gives flexibility and robustness with respect to the outside environment, which might be of adversarial nature. A practical case of environmental unpredictability is the management of crisis situations, for example during an earthquake, when the environment temporarily strongly deviates from its usual conditions, as described in Kendra and Wachtendorf [13].

These techniques fall under the general problem of control improvisation: given a system (e.g. the robot) and a set of commands (e.g. the motion patterns) as given by an external operator, generate new, valid commands of the systems. This problem finds a fruitful instantiation within the field of music informatics, where improvisation and creativity are at the core of musical composition and performance, under the generic name of *style modeling*: generating new music "in the style" of some given input material.

The approach applied in the following blends algorithms and formal language techniques for the automated analysis of music, through the use of automaton-based systems, a popular solution in machine-improvisation approaches: the *Factor Oracle* (FO) [1] is a compact means of representing sequential relations within a musical piece and provides an efficient way of generating new similar pieces based on given original material. In short, the FO is an automaton, a variant of suffix automata, recognizing all sub-sequences of a given input text. Its underlying graph structure makes it usable to generate new music by moving along its edges in a non-deterministic way, producing output resembling the input text use to build the automaton, all thanks to the tight formal structure of the automaton. An extension of FO exists, named *Variable Markov Oracle* (VMO), introduced by Wang and Dubnov [23], which operates on generic data-types through a process of symbolization.

The FO has already been applied to the problem of style modeling within the OMax system by Assayag et al. [3], but it does not yet constitute a completely satisfactory answer to the problem of structured improvisation. Indeed, the VMO's inherent non-deterministic way of generating music, using only a local scope, induces a lack of global structure in its output: no general themes arise, no salient repetitions or cues occur... In other words, the contents produced by the VMO are locally meaningful and resemble the original material but, in a test similar to the Turing-test [2], its lack of global constraints makes it easy for a human ear to determine that the content was generated by a machine and not by a musician [1]. For references to reviews of structure in music, readers are referred to the book by Nattiez [15] and the article by Deliège [5].

The present work aims at tackling this lack of global structure by providing means of controlling the oracle's output. This is done using a verification-based approach: the musical output is constrained by adding a logical layer to the FO (and its extension, the VMO), which is then

[1] Currently, an average time of 1 minute has been shown to be necessary for listeners to complete this task

model-checked via external tools. To the authors' knowledge, such a model-checking approach has not yet been applied to the generation of structured music and therefore paves the way for future works: the developments of the theoretical aspects (defining appropriate logics) as well as the practical tools (leveraging efficient model-checking engines for these logics). Note that the model-checking techniques presented in the following are generic and only very slightly dependent (as will be seen in the development) on the structure of FO / VMO. Thus, any other automaton for music generation could be plugged into the system with little effort. FO and VMO are used here because they experimentally gave good results in prior experiments in terms of capturing style and creativity [3, 22, 16].

1.1. Related works

Several other approaches exist for the computerized generation of music, which can be roughly sorted into two main categories:

Statistical machine-learning approaches

A first group makes use of statistical machine learning techniques, such as applied within Microsoft Research's Songsmith [20]. These approaches are only as good as the set of examples used is meaningful. Moreover and more critically, they fail at providing any insights into the underlying structure of the models built, as is often the case with pure statistical machine learning approaches. As such, they don't shed any light into the structure of the music they aim at emulating. More recently though, the project MorpheuS [2] [7] takes on the same approaches with added elements of harmony theory, using more advanced machine-learning techniques than those applied for Songsmith.

Formal methods

The second group, to which the methods applied in this paper belong, leverages formal methods. These require to some extent a formal model of music and as such are more capable of providing new insights into the structure of music and what makes it interesting – or uninteresting! Within this field, the work on grammatic structures of Lerdahl and Jackendoff [14] is significant: their *Generative Theory of Tonal Music* introduces a generative grammar incorporating elements of cognitive science aiming at reproducing the way a listener unfolds and understands the musical structure of a work. Such grammars could be used in a probabilistic way to automatically generate music that is correct with respect, say, to the rules of Western tonal harmony, and are a very useful way of automatically evaluating the "well-formedness" of generated music, but they don't provide a straightforward way of generating music following external constraints, e.g. resembling that of a given composer.

[2] Hybrid machine learning – optimization techniques to generate structured music through morphing and fusion.

The approach proposed in this article has the benefit of being founded in structure analysis but at the same time providing a convenient way of generating new content, drawing from this analysis.

In that sense, models based on probabilistic automata (e.g. Markov chains) have been developed. Among these, the works by Pachet and Roy [18] and by Eigenfeldt [9] on constrained generation using Markov chains can be mentioned. Pachet's approach is based on *Elementary Markov Constraints* (EMCs), which incorporate constraints within the standard Markov chain random-walk algorithm. Their structures are Markov models which replicate the conditional continuation probabilities of the learned corpus and simultaneously try to satisfy some additional constraints. These constraints aim at optimizing a given cost function such as the generated sequence's length, and as such are somewhat complementary to the approach studied in this paper.

Closer to the techniques applied here, an approach based on adding control layers on top of the FO has been undertaken by Donze et al. [6], under the name of *control improvisation*, but uses parallel products of automata to add constraints to the transition systems rather than model-checking them. Both approaches are therefore dual to some extent: control improvisation enforces correctness of the generated solutions by discarding unsatisfactory paths on the FO through parallel products, whilst in the model-checking framework no transitions are removed and the modification is on the way the automaton is run on to generate new content, so that it only generates satisfactory sequences. Indeed, in the parallel product approach proposed in their article, the constraining is done directly on the automaton's very structure, by removing unwanted transitions. Thus, adding musical content to the FO requires a recalculation of the whole constrained structure, with a quadratic computation cost. In the model-checking approach, the constraining is external to the FO, it is done within a logical layer and an external model-checking tool is used to run on the FO and check if the requested property is satisfied. Consequently, if one wishes to add content to the FO – as is the case when it is built on-line during a live performance –, then one only needs to update the model and not the logical layer, which can be done efficiently on-line (in linear time in the number of states in the automaton). Note that both methods have already been studied in conjunction by Ziller and Schneider [24] outside from the scope of music. The results of this work could be applied for further developments of the techniques presented in this paper.

