Specification Mining for Machine Improvisation with Formal Specifications

Rafael Valle, UC Berkeley, CNMAT
Alexandre Donzé, UC Berkeley
Daniel J. Fremont, UC Berkeley
Ilge Akkaya, UC Berkeley
Sanjit A. Seshia, UC Berkeley
Adrian Freed, UC Berkeley, CNMAT
David Wessel, UC Berkeley

We address the problem of mining musical specifications from a training set of songs and using these specifications in a machine improvisation system capable of generating improvisations imitating a given style of music. Our inspiration comes from *Control Improvisation*, which combines learning and synthesis from formal specifications. We mine specifications from symbolic musical data with musical and general usage patterns. We use the mined specifications to ensure that an improvised musical sequence satisfies desirable properties given a harmonic context and phrase structure. We present a specification mining strategy based on pattern graphs and apply it to the problem of supervising the improvisation of blues songs. We present an analysis of the mined specifications and compare the results of improvisations generated with and without specifications.

Additional Key Words and Phrases: Specification Mining, Formal Methods, Control Improvisation, Machine Learning

1. INTRODUCTION

The field of *machine improvisation*, i.e. computer music improvisation, has been investigated under mainly two approaches: rule-based and data-driven. Rule-based approaches attempt to define rules characterizing "good" improvisations and generate pieces of music that follow these rules. However, it has been observed that it is difficult to come up with the "right" rules, resulting in systems that are either too restrictive, limiting creativity, or too relaxed, thereby allowing undesirable behavior [Cope 1991; Dubnov and Assayag 2002; Keller and Morrison 2007]. Data-driven approaches tend to employ machine learning techniques to learn generative models from music samples and use these models to generate new melodies. Examples of such models include stochastic context-free grammars (SCFGs) [Gillick et al. 2009; Keller 2012], hidden Markov models (HMMs) [Gillick 2009; Paiement et al. 2009], and universal predictors [Dubnov et al. 2003; Dubnov and Assayag 2002; Assayag and Dubnov 2004; Cabral et al. 2006]. Some systems combine rule-based and data-driven approaches; e.g. the Impro-visor system [Keller and Morrison 2007] based on SCFGs uses rules learned by grammatical inference from training licks [Gillick et al. 2009]. Related to our work, [Pachet et al. 2001] describe non-homogeneous Markov processes with control constraints, e.g. last pitch must be a specific note. While Pachet's work focuses on unary constraints manually created by the user and binary constraints that are within the scope of the Markov order, this paper focuses on learning constraints from data as formal specifications. Our recent efforts in this direction are presented in [Donzé et al. 2014], in which we define the problem of machine improvisation with formal specifications. In computer science, a *formal specification* is a mathematical statement of expected behavior of a system, typically given in mathematical logic or as an automaton. In [Donzé et al. 2014], we considered the scenario of improvising a monophonic jazz melody given a training sequence (melody) and a chord progression. Overall, our approach described in [Donzé et al.

DOI: http://dx.doi.org/10.1145/0000000.0000000

2014] consists of two stages: a generalization stage, where a reference sequence (e.g. obtained from a human improviser) is used to learn an automaton generating similar sequences, and a supervision stage, that enforces specifications on pitch and rhythm.

In [Donzé et al. 2014], the specification, formally represented as a finite state automaton (FSA), encodes rhythmic and harmonic constraints adapted and simplified from generic jazz improvisation guidelines found in Keller's *How to Improvise Jazz Melodies* [Keller 2012]. Although these hand-crafted guidelines can be manually converted into formal specifications, this task is time-consuming even in simple cases. Generally, writing specifications requires knowledge of logic not possessed by most composers and, conversely, musical knowledge not possessed by most logicians. *Specification mining* offers a solution that is either entirely automatic or only requires the much simpler task of creating templates for the musical patterns of interest. Our engine statistically learns, in the form of a pattern graph, the musical characteristics of a song dataset by mining predefined musical and general usage patterns from it. In this paper, we evaluate our approach using a dataset of traditional blues songs and the predefined patterns focus on properties related to rhythm, pitch, melodic contour and chord/non-chord tones.

The paper is organized as follows. Section 2 gives an overview of our approach, using a simplified presentation and a small example, followed by related work in Section 3. Section 4 describes the control improvisation and specification mining formalisms our techniques are based on, while the algorithms themselves are described in Section 5. Next, Section 6 details the specific musical features used in our experiments, whose results are presented in Section 7. Finally, we conclude in Section 8 with a summary and directions for future work.

2. OVERVIEW

In this section, we give an informal overview of the machine improvisation approach that we developed, sketching the different components using a simplified formalization and a small example. As sketched in Figure 1, the overall flow begins with a *training* set of songs $\mathfrak{D} = \{\mathfrak{s}_1, \ldots, \mathfrak{s}_N\}$ and a *reference* song $\mathfrak{s}$, and produces as output a set of specifications used in a *controller*, and an *improviser*, which is the actual object generating new improvisations, i.e., new sequences of notes. The improviser and the procedure to enforce the specifications are implemented closely following [Donzé et al. 2014]: the main contribution of the present paper is how to *generate* the specifications.

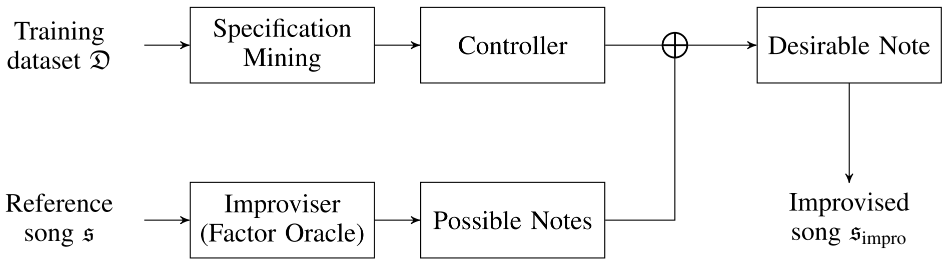
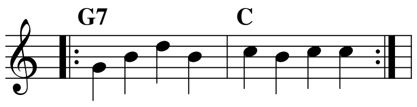

Training dataset $\mathfrak{D}$
Specification Mining
Controller
Desirable Note
Reference song $\mathfrak{s}$
Improviser (Factor Oracle)
Possible Notes
Improvised song $\mathfrak{s}_{\text{impro}}$

Fig. 1: Workflow of our approach.

All songs are assumed to be in *lead sheet* format, i.e. with a single instrument melody line and an accompaniment specified as a simple chord sequence. In the following, assume $\mathfrak{s}$ is given by:

$\mathfrak{s} =$ 

Using standard chord and pitch notations, such as **C**, **Am**, **D7** and `a`, `a#`, `b`, `c`, `c#`, `d`, etc., we can write $\mathfrak{s}$ as a sequence of pitch and chord pairs:

(`g`,**G7**)(`b`,**G7**)(`d`,**G7**)(`b`,**G7**)(`c`,**C**)(`b`,**C**)(`c`,**C**)(`c`,**C**)

We say that $\mathfrak{s}$ can be encoded using two *alphabets*, one for chords and another for pitches. Here, for the sake of simplicity, we ignore duration information and other nuances that can be encoded using other alphabets.

Specifications. The songs $\mathfrak{s}_1, \ldots, \mathfrak{s}_N$ in our training set can be represented in the same way and we assume that they all satisfy some *a priori* unknown set of *specifications*, which are properties of the combined sequence of pitches and chords. Simple examples of such specifications include:

— φ_1: the current pitch belongs to the current chord, e.g.,
  — (`c`,**C**),(`g`,**C**),(`f`,**G7**),(`b`,**G7**) satisfies φ_1
  — (`c#`,**C**),(`a`,**C**),(`c`,**G7**),(`e`,**G7**) does not satisfy φ_1

— φ_2: the current pitch does not belong to the chord, but the one before did and the one after will, and they are at an interval not greater than one tone, e.g., (`c`,**C**),(`b`,**C**) is a sequence satisfying φ_2 *iff* the next note is (`c`,**C**).

— φ_3: about 70% of the time, a (`g`,**C**) is followed by a (`c`,**C**).

In [Donzé et al. 2014], such specifications were described and implemented manually, whereas in this work, we describe how to explicitly and implicitly[1] extract these specifications from the training set $\mathfrak{D}$. Note also that φ_1 and φ_2 are non-probabilistic (hard) specifications and φ_3 is probabilistic (soft). In [Donzé et al. 2014], we only considered non-probabilistic specifications.

The purpose of the improviser is to generate new songs of arbitrary length, e.g.,

(`g`,**G7**)(`b`,**G7**)(`d`,**G7**)(`b`,**G7**)(`c`,**C**)(`d`,**C**)(`e`,**C**)(`c`,**C**)(`b`,**G7**)(`a`,**G7**)(`g`,**G7**)(`f`,**G7**)(`e`,**C**), ...

that satisfy several criteria, which we state informally below:

(a) All generated sequences satisfy at least one of the non-probabilistic specifications at all times;
(b) The distribution of generated sequences is sufficiently diverse (i.e. there is a variety of different improvisations);
(c) The melody diverges from the reference melody in some controllable way, i.e. it can be made very similar or arbitrarily different;
(d) The distribution of generated sequences satisfies the probabilistic specifications.