Lastly, recent work on ImproteK by Nika et al. [17] should be noted. Their approach is based on symbolic *scenarios* (e.g. chord progression) which describe the desired long-term evolution of the generated music, as well as dynamic calls to a reactive engine which allow to use the generation model in a real-time context. Given such a dynamic scenario and a sequential representation of the musical examples, sequences satisfying the scenario are looked for by means of string matching and forward search, as opposed to the backward search inherent to model-checking algorithms. This makes for a more straightforward extension to real-time applications in their system, at the cost of sub-optimality in the generated sequences.

2. RESEARCH BACKGROUND: FROM FO TO VMO

This paper only offers a very condensed overview of the underlying tools and interested readers are referred to the related papers in the references [1, 8, 23, 22].

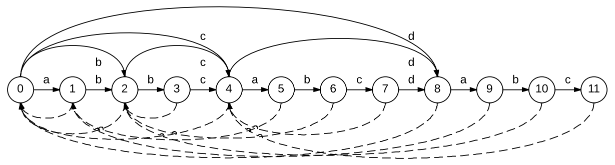

Figure 1. Example of an oracle structure for sequence $\mathcal{S}$ = "abbcabcdabc"

2.1. Factor Oracle

At the basis of the tools applied in the following, the Factor Oracle is a variant of suffix automata with an added notion of creativity. It has already been applied with success in music improvisation contexts. FO focuses on extracting repeated subsequences within a given *symbolic* sequence, and does so in a incremental fashion with an automaton that can be built on-line in linear time and space – indeed, the automaton has $T + 1$ states for an input sequence of length T. The practical algorithms for building FO are presented in the paper by Allauzen, Crochemore, and Raffinot [1], only the most useful and significant properties that are maintained during this creation are introduced here.

An example of FO is given on Figure 1. In FO, each state has labelled *forward links* and a link to its successor. Querying FO with a sequence of labels is done by following the forward links according to the query, which allows to retrieve the position of a matching sequence from the automaton. Finally, for each state q_k in the automaton, there is a unique unlabelled link to the state q_p with $q_p < q_k$ such that the longest repeated suffix of the sequence $\alpha_1 \cdot \alpha_k$ is recognized in state q_p (or to the initial state in case no such suffix exists). These links are called *suffix links* – the dotted links on Figure 1 – and entail the FO's creativity: using these transparent links – recall that they are unlabelled –, new sequences can be generated from $\mathcal{A}$. For example, the automaton from figure 1 can generate the sequence $aaabb$, which is not a sub-sequence of the original sequence $\mathcal{S}$.

The main problem in using FO is the following: how to actually turn a general musical data stream, such as an audio file or a complex musical sequence (e.g. a polyphonic MIDI file) into a symbolic sequence for FO to work with? This is the purpose of the VMO.

2.2. Variable Markov Oracle

The Variable Markov Oracle can be seen as a tool that produces a standard Factor Oracle representation of musical structure with an added layer of symbolization, allowing to work with completely generic data-types. It is presented in Wang and Dubnov [23] with an application to 3D-gesture clustering and query-matching.

The symbolization is done as follows: start from a discretized sequence of observations $O[n]$ with $O : \mathbb{N} \to \mathcal{T}$ and simply suppose provided a distance δ comparing observations in O, i.e. $\delta : \mathcal{T} \times \mathcal{T} \to \mathbb{R}$. This is highly generic and does not only apply to music: for instance O could be a sequence of colors and δ the Euclidean distance on the associated RGB vectors. Then, a threshold $\theta \in \mathbb{R}_+^\star$ is fixed and the iterative algorithm for the construction of Factor Oracle is launched on the sequence of observations $O[n]$ with the added step of turning each observation $O[k]$ into a symbol $\alpha(O[k])$, which in turn leads to a new state in the constructed automaton.

For this symbolization, the following two properties are maintained, where $b(\alpha)$ denotes the *symbolic cluster* associated to symbol α, that is, the set of observations $O_{i_1}, \ldots O_{i_{|b(\alpha)|}}$ labelled by symbol α during the process:

1. Any given state shares the same symbol as the state to which its suffix link points.

2. For any symbol α and for any (if they exist) two observations $O[start]$ and $O[end]$ in $b(\alpha)$, $O[start]$ and $O[end]$ can be connected by a path in $b(\alpha)$ where any two successive states are distant by at most θ, i.e. $O[start]$ and $O[end]$ are *not necessarily* distant by at most θ but can be smoothly connected within $b(\alpha)$ with respect to δ.

On the level of the automaton, the symbolization step therefore maps states (or equivalently observations) into congruence classes up to θ where states sharing the same symbol all share a common suffix in the observation sequence (up to the precision θ). With this property, and supposing that the chosen distances and thresholds are meaningful with regards to the actual content of the observations, the Factor Oracle can then generate improvisation sequences which make sense on the symbolic level as well as on the concrete-data level.

Improvising with the VMO Regarding the actual process of generating new content from an existing VMO, previous experiments consisted in launching a purely random walk on the automaton and outputting, as is often done when synthesizing new content from an automaton, for each symbol α seen on a link during this walk, one of the observations $O[n_\alpha]$ in the set $b(\alpha)$. Note that the unlabelled suffix links are therefore completely transparent and allow for silent jumps within a given musical context.

This method is effective, yet inherently randomized and limited to offering only a local scope, it therefore necessarily lacks a notion of global structure. To tackle this, another control method different from the approach presented in this article was already proposed for the VMO. This method is called *guided improvisation* [22] and operates by generating a path "resembling" a provided input example.