In Section 4 we will see how criteria (a), (b), and (c) naturally fit into the framework of control improvisation. This is not the case for criterion (d), since [Donzé et al. 2014] did not consider probabilistic specifications.

Factor Oracle-based improvisation. To construct an improviser satisfying the above criteria, we start by following the approach presented in [Assayag and Dubnov 2004]: we construct the *factor oracle* [Cleophas et al. 2003] corresponding to the melody line of the reference song $\mathfrak{s}$. A factor oracle is a finite state machine with $n + 1$ states (where n is the number of notes) and edges labeled with the pitches of the melody. The factor oracle corresponding to the above reference melody is shown in Figure 2. It is constructed in such a way that if one follows the edges and reads labels, it produces a sequence which is a concatenation of subsequences of the reference sequence. Moreover,

[1]In our system, the pattern φ_1 is learned implicitly given chord degree specifications.

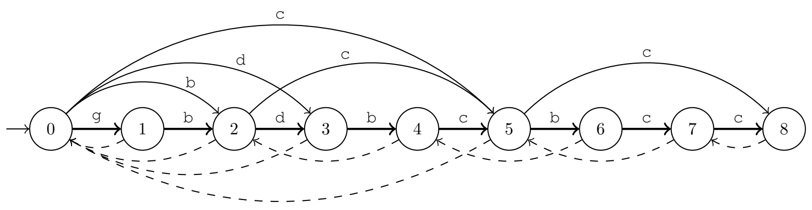


Fig. 2: Factor oracle constructed from the example melody.

if one takes only "direct" transitions, i.e. those from a state i to $i + 1$, and no other forward or backward transitions, then the sequence of labels reproduces exactly the original sequence. It was then observed in [Assayag and Dubnov 2004; Donzé et al. 2014] that by assigning a fixed probability p, called the *replication probability*, to direct transitions, and uniform probabilities to other "branching" transitions, one obtains a stochastic generator which produces sequences similar in some sense to the original sequence, where the degree of similarity is controlled by p. Such a generator satisfies criteria (b) and (c) above for appropriate values of p.

Enforcing Specifications. The factor oracle improviser we just described generates notes without taking into account the harmonic context, and more generally the type of (musical) specifications that we are interested in. Extending our work in [Donzé et al. 2014], and as described in Section 5, our approach works by enforcing the desired or mined specifications over sequences of notes proposed by the factor oracle. For example, without specifications the factor oracle of Figure 2 might generate the sequence `gbdcb` by going through states 0, 1, 2, 3, 0, 5, and 6 . Once combined with the chord sequence, we get (g,**G7**)(b,**G7**)(d,**G7**)(c,**G7**)(b,**C**) which clearly violates both φ_1 and φ_2 above, since c does not belong to the **G7** chord, and b does not belong to the **C** chord. In this situation, our approach would have prevented this improvisation by blocking the transition in the factor oracle from state 5 to 6, forcing the improviser to either take a c transition (valid because this would cause the last three notes to satisfy φ_2) or to go back silently to state 0 and take another transition satisfying either φ_1 or φ_2.

3. RELATED WORK

The concept of *control improvisation* was first introduced in [Donzé et al. 2013b; Donzé et al. 2014]. It was presented as a variation of the standard supervisory control problem for discrete event systems [Cassandras and Lafortune 2006], with applications to the generation of music. A real time implementation was presented in [Donzé et al. 2013a], and the problem was investigated theoretically in [Fremont et al. 2015]. As indicated in the previous section, the core of the improvisation process rests on the notion of a factor oracle. The factor oracle (FO) was initially introduced in [Cleophas et al. 2003] as an algorithm for optimal string matching, and later suggested as a suitable data structure for machine improvisation in [Assayag and Dubnov 2004]. It is in use in several prominent improvisation systems such as OMax[2] and its variant ImproTek [Nika and Chemillier 2012].

In software engineering literature, specification mining is an efficient procedure to automatically infer, from empirical data, general rules that describe the interactions of a program with an application programming interface (API) or abstract datatype (ADT) [Ammons et al. 2002]. It has convenient properties that facilitate and optimize the process of developing formal specifications. First, specification mining is either entirely automatic, or only requires the relatively simple task of creating templates. In addition, it can exploit latent properties that are unknown to the user and only reflected in the data, offering valuable information on commonalities in large datasets.