Let V a VMO, built on some material of a given datatype $\mathcal{T}$. Let R a *request*, i.e. a sequence of data also of type $\mathcal{T}$. For guided improvisation, the generation is also done by a walk on the automaton, but instead of a purely non-deterministic walk, the algorithm also runs through R in parallel with V and each random choice of a link to follow is replaced by a choice maximizing the proximity of the sequence generated with the request R. This first improvement thus aims at creating output complying with the VMO's structure whilst also being close in content to the input request R. This method can be seen as a low-level constraining system. This paper instead follows the opposite road and goes for a higher-level, more flexible and generic approach, based on logics and abstraction.

Additionally, it should be noted that in guided improvisation the "guiding" sequence must be of the same datatype and have the same temporal granularity as the original data used to construct the oracle. In the approach proposed here, the logical specification can be done using different representations of the sequence, thus capturing different aspects of music than the one used to derive the oracle itself (e.g., the oracle can be constructed using melodic considerations, whilst the logical specification refers to harmony).

More critically, *guided improvisation* is a *forward, greedy* algorithm, making only local decisions as it progresses through the request, and as such may fail at extracting the optimal sequence following a given request. The machine-learning approach proposed here works in a *backwards* way, building the example sequence from the end (e.g. to generate a sequence starting on the chord C and ending in G, it starts by looking for G chords, then starting from those, looks for states with the chord C leading to those G chords) and returning a result only if a sequence exactly satisfying the whole request is found (i.e., as will be seen, if it exists), i.e. an *optimal* example.

3. TECHNICAL BACKGROUND

Prior to introducing the proposed solution to the problem of structured music generation using the VMO, the main logical tool at stake is introduced: CTL, a temporal logic.

3.1. CTL

Only a high-level introduction to CTL is given here, interested readers are referred to the introductory paper by Emerson and Halpern [10].

Definition 1 (Transition system)**.** A *transition system* (or *Kripke structure*) is a tuple $M = (S, T, I, \mathrm{AP}, \ell)$ such that

- $S = \{s_1, s_2, \dots\}$ is a set of states (finite or infinite),
- $T \subseteq S \times S$ is a set of transitions,
- $I \subseteq S$ is a set of initial states,
- $\mathrm{AP} = \{p, q, \dots\}$ is a set of atomic propositions,
- $\ell : S \to 2^{\mathrm{AP}}$ is a labeling function.

In the following, only *infinite paths* are considered, which is only valid under the assumption that the transition relation T is *non-blocking*, ie. $\forall s \in S, \exists\, s' \in S, (s, s') \in T$.

Remark.

1. The labeling function can be interpreted as a truth mapping: for a state $s \in S$ and an atomic property $p \in \mathrm{AP}$, $p \in \ell(s)$ if and only if p is true in the state s.
2. For $s, s' \in S$, this paper follows standard conventions with the notations $s \to s'$ if $(s, s') \in T$ and $s \rightsquigarrow s'$ if there exists a path in M starting from s and ending in s'.

Definition 2 (Successor)**.** Given a Kripke structure $M = (S, T, I, \mathrm{AP}, \ell)$ and a state s of M, a *successor* of s is any state s' in S such that $s \to s'$.

CTL (Computation Tree Logic) is a logic expressing properties on Kripke structures. Its main characteristic – and the reason why it was chosen in this paper – is the ability to quantify existentially and universally on branching paths, allowing for an extensive characterization of non-deterministic systems, making it appropriate for the oracle, an inherently branching system.

Definition 3 (Syntax of CTL)**.** A CTL formula is a formula derived from the following grammar:

$$\begin{aligned}\varphi, \psi ::= &\top \mid p \mid \neg\,\varphi \mid \varphi \vee \psi \mid \mathsf{EX}\,\varphi \mid \mathsf{E}\,(\varphi\,\mathsf{U}\,\psi) \mid \dots\\ &\mathsf{AX}\,\varphi \mid \mathsf{A}\,(\varphi\,\mathsf{U}\,\psi)\end{aligned}$$

where p is an atomic proposition.

Definition 4 (Semantics of CTL, satisfaction relation)**.** Let $M = (S, T, I, \mathrm{AP}, \ell)$ a Kripke structure and a state s of M. Let φ a CTL property.
The *satisfaction relation* ($\vDash$), which states, when applied as follows:

$$(M, s \vDash \varphi)$$

that the Kripke structure M *satisfies* the property φ in the state s, is inductively defined as indicated on Table 1 (considering only infinite paths).

Remark (Intuitions)*.* The names of the operators in CTL can be understood as follows:

- E stands for *exists*,
- A stands for *all*,
- X stands for *next*,
- U stands for *until*.

$M, s \vDash \top$	always
$M, s \vDash p$	if $p \in \ell(s)$
$M, s \vDash \neg\, \varphi$	if $M, s \nvDash \varphi$
$M, s \vDash \varphi \vee \psi$	if either $M, s \vDash \varphi$ or $M, s \vDash \psi$
$M, s \vDash \mathsf{EX}\, \varphi$	if there exists a successor s' of s such that $(M, s' \vDash \varphi)$
$M, s \vDash \mathsf{E}\,(\varphi\, \mathsf{U}\, \psi)$	if there exists a path $s_0, s_1, \ldots$ in M with $(s = s_0)$ such that one of the following holds: • there exists k in $\mathbb{N}$ such that: $(M, s_k \vDash \psi)$ and $\forall\, 0 \leq i < k : (M, s_i \vDash \varphi)$ or • for all i in $\mathbb{N}$: $(M, s_i \vDash \varphi)$
$M, s \vDash \mathsf{AX}\, \varphi$	if for all successor s' of s in M, $(M, s' \vDash \varphi)$
$M, s \vDash \mathsf{A}\,(\varphi\, \mathsf{U}\, \psi)$	if for all path $s_0, s_1, \ldots$ in M with $(s = s_0)$, one of the following holds: • there exists k in $\mathbb{N}$ such that: $(M, s_k \vDash \psi)$ and $\forall\, 0 \leq i < k : (M, s_i \vDash \varphi)$ or • for all i in $\mathbb{N}$: $(M, s_i \vDash \varphi)$