[2]http://repmus.ircam.fr/omax/home

Techniques to automatically generate specifications date back to the early seventies, including [Caplain 1975; Wegbreit 1974]. More recent efforts that analyze the problem of specification inference include [Alur et al. 2005; Ammons et al. 2002; Engler et al. 2001; Li et al. 2010]. In general, specification mining tools infer temporal properties in the form of mathematical logic or automata. Broadly speaking, the two main strategies for building these automata include: learning a single automaton and inferring specifications from it; learning small templates and designing a complex automaton from them. For example, [Ammons et al. 2002] learns a single probabilistic finite state automaton from the trace and then extracts likely properties from it. The other strategy circumvents the NP-hard challenge of directly learning a single finite state automaton [Gold 1967; Gold 1978] by first learning small specifications and then post-processing them to build more complex state machines. The idea of mining simple alternating patterns was introduced by [Engler et al. 2001], and several subsequent efforts [Gabel and Su 2008a; Gabel and Su 2008b; Weimer and Necula 2005; Yang et al. 2006] built upon this work.

In music, and specifically music improvisation, the task of learning or describing such general rules is difficult, even for experts, due to music's large parameter space and richness of interpretation. Therefore, specification mining is very attractive because it offers a systematic and automatic mechanism for learning these specifications from large amounts of data.

4. CONTROL IMPROVISATION AND SPECIFICATION MINING

4.1. Control Improvisation

We now describe more formally the automata-theoretic concepts used in this paper, including the control improvisation problem. For a fully formal definition and theoretical treatment of control improvisation, see [Fremont et al. 2015].

4.1.1. Notation and Background. As will be discussed later in Section 6, in this paper we work entirely with discrete, symbolic representations of musical data (pitches, durations, chords, etc.). Objects such as melodies are represented as finite sequences, or *words*, whose elements are drawn from a finite *alphabet* of symbols Σ. We write ϵ for the empty word consisting of no symbols, and $|w|$ for the length of a word w. Words can be combined by concatenation, which we denote as multiplication: for example, ab is a word of length 2 if a and b are symbols in Σ.

A convenient formalism for expressing sets of words is *regular expressions*. The simplest regular expressions are written a for some $a \in \Sigma$, and denote the set of words consisting of the single word a. We also use Σ to denote the set of all of these singleton sets. These basic expressions can then be combined using three operators: *concatenation*, *union*, and *Kleene star*. Given regular expressions p and q, their concatenation pq simply consists of all words which are concatenations of a word in p with a word in q. The union $p \cup q$ is just the set-theoretic union, consisting of all words in either p or q. Finally, the Kleene star p^* is the set of all concatenations of finitely many words in p (including the empty concatenation ϵ). For example, Σ^* is precisely the set of all words over the alphabet Σ, and $(a \cup b)^*$ is the set of all words using only the symbols a and b.

Another useful way to represent sets of words is with finite state automata:

Definition 4.1. A finite state automaton (FSA) is a tuple $\mathcal{A} = (Q, q_0, F, \Sigma, \rightarrow)$ where Q is a set of *states*, $q_0 \in Q$ is the *initial state*, $F \subset Q$ is the set of *accepting states*, Σ is a finite set called the *alphabet* and $\rightarrow \subset Q \times (\Sigma \cup \{\varepsilon\}) \times Q$ is the *transition relation*. We use the notation $q \xrightarrow{\sigma} q'$ to mean that $(q, \sigma, q') \in \rightarrow$.

A word $\sigma_1\sigma_2 \ldots \sigma_n$ is a *trace* of a FSA $\mathcal{A}$ *iff* there exists a sequence of states $q_i \in Q$ such that $q_0 \xrightarrow{\sigma_1} q_1 \xrightarrow{\sigma_2} \ldots \xrightarrow{\sigma_{n-1}} q_{n-1} \xrightarrow{\sigma_n} q_n$. It is an *accepting trace* of $\mathcal{A}$ *iff* q_n is in F. The *language* of $\mathcal{A}$, denoted $\mathcal{L}(\mathcal{A})$, is the set of accepting traces of $\mathcal{A}$.