Table 1. Semantics of CTL

Definition 5 (CTL model-checking)**.** CTL model-checking can be considered at two-levels: the state level and the global, system-wide level:

- The *model-checking* of a property φ on the model M in the state s is the evaluation of the satisfaction relation $M, s \vDash \varphi$.
- The *global* model-checking of a property φ on a model $M = (S, T, I, \mathrm{AP}, \ell)$, denoted as the satisfaction problem $M \vDash \varphi$ is the conjunction of the model-checking over all initial states of the structure:

$$\bigwedge_{s_i \in I} (M, s_i \vDash \varphi)$$

The model-checking problem $M \vDash \varphi$ is in P: it can be solved in time $O(|M| \cdot |\varphi|)$. The basic techniques for model-checking use standard reachability algorithms, but optimized model-checkers use more advanced techniques to reduce the memory and time costs of the model-checking.

Remark (Additional CTL syntax)*.* Some useful symbols can be defined on top of CTL's core syntax, with semantics directly inferred from definition 4. These additional symbols include:

- EF, standing for *exists finally*, with the following equivalence:

$$\forall\, M, s, \forall\, \varphi,\, (M, s \vDash \mathsf{EF}\, \varphi) \iff (M, s \vDash \mathsf{E}\,(\top\, \mathsf{U}\, \varphi))$$

- EG, standing for *exists globally*, with the following equivalence:

$$\forall\, M, s, \forall\, \varphi,\, (M, s \vDash \mathsf{EG}\, \varphi) \iff (M, s \vDash \mathsf{E}\,(\varphi\, \mathsf{U}\, \bot))$$

The universal counterparts to those operators (AF and AG) exist and are also being employed.

- CTL also entails a notion of causality, and the associated *leadsto* operator $\rightsquigarrow_{leadsto}$ can be defined as follows:

$$\forall\, M, s, \forall\, \varphi,\, \psi, (M, s \vDash \varphi \rightsquigarrow_{leadsto} \psi) \iff (M, s \vDash \mathsf{AG}\,(\varphi \Rightarrow \mathsf{AF}\, \psi))$$

where the implication $\Rightarrow$ is defined in CTL as it is in boolean logic:

$$(a \Rightarrow b) \iff (\neg a \vee b)$$

4. PROPOSED SOLUTION: MODEL-CHECKING THE VMO

In order to tackle the lack of structure of the VMO's native random output, the use of model-checking techniques is proposed. The approach abstracts the oracle into a Kripke structure replicating the automaton's graph-structure. The atomic properties used currently describe the harmonic content of the fragments held in each state.

Because each state in the VMO (except the last) has at least two distinct successors – reached using either the direct link to its successor or the link to its associated suffix state –, the transition system obtained has an inherent branching nature. In order to correctly capture this, CTL model-checking is applied to the models yielded by the transcription and obtain properties on the VMO from there.

First, the current, fairly simple logic used to express properties on the music being generated by the VMO is presented. An application of this logic is also introduced, which provides a means of morphing two distinct music pieces via a CTL formula (think extracting an aria of Handel's *La Resurrezione* from Bach's *Mass in B minor*). Then the way in which the VMO is abstracted into a transition system is described – a straightforward process given the automaton structure of the oracle. Finally, a note on the associated implementation is made and the model-checker currently under use for the application is mentioned as well as some reasons for this choice.

4.1. A logic of music

This section introduces the way in which musical properties on the oracle are expressed.

4.1.1. From tonal theory to CTL formulae: atomic formulae

As illustrated by the numerous works mentioned in the introduction trying to provide precise formal models of music, describing musical content and its perception by a listener is no easy task. In order to be able to prototype the system and in an Occam's Razor-approach, the logic currently used is only a very rough description of the musical properties of the system. The general ideas underlying this logic are as follows: as usual with temporal logics, the atomic properties in the logic have a local scope and CTL is used to extend those to properties with a global scope. In order to be able to control the harmonic content of the music generated, a *local descriptor of harmony* is therefore required. A possibility is to use *tonal roots* to abstract chords, with the tonal roots estimated by detecting for a given chord the pitch within it which holds the most thirds stacked on it, a simple yet often on point method[3].

The possibility to express constraints on melodic motion between consecutive fragments is also included within the logic language. It is a simple yet effective addition to the expressiveness and constraining power of the logic and is done via a variable called $motion$.

In more details, the systems considered are composed of a number of *frames* of music, each frame holding an arbitrary musical content (rhythms, pitches, harmony… are not constrained), with the only restriction that they should be of finite duration. Each frame is then abstracted into a single pitch obtained by agglomerating all notes within it into a single chord (discarding their individual durations) and taking the root of this chord, using the simple algorithm described before. This root roughly describes the tonal harmony within the frame[4].

The actual representation of the notes in the system is done via the MIDI standard, allowing to work in a bounded integer interval. Using this integer representation, testing for melodic motion reduces to an integer comparison between the current value and the next possible value of the variable $pitchRoot$[5].

4.1.2. From tonal theory to CTL properties: an application

Now an application of the atomic formulae described in the previous section is presented, allowing to extract chord progressions (or *cadences*) from a VMO's content. This is done by first turning the chord progression into a CTL formula stating its existence, then turning the VMO into a Kripke structure (as defined in section 3.1) reflecting its automaton structure, following a process which will be described in section 4.3, in order to finally model-check the generated CTL property on the Kripke structure and return the result to the user – in the form of an example of a path following the requested chord progression if it actually exists.

The ability to constrain chord progressions shows very useful when trying to add structure to musical content, as demonstrated in the following use cases.

Examples of usage One use of cadences arises in the case of a musical dialogue between an improvising VMO and a live musician: if the musician wishes at any time to move on to a different tonality than the one he is currently using, he could with this tool request the oracle to find a way of reaching said goal tonality from its current state with a smooth transition, in order to avoid breaking the continuity of the music, allowing to have a common movement between human and machine.

Another possible use of these harmonic constraints is the morphing of musical streams: for this problem, given two music pieces, A and B, one wishes to extract an interpretation of A from B. This can be done by reading A with a given frame size and extracting the root for each of the successive frames, thus turning A into a chord progression, then using a VMO built on B and the model-checking tools to extract this cadence from A.

This morphing feature has been implemented in the proposed package (using degree progressions for added flexibility) and has been tested to work on various examples. Still, requesting the complete degree progression from A is asking a lot, and it is not sure that all of it will appear in B. Such a strong constraint may not be satisfiable

[3] This method is used by default in the Python library `music21` which is used for the high-level representation of music streams in the proposed implementation.

[4] As is the case in Fourier analysis, a long frame size will often have a smoothing effect, detecting only general harmonic regions. On the opposite, a very short frame size will mistake melody for harmony: each single note will be seen as an harmonic region in itself, leading to very frequent harmonic changes, and loosing continuity.
A frame size that corresponds in terms of its time scale to the analyzed track's time-signature is often a reasonable choice (ie. between one measure and one quarter note), allowing a trade-off between smoothness and precision.
The same remark applies in the following to the slicing of music streams before turning them into a VMO.

[5] If melodic motions are unused, it is also possible to discard octave values and work solely on pitch classes, thus reducing the possible values for $pitchRoot$ to only 12 different pitch classes (using the chromatic scale) and improving the efficiency of the model-checking.

and this would currently lead to the system not returning anything. In order for the system to be really usable as a generation tool (e.g. in a live context), it will therefore be necessary to add within the system a mechanism of constraint relaxation. Such a process would operate as follows:

- If the generation of a path satisfying property P is not possible,
- Then try and generate a generalization $\tilde{P}$ of P, i.e. a weaker, less constraining property than P on the logical point of view, in the hope that some path on the structure may satisfy $\tilde{P}$,
- Iterate this relaxation until a path satisfying some generalization of P is satisfied, with the last possibility being a completely random path, if the generation of a meaningful path has failed.

Such a relaxation mechanism should be application-specific and for instance, in the case of chord progression extraction, a first step of relaxation would be to allow substitutions of a tonic with its fifth, then, if this did not succeed, replacements with thirds and so forth, making use of tonal (or jazz) theory.

4.2. Algorithms

The algorithms used for the chord progression extraction feature can now be described. To begin with, consider how this extraction is made possible by the previously presented tools: the local constraints given by the variable $pitchRoot$ allow for the step-by-step constraining of the chords used, and the existential capacities of CTL allow for the choice of branches satisfying the successive requirements, effectively extending the local constraints to a global one, the chord progression.

A key element in this process is the transcription function MAKE_PROG_PROP, which converts a chord progression into a CTL formula. Below are some examples of the expected behavior:

Example. This first example expresses the reachability of a sequence of frames of length at least zero with harmony in C:

$$\text{MAKE\_PROG\_PROP}([C]) = \mathsf{EF}\begin{pmatrix}(pitchRoot = C) \wedge \\ \mathsf{E}\,(pitchRoot = C \;\mathsf{U}\; \top)\end{pmatrix}$$

This property is to be understood as follows: one eventually (as expressed by the EF) reaches a frame with harmony in C ($pitchRoot = C$) from which a path (possibly reduced to this single state) departs ($\mathsf{E}\,(\cdot \;\mathsf{U}\; \cdot)$) going only through frames in C before reaching any other state ($\top$).

A second, more involved example, with several notes and melodic motion constraints, in the form of plus (meaning the motion reaching this note should be ascending) or minus (for descending motions) signs:

Let `prog` $= [C, +G, -F]$, then:

$$\text{MAKE\_PROG\_PROP}(\texttt{prog}) = \mathsf{EF}\left(\begin{array}{l}(pitchRoot = C \wedge \mathsf{E}\,(pitchRoot = C \;\mathsf{U}\; \ldots \\ \ldots motion = \texttt{ascending} \wedge pitchRoot = G \\ \quad \wedge\, \mathsf{E}\,(pitchRoot = G \;\mathsf{U}\; \ldots \\ \ldots motion = \texttt{descending} \wedge pitchRoot = F \\ \quad \wedge\, \mathsf{E}\,(pitchRoot = F \;\mathsf{U}\; \top)))\end{array}\right)$$

This formula states the existence of a path on the model eventually reaching the beginning of this chord progression and thereafter following it chord by chord.

The code for the recursive function MAKE_PROG_PROP is displayed in Algorithm 1. It converts its input sequence `prog` of pitch classes into a CTL property `prop`. The version displayed does not account for motions, for sake of brevity, though the actual implementation recognizes those. Note that the actual implementation is also more generic in that it supports minor scales and could very easily be upgraded to support arbitrary scales.

Finally, the durations in this version are not constrained: the consecutive degrees can each be held for an arbitrary time, as long as they each last for at least one frame. The implementation actually supports time-constraints, allowing to set a duration for each of the chord intervals to extract, in an interval fashion: the user inputs the minimum and maximum acceptable durations for the given degree and the system in turns only considers paths satisfying these added constraints. This is done using real-time CTL model-checking, as introduced by Emerson et al. [11], a straightforward extension of CTL with time quantifications.

Definition 6 (Real-time CTL)**.** Let φ and ψ two CTL properties and (t_{min}, t_{max}) a pair of integers. In real-time CTL, the "exists until" operator application $\mathsf{E}\,(\varphi \;\mathsf{U}\; \psi)$ becomes the following "bounded-until existential":

$$\mathsf{E}\,(\varphi \;\mathsf{BU}^{t_{min} \leq t < t_{max}}\; \psi)$$

This property expresses the fact that: "There exists a path (starting from the current state) such that:

- Property ψ holds at a time instant t_ψ (counted in number of steps from the beginning of the path) with $t_{min} \leq t_\psi < t_{max}$,
- And property φ holds for all time instants t such that $t_{min} \leq t < t_\psi$".