4.1.2. Problem Definition. As described above, control improvisation seeks to generate random variations on a *reference word* w_{ref}, all of which must satisfy a given specification and whose similarity to the reference can be controlled. Following [Donzé et al. 2014], the possible variations are given as the language of a plant FSA $\mathcal{A}^p$, and the specification is given by another FSA $\mathcal{A}^s$. Dissimilarity to the reference word is measured by a *divergence measure* $d_{w_{\text{ref}}}$, a nonnegative function on words such that $d_{w_{\text{ref}}}(w_{\text{ref}}) = 0$. These together with parameters indicating how much randomness and similarity to w_{ref} is desired specify a control improvisation problem:

Definition 4.2. (Control Improvisation Problem) A control improvisation problem $\mathcal{P}$ consists of FSAs $\mathcal{A}^p$ and $\mathcal{A}^s$ with a common alphabet Σ, an accepting trace w_{ref} of both $\mathcal{A}^p$ and $\mathcal{A}^s$, a divergence measure $d_{w_{\text{ref}}}$, an interval $I = [\underline{d}, \overline{d}]$, and parameters $\varepsilon, \rho \in (0, 1)$. A solution of $\mathcal{P}$ is a probabilistic algorithm generating words w in Σ^* such that the following conditions hold:

(a) *Safety:* each w is an accepting trace of both $\mathcal{A}^p$ and $\mathcal{A}^s$;
(b) *Randomness:* the probability of generating each w is smaller than ρ;
(c) *Bounded Divergence:* $\Pr(d_{w_{\text{ref}}}(w) \in [\underline{d}, \overline{d}]) > 1 - \varepsilon$.

To illustrate how this problem is useful, let us cast the example of Section 2 as an instance of control improvisation. Recall that songs were represented as sequences of pitch and chord pairs: thus our alphabet Σ consists of all possible such pairs. The plant automaton $\mathcal{A}^p$ encodes a model for generating improvisations without enforcing any specifications, which in our case is a factor oracle over the reference song (as described in Section 3). The automaton $\mathcal{A}^s$ depends on which specifications we desire. For the specification φ_1 from Section 2, for example, we might use the automaton in Figure 3. To simplify the diagram we have consolidated some transitions: from q_0 there are separate transitions for input symbols $(\mathtt{c}, \mathbf{C})$ and $(\mathtt{g}, \mathbf{C})$, for example, and generally for all pairs where the pitch is contained in the chord, but since all these transitions lead back to q_0 we have drawn them as a single arrow.

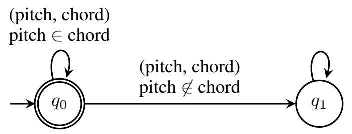


Fig. 3: Specification automaton for φ_1.

Now we can see how requirements (a), (b), and (c) on a generated word in the control improvisation problem correspond exactly to the requirements stated in Section 2 for improvised songs:

(a) The improvisation being an accepting trace of $\mathcal{A}^p$ ensures it can be generated by the underlying specification-free improvisation system (e.g. a factor oracle), while being an accepting trace of $\mathcal{A}^s$ ensures that it satisfies our specifications[3].
(b) By requiring that each improvisation w be generated with probability at most ρ, we ensure that at least $1/\rho$ improvisations can be generated. So by making ρ small we can ensure a diverse distribution of improvisations.
(c) By defining the interval I and parameter ε appropriately, this condition allows us to control how much the improvisations can diverge from the reference song (according to the similarity metric used).

This leaves only requirement (d) from Section 2, namely enforcement of probabilistic specifications. This does not fit into the definition of control improvisation as stated above, and so will be discussed in Section 5.2 below.

[3]In Section 2 we required that the improvisation satisfy *at least one* of several specifications at each event, but those can easily be combined into a single specification that is required to hold over the entire improvisation.

4.2. Specification

In this current work, we expand our previous efforts in [Donzé et al. 2014] by developing an inference engine that mines specifications from a song dataset in the form of *pattern graphs* learned using a set of pre-defined pattern templates. The following paragraphs adapt the work of [Li et al. 2010] to formally describe specification mining in music.

4.2.1. Events and Patterns. Let F be the set of feature vectors extracted from a song $\mathcal{S}$, e.g. pitch, duration, and so forth. For every feature $f \in F$, we use the notation $v_{f,t}$ to indicate the valuation of f at time t.

Definition 4.3 (*Event*). An event is a tuple $(\vec{f}, \vec{v}, t)$, where $\vec{f}$ is a set of musical features and $\vec{v}$ is their corresponding valuations at time t. The alphabet Σ_f is the set of possible events for feature f, and a finite trace τ is a sequence of events ordered by their time of occurrence. We address monophonic music in this paper, so there is only one event at a time.

Definition 4.4 (*Projection*). The projection $\pi_\Sigma(\tau)$ of a trace τ onto an alphabet Σ is defined as τ with all events not in Σ deleted.