In this specification, the first $t_{min} - 1$ steps in the path bear no constraint at all.

Once the conversion is done, the chosen external model-checking engine is called with the generated property and the model. This is achieved by a function EXISTS_PROG, presented in Algorithm 2. Given a chord progression $\mathcal{P} = (d_0, \ldots, d_k)$ with $d_i \in \{\text{I}, \ldots, \text{VII}\}$ for $i \in [\![\, 0,\, k \,]\!]$ and

Algorithm 1 Function MAKE_PROG_PROP

```
Require: prog is a sequence of pitch classes
 1: function MAKE_PROG_PROP(prog)
 2:     function MAKE_AUX(prog)
 3:         if ISEMPTY(prog) then
 4:             ▷ An empty progression trivially exists on any structure
                return $\top$
 5:         else
 6:             root :: roots ← prog
 7:             ▷ Recursively build remaining CTL property,
                combine both afterwards using CTL's compositionality
 8:             next_prop ← MAKE_AUX(roots)
 9:             ▷ Constrain the next harmonic content to be of root root
                until the machine reaches the following chord in the progression
10:             return $(pitchRoot = \texttt{root} \wedge$
                        $\mathsf{E}\ (pitchRoot = \texttt{root}\ \mathsf{U}$
                          next_prop$)$
11:         end if
12:     end function
13:     ▷ The EF · operator relaxes the constraints on the first steps in the machine
        return (EF MAKE_AUX(prog))
14: end function
```

Algorithm 2 Function EXISTS_PROG

```
Require: $\mathcal{P}$ is a sequence of integers between 1 and 7, M
 1: function EXISTS_PROG($\mathcal{P}$, $M$)
 2:     result ← $False$
 3:     $i \leftarrow 0$
 4:     while $(i \leq 12) \wedge (\texttt{result} = False)$ do
 5:         tonic ← $i$
 6:         prog ← INSTANTIATE_PROG($\mathcal{P}$, tonic)
 7:         prop ← MAKE_PROG_PROP(prog)
 8:         result ← MODEL_CHECK($\neg$ prop, $M$)
 9:         $i \leftarrow i + 1$
10:     end while
11:     return result
12: end function
```

a tonal Kripke structure M, EXISTS_PROG iteratively tests for the existence of an instantiation $\mathcal{P}_{tonic}$ of $\mathcal{P}$ on M with $tonic$ chosen in the set $\{C, C\sharp, \ldots, B\flat, B\}$.

Note that the model-checker, called via the function MODEL_CHECK, operates on the negation of the chosen specification: in order to generate an example of path a satisfying the requested existential constraint, the algorithm attempts to generate a counter-example to the universal property stating that no such path exists, a strictly equivalent problem in classical logic[6] .

4.3. Abstracting the VMO

In this section, the way the VMO can be turned into a Kripke structure is introduced, a procedure made straightforward by the VMO's automaton structure – with a conceptual difficulty overcome by using the FO's specificity.

Consider a VMO V built by slicing an original musical stream $\mathcal{S}$ into frames of a given frame size fs, i.e. the i-th state of V holds the i-th frame of duration fs of $\mathcal{S}$.

The goal is to build a Kripke structure

$$M_V = (S, T, I, \mathrm{AP}, \ell)$$

modeling V and such that a path on M_V can be immediately and unambiguously transcribed into a new musical stream. Indeed, the labelled or unlabelled nature of the links in V will be lost when turning it into a Kripke structure, thus potentially leading to ambiguities. It is therefore necessary to create a Kripke structure such that each step in the Kripke structure implies the generation of a new frame of output, that is, a Kripke structure extracted from the VMO were all ε-transitions have been suppressed.

The set of atomic propositions AP thus consists of the propositions $(pitchRoot = p)$ for p in $[\![0, 127]\!]$, to constrain tonal roots, and $(motion = m)$ for m in $\{ascending, descending\}$, in order to constrain melodic motion. Plus, for each variable a special "None" value for the empty, initial state and for silent frames (frames containing no musical content).

Because of the notion of memory implied by the use of motions (one has to know the pitch root of the previous , the set of states S requires a bit more work: each state in V has two copies in S. Both copies are labelled with $(pitchRoot = p)$, where p is the root obtained as presented before by turning all the content of the frame associated to the state into a single chord. One of the copies is labelled with $(motion = ascending)$, the other with $(motion = descending)$. When building a transition in the structure to a given state, the choice of either of its copies as destination will then depend on the pitch root of the transition's origin.

Now, regarding the set of transitions T, as stated before, the distinction between labelled and ε-transitions (the suffix links) is lost when going from automaton to Kripke structure, and the reconstruction could suffer from this. This could be fixed by determinizing the automaton, but with an exponential cost in both time and space, which would greatly impact the model-generation and model-checking

[6] This approach may seem unnecessarily complicated but is due to the fact that – up to the author's knowledge – model-checking tools do not offer example generation to existential properties but rather (though not all even do so) counter-example generation to universal properties. This most probably lies in the fact that the primary industrial intent of these tools is to check universal properties of the form: "for all execution of the system, it does not go wrong", and to return an counter-example to this safety property in case it is not verified. A more appropriate tool for the application presented in this article would therefore be the dual of a model-checker (if such a tool exists, in an implementation as efficient as model-checkers): a path-finding engine working under constraints.

performances, making it hardly usable in real-time applications. It is thus necessary to remove all ε-transitions without having the state space explode.

Thankfully, the VMO allows just that: as presented in section 2.2, the structure keeps tracks of all symbol clusters and these clusters can equivalently be seen as reachability classes via undirected suffix links. Therefore, the states reachable via a uniquely labelled path – thus effectively generating a symbol – from a given state s are the following:

$$\bigcup_{s' \in cluster(s)} \{s'' \mid \exists \text{ a forward link } (s', s'') \text{ in } V\}$$

These states are reachable via a (possibly empty) succession of suffix links and a final, labelled, forward link. They bear the label of this final link, i.e. the symbol of the final state of the path.

Using these *precomputed* clusters, it is possible to build a model bypassing all suffix links and producing an unambiguous symbol at each step. These transitions, along with the specification given previously regarding motions, are reported in T. Combining these specifications, we obtain a complete definition for the Kripke structure associated to V, built in time and space linear with respect to $|V|$, thus completing the construction.

Remark. If one were to apply the techniques presented in this paper to another type of automaton than the VMO, these clusters would therefore have to be computed. Apart from this, the model-checking procedure is essentially the same.

4.4. The model-checker: nuXmv

In this section, some key aspects of *nuXmv*, the actual model-checking engine currently used in the proposed implementation, developed by Cavada et al. [4], are presented.

Amongst the pros for this model-checker, compared to other considered options like *Prism* and UPPAAL, is its support for the whole of CTL for the specifications and the support for real-time CTL model-checking, when some other CTL model-checker actually only support a subset of the core logic. It also offers counter-example generation, a must for the considered application, since those are needed to actually generate paths satisfying the requested constraints.

Nonetheless, some potentially useful features are lacking. For instance, the counter-example generation engine currently only generates one counter-example and there is therefore no way to generate different paths satisfying the same constraint, which could be interesting in a composition context, as it would give the composer the ability to choose between different, logically fitting solutions. It also lacks support for *reward-based model-checking*. Disposing of a model-checker aiming at satisfying a constraint whilst maximizing a given reward measure would allow for even finer control of the output, for instance by trying to maximize the diversity within the generated music by maximizing the number of different states visited during the run. For now, the authors did not read about a model-checker for CTL working with reward constraints.

5. EXPERIMENTAL RESULTS

Some example runs of the system are presented in this section, extracting chord progressions from MIDI Jazz piano transcriptions.

The configuration used is the following: the pieces are from the *Omnibook* [7], a set of transcriptions of recordings by Charlie Parker. The distance used to create the oracle is the Tonnetz-distance – as presented in Harte, Sandler, and Gasser [12] –, a music-theoretic distance on chroma vectors. The threshold θ is fixed by hand at 0.2, a value giving reasonable clustering of the harmonic content with respect to the Tonnetz-distance.

Four pieces were chosen: *Blues for Alice*, *Ornithology*, *Now's The Time*, *Marmaduke*. The two degree progressions used as request are: I-IV-VII-III-VI-VII-III-VI and I-IV-II-V-I-VI-II-V.

These progressions were analyzed in a paper by Pachet and Dubnov [19]. The characterization there was done by applying Lempel-Ziv parsing, building a continuation tree of symbols for compression purposes. This method was the basis for several improvisation systems. The two sequences presented here were characterized as expected (first) and containing surprising transitions (second). The criteria for surprise is not considered here but briefly it is related to the number and length of continuations found in the data. It would be interesting to see the relationship between statistical and logic based methods in the future.

For each piece, an oracle is created with the selected settings, then the algorithm EXISTS_PROG is launched on this oracle with each of the two proposed degree progressions – the tonality of the transcriptions is therefore not a meaningful parameter, since EXISTS_PROG works *modulo* transpositions. The results – in the form of instantiation of the tonic for the actually extracted chord progressions – are displayed on Table 2. Resulting MIDI files along with an iPython notebook detailing the experiment are available in the GitHub repository linked to in Appendix.

Because the second progression cannot be extracted on some of the pieces, it is suggested to concatenate all of the four pieces, building a small corpus of works and then try and generate the progression with this added musical diversity. This indeed succeeds and the progression can be extracted from this corpus (note that this is expected, since it is a superset of some pieces, some of which satisfy the request).

The code for the current implementation of the tools introduced in this article is available on the cited GitHub repository [21], along with the code used to generate these examples. It extends on an initial implementation of the VMO by Cheng-i Wang.

[7] http://tostud.free.fr/Parkeromnibookmidi.html

	Progression 1	Progression 2
Blues for Alice	C	$\emptyset$
Marmaduke	C	C
Now's the Time	$E\flat$	$\emptyset$
Ornithology	C	C
Concatenation	C	C

Table 2. Chosen tonic for extracted chord progressions from the *Omnibook*

6. CONCLUSION AND PERSPECTIVES

An original extension has been introduced to a system already being used for the generation of music, the VMO. The extension, making use of CTL model-checking, offers fine control of the content generated by the system, and extends on previous, lower-level techniques to constrain this output. It should be furthermore noted that both the VMO and this extension are highly generic and could work on any type of data, up to the definition of appropriate atomic propositions to describe properties on this data-type. In this purpose, the implementation of all tools was done with a focus on genericity and extendability: integrating a new logic or a new model-checker should be straightforward. An example of future use of this genericity considered by the authors is the generation of graphic patterns following live music, a form of *automated Video Jockey*.

Future possible works include an extensive testing of the tools in musical contexts, to obtain a more satisfactory validation of the methods employed, for instance via statistical experiments with listeners judging the quality of the output. Further development of the – for now rather simplistic – logic used to express musical properties is also required, in the form of new, meaningful atomic propositions – e.g., rhythmic constraints. Great improvements regarding the model-checking engine used in the back-end are possible as well: support for reward-based model-checking, multiple counter-example generation… Finding a tool with these capacities would represent a valuable improvement to the control capacities offered by the proposed system.

Finally, the system should be tested in real-time contexts: for now, due to the apparent inability of nuXmv to access, store and modify the models in-place, in order to use the model-checking capacities in real-time situations with the VMO, one must make a new, complete call to the model-checking engine and rebuild the whole system whenever new states were added. This could become a problem with large systems and requires attention. Nevertheless, provided that such an in-place implementation existed, an online application of the tools presented in this article would then be possible, where one would receive new data and grow the underlying Kripke structure accordingly in real-time, given a fixed logical specification. This could indeed be done by updating the structures online and in-place and making periodic calls to the model-checking engine, thus refreshing the current optimal path for the requested specification, starting from the current state and based on the most up-to-date Kripke structure.