Definition 4.5 (*Specification Pattern*). A specification pattern is an FSA over symbols Σ. Patterns can be parametrized by the events used in this alphabet; for example, we use "the $\mathcal{A}$ pattern between events a and b" to indicate the pattern obtained by taking an FSA $\mathcal{A}$ with $|\Sigma| = 2$ and using a as the first element of Σ and b as the second. A pattern *occurs*[4] in a trace τ with alphabet $\Sigma_\tau \supseteq \Sigma$ if and only if there is a subword σ of τ such that $\pi_\Sigma(\sigma) \in \mathcal{L}(\mathcal{A})$.

Definition 4.6 (*Binary Pattern*). A specification pattern with alphabet size 2. We denote a binary pattern between events a and b as a **R** b, where **R** is a label identifying the pattern.

Now that we have described patterns as a general concept, we define three types of patterns that we will use in this paper. Each pattern corresponds to common musical behaviors such as harmonic resolutions and ornaments. For simplicity we define them using regular expressions, which are equivalent to FSAs.

Followed (F): The followed pattern between two events a and b occurs when a is immediately followed by b. It provides information about possible transitions between events, which can be used, for example, to specify the resolution of non-chord tones. We denote the followed pattern as a **F** b and can match it with the regular expression (ab).

'Til (T): The 'Til pattern between two events a and b occurs when a occurs two or more times in sequence and is then immediately followed by b. Compared to the followed pattern, it provides more specific information about what transitions are possible after self-transitions are taken. We denote this pattern as a **T** b and can match it with the regular expression (aaa^*b).

Surrounding (S): The surrounding pattern between two events a and b occurs when event a immediately precedes and succeeds event b. It provides information over a time-window of three events and we musically describe it as an ornamented self-transition. We use a **S** b to denote this pattern and can match it with the regular expression (aba).

4.2.2. Pattern Merging. If every match to a pattern $P_2 = a\ \mathbf{R}\ b$ occurs inside a match to a pattern $P_1 = a\ \mathbf{Q}\ b$, we say that P_1 *subsumes* P_2 and write $P_1 \implies P_2$. When this happens, we only add the *stronger* pattern P_1 to the pattern graph. The purpose of merging is to emphasize longer musical

[4]Note that this is different from the trace *satisfying* the pattern in the sense of [Li et al. 2010]: we are interested in occurrences of patterns *within* a trace, whereas they require the entire trace to match the pattern.

structures: if one pattern always occurs only as part of a longer one, then we will only allow the longer pattern to occur in our generated phrases, but not the shorter pattern by itself.

An example is the chord degree[5] specification mined from the song *Crossroads Blues*, shown in Figure 4. Here, chord degree 10 (note f) is followed by chord degree 7 (note d), so without merging we would learn the pattern *10* **F** *7*. This would allow generating words such as $(10, 7, 10, 7)$, which is inconsistent with the melodic motives in the song, which always have *multiple* occurrences of 10 before transitioning to 7. Thus every match to *10* **F** *7* is contained in a match to *10* **T** *7*, and so with pattern merging we only learn *10* **T** *7*, thereby forbidding $(10, 7, 10, 7)$. In fact, *10* **T** *7* shows how a pattern can subsume multiple patterns, since in this example it also subsumes *10* **F** *10* and *10* **T** *10* (both of which we would otherwise learn).

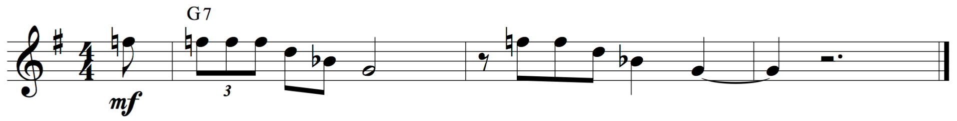


Fig. 4: First phrase of Crossroads Blues by Robert Johnson as transcribed in the Real Book of Blues.

4.2.3. Specifications from Patterns. For each feature $f \in F$, the specifications on f that we mine are of several different types:

(1) Which values of f can occur at the beginning of a phrase.
(2) Which values of f can occur at the end of a phrase.
(3) Which patterns over Σ_f can occur in the phrase.
(4) The empirical probabilities of these patterns.

More formally, the type (3) specification requires that every pattern that matches the trace either is allowed (i.e. occurred in the training data) or is subsumed by one that is allowed. For example, suppose the specification was learned from the single word (a, b, a, b, a). Due to merging, we only learn the pattern *a* **S** *b*, even though for example the word matches *b* **F** *a*. Now consider the word (b, a). The only match to any pattern in this word is the entire word itself, which matches *b* **F** *a*. This pattern was not learned, and is trivially not subsumed by a learned pattern since there are no other matches to subsume it. Therefore (b, a) does not satisfy the specification. However, the word (a, b, a) does satisfy the specification: it matches *a* **S** *b*, which was learned; it also matches *a* **F** *b* and *b* **F** *a*, but both matches are subsumed by the match to *a* **S** *b*.