Furthermore, as mentioned earlier, in order to make the system really usable in live contexts, a procedure to meaningfully and automatically relax the requested properties when the generation failed with the stronger property should be implemented. This would avoid leaving the user with no other option than random generation in critical cases!

Acknowledgment

The authors would like to thank *IdEx - Université de Bordeaux* for partially funding this work. The authors also extend their thanks to Jérôme Nika and the anonymous reviewers, whose helpful remarks helped improve the final version of this paper.

References

[1] Allauzen, C., Crochemore, M., and Raffinot, M. "Factor oracle : a new structure for pattern matching". In: *26th Seminar on Current Trends in Theory and Practice of Informatics (SOFSEM'99)*. Ed. by Jan, P., Gerard, T., and Miroslav, B. Vol. 1725. LNCS. Milovy, Czech Republic, Czech Republic: Springer-Verlag, Nov. 1999, pp. 291–306.

[2] Ariza, C. "The Interrogator As Critic: The Turing Test and the Evaluation of Generative Music Systems". In: *Comput. Music J.* 33.2 (June 2009), pp. 48–70. ISSN: 0148-9267.

[3] Assayag, G. et al. "OMAX Brothers: A Dynamic Topology of Agents for Improvisation Learning". In: *ACM Multimedia Workshop on Audio and Music Computing for Multimedia*. Santa Barbara, United States: Santa Barbara, 2006.

[4] Cavada, R. et al. "The nuXmv Symbolic Model Checker". In: *CAV*. 2014, pp. 334–342.

[5] Deliège, I. "Similarity Perception $\leftrightarrow$ Categorization $\leftrightarrow$ Cue Abstraction". In: *Music Perception: An Interdisciplinary Journal* 18.3 (2001), pp. 233–243. ISSN: 07307829, 15338312.

[6] Donze, A. et al. *Control Improvisation with Application to Music*. Tech. rep. UCB/EECS-2013-183. EECS Department, University of California, Berkeley, Nov. 2013.

[7] Dorien, H. and Kenneth, S. "Composing first species counterpoint with a variable neighbourhood search algorithm". In: *Journal of Mathematics and the Arts* 6.4 (2012), pp. 169–189.

[8] Dubnov, S., Assayag, G., and Cont, A. "Audio Oracle Analysis of Musical Information Rate". In: *Semantic Computing (ICSC), 2011 Fifth IEEE International Conference on*. Sept. 2011, pp. 567–571.

[9] Eigenfeldt, A. “Realtime Generation of Harmonic Progressions Using Controlled Markov Selection”. In: 2010.

[10] Emerson, E. A. and Halpern, J. Y. “Decision procedures and expressiveness in the temporal logic of branching time”. In: *Journal of Computer and System Sciences* 30.1 (1985), pp. 1–24. ISSN: 0022-0000.

[11] Emerson, E. A. et al. “Quantitative Temporal Reasoning”. In: *Proceedings of the 2Nd International Workshop on Computer Aided Verification*. CAV ’90. London, UK, UK: Springer-Verlag, 1991, pp. 136–145. ISBN: 3-540-54477-1.

[12] Harte, C., Sandler, M., and Gasser, M. “Detecting Harmonic Change in Musical Audio”. In: *Proceedings of the 1st ACM Workshop on Audio and Music Computing Multimedia*. AMCMM ’06. Santa Barbara, California, USA: ACM, 2006, pp. 21–26. ISBN: 1-59593-501-0.

[13] Kendra, J. and Wachtendorf, T. “Improvisation, creativity, and the art of emergency management”. In: *From Understanding and Responding to Terrorism*. Ed. by Durmaz, H. et al. IOS Press, 2007, pp. 324–335.

[14] Lerdahl, F. and Jackendoff, R. *A generative theory of tonal music*. Cambridge. MA: The MIT Press, 1983. ISBN: 0262120941.

[15] Nattiez, J.-J. *Fondements d’une sémiologie de la musique*. French. "10/18". Paris Union générale d’éditions, 1975. ISBN: 2264000031.

[16] Nika, J., Chemillier, M., and Assayag, G. “ImproteK: Introducing Scenarios into Human-Computer Music Improvisation”. In: *ACM Computers in Entertainment, Special issue on Musical Metacreation* (2016). (To appear).

[17] Nika, J. et al. “Guided improvisation as dynamic calls to an offline model”. In: *Sound and Music Computing (SMC)*. Maynooth, Ireland, July 2015.

[18] Pachet, F. and Roy, P. “Markov constraints: steerable generation of Markov sequences”. In: *Constraints* 16.2 (Mar. 2011), pp. 148–172.

[19] Pachet, F. and Dubnov, S. “Surprising harmonies”. In: *International Journal of Computing Anticipatory Systems* 4 (1999), pp. 139–161.

[20] Simon, I., Morris, D., and Sumit, B. “Automatic Accompaniment Generation for Vocal Melodies”. In: *Proceedings of ACM CHI 2008 (the 26th SIGCHI Conference on Human Factors in Computing Systems)*. 2008, pp. 725–724.

[21] Wang, C.-i. and Bazin, T. *vmo - Python Variable Markov Oracle Library*. GitHub repository. 2015. URL: https://github.com/wangsix/vmo.

[22] Wang, C.-i. and Dubnov, S. “Guided Music Synthesis with Variable Markov Oracle”. In: *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*. 2014.

[23] Wang, C.-i. and Dubnov, S. “Variable Markov Oracle: A Novel Sequential Data Points Clustering Algorithm with Application to 3D Gesture Query-Matching”. In: *Multimedia (ISM), 2014 IEEE International Symposium on*. Dec. 2014, pp. 215–222.

[24] Ziller, R. and Schneider, K. “Combining Supervisor Synthesis and Model Checking”. In: *ACM Trans. Embed. Comput. Syst.* 4.2 (May 2005), pp. 331–362. ISSN: 1539-9087.