The first three types of specification are hard constraints that the phrases we generate must respect, while the last can be viewed as a kind of soft constraint. We encapsulate all four types in a data structure we call the *pattern graph*.

Definition 4.7 (*Pattern Graph*). A pattern graph is a labelled directed multigraph whose nodes are elements of Σ_f, i.e. values of a feature f. A node can be labelled as a starting node, an ending node, or neither. Edges are labelled with a type of binary pattern and a count indicating how many times the pattern occurred in the dataset.

For example, an edge (a, b) labelled $(\mathbf{R}, 3)$ in the pattern graph means the pattern *a* **R** *b* occurred 3 times in the dataset. A complete example of a pattern graph is shown in Figure 5, where we have indicated starting nodes with an unlabelled incoming arrow and ending nodes with a double circle (by analogy to the standard notation for FSAs).

While a pattern graph represents the hard specifications above, it is not itself an automaton. However, our method still fits into the framework of control improvisation as presented in Section 4.1.2,

[5]In this paper, chord degree is represented by the distance, in semitones, from a note to the current chord's root note.

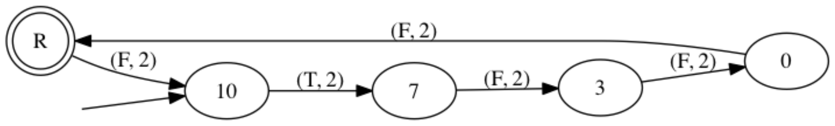


Fig. 5: Pattern graph learned on the chord degree feature (interval from root) extracted from the phrase in Fig. 4.

because the pattern graph can be converted into a specification automaton $\mathcal{A}^s$. In fact, as explained in Section 5.2, our improvisation algorithm does not need to perform this conversion.

5. LEARNING AND ENFORCING SPECIFICATIONS

In this section we describe how we learn hard and soft specifications in the form of pattern graphs, and how those graphs are used to guide the improvisation process.

5.1. Learning Specifications

In this subsection we describe the procedure used to learn pattern graphs from a dataset $\mathfrak{D}$ with features $\mathcal{F}$. The procedure also takes as input a set $\mathcal{P}$ of patterns, consisting of a function for each pattern type **R** that maps feature values a, b to a regular expression defining the pattern a **R** b. As a preprocessing step, the songs in $\mathfrak{D}$ are segmented into three phrases (A A' B)[6]. An example of a segmented dataset is shown in Table I.

After segmentation, for each feature f a pattern graph $\mathcal{G}_f$ is constructed by Algorithm 1, whose main steps are as follows. First, for each phrase we add the first and last feature values as starting/ending nodes respectively in the graph. Second, for every $a, b \in \Sigma_f$ we find all matches to a **R** b for every pattern type **R**. If there is a match to a **R** b which is not subsumed, then we add a corresponding edge to the pattern graph labelled with the number of times the pattern occurs. A naïve implementation of this algorithm could explicitly compute the locations of every match by instantiating all possible regular expressions for the patterns in $\mathcal{P}$ and finding all ways each expression matches the phrase. Then the subsumption check is simply a matter of comparing the locations of the matches. If the number of feature values or pattern types is large, this could be highly inefficient, but for the pattern types used in this paper all non-subsumed patterns can be found with a fast linear search.

5.2. Improvising with Specifications

The core of our improvisation approach is the factor oracle (FO), briefly described in Section 2. Following [Donzé et al. 2014], we enforce specifications on top of the factor oracle by solving a control improvisation problem where the plant $\mathcal{A}^p$ encodes the factor oracle built from w_{ref} and enforces its chord progression. As described in [Donzé et al. 2014], if the specification automaton $\mathcal{A}^s$ is non-blocking in the sense that an accepting state is always reachable, i.e. there are no deadlocks, we can solve the control improvisation problem by restricting the factor oracle to only take transitions consistent with $\mathcal{A}^s$. If there are no such transitions in the factor oracle, we use heuristics to decide which destination state is most appropriate. This procedure can be extended to general specifications using techniques from supervisory control (see [Donzé et al. 2014]).

As mentioned above, the pattern graphs learned by the algorithm in the previous section can be converted into a specification automaton $\mathcal{A}^s$. However, this construction involves taking the product of many automata — one for each pattern — resulting in a final automaton whose size grows exponentially with the number of patterns. So in practice $\mathcal{A}^s$ is likely to be too large to construct explicitly. Therefore, we use the following heuristic: we assume $\mathcal{A}^s$ is non-blocking, and if we

[6]Phrase boundaries are automatically extracted and manually corrected if necessary.

ALGORITHM 1: Specification Mining Algorithm

```
Input: dataset $\mathfrak{D}$ over features $\mathcal{F}$; patterns $\mathcal{P}$
Output: a pattern graph $\mathcal{G}_f$ for each $f \in \mathcal{F}$
1  for $f \in \mathcal{F}$ do
2      $\mathcal{G}_f \leftarrow$ new pattern graph on vertices $\Sigma_f$
3      for $song \in \mathfrak{D}$ do
4          for $phrase \in song$ do
5              $phrase_f \leftarrow$ the sequence of values of the feature $f$ in $phrase$
6              label the first element of $phrase_f$ as a starting node in $\mathcal{G}_f$
7              label the last element of $phrase_f$ as an ending node in $\mathcal{G}_f$
8              for $a, b \in \Sigma_f$ do
9                  $counts \leftarrow$ countPatternMatches$(a, b, phrase_f, \mathcal{P})$
10                 foreach pattern $P$ with $counts(P) > 0$ do
11                     add to $\mathcal{G}_f$ the edge $(a, b)$ with label $(P, counts(P))$
12                 end
13             end
14         end
15     end
16 end
```

reach a blocked state (i.e. one with no outgoing transitions), we reject our current improvisation and start over. Then we can use the procedure for non-blocking automata cited above, which only requires being able to compute for any state of $\mathcal{A}^s$ the set of outgoing transitions. As we will show below, this information can be read off from the pattern graph without having to construct $\mathcal{A}^s$. In practice we find that reaching a blocked state is rare, so few restarts are required and this procedure is efficient.

Determining the transitions allowed by $\mathcal{A}^s$ from the current state is straightforward. At the beginning of an improvisation, we only allow symbols which are labelled as starting nodes in the pattern graph. Likewise, we only allow an improvisation to end on symbols labelled as ending nodes. Finally, if the last generated symbol was a, a transition on symbol b is allowed only in the following situations:

(1) the pattern graph has an edge from a to b labelled with pattern **F**;
(2) the pattern graph has an edge from a to b labelled with pattern **S**; in this case we require the next transition to be on symbol a;
(3) the pattern graph has an edge from a to b labelled with pattern **T**, and the symbol before a was also an a (as the a **T** b pattern requires two or more copies of a);
(4) $a = b$, i.e. the transition would generate another a, and the pattern graph has an edge from a to any symbol c labelled with pattern **T** (since a **T** c allows arbitrarily many copies of a prior to c) .

It is easy to see that this method correctly enforces our specification $\mathcal{A}^s$: the generated words match only patterns that occur in the pattern graph or are subsumed by such patterns.

As an example, consider the pattern graph built from the single word $aaab$. Because of pattern merging, the graph will only have a single edge, from a to b and labelled $(\mathbf{T}, 1)$. Initially, we only allow a transition on a since it is the only node labelled as a starting node. Next, situations (1), (2), and (3) above do not hold (the last because we have not generated any symbol prior to the a), but situation (4) does and allows another transition on a. Now by (3) and (4) we can transition on either b or a respectively — suppose we choose the former (as we will discuss below, we actually pick between transitions randomly if more than one is available). Since b is labelled as an ending node in the graph, we can stop here with the improvisation aab.

One remaining question is how to pick the transition to follow in the factor oracle when more than one choice is consistent with $\mathcal{A}^s$. This is where we incorporate the probabilistic or "soft" specifications mentioned in Section 2. Building on a suggestion in [Donzé et al. 2014], we randomly sample from the consistent transitions: a direct transition gets the replication probability p, and the other transitions get probabilities related to their empirical probabilities in the dataset. Specifically, the probability for a non-direct transition is computed as follows: we sum the counts in the pattern graph for every edge that can allow the transition according to the rules above, and assign a probability proportional to this sum.

To illustrate this computation, consider the pattern graph learned from the word $aabcaabcaa$, and say we have generated aa so far. The pattern graph has an edge from a to b labelled $(\mathbf{T}, 2)$, and an edge from a to a labelled $(\mathbf{F}, 1)$. According to the rules above, the first edge allows us to transition on b (by (3)), and both edges allow us to transition on a (by (4) and (3) respectively). Adding up the corresponding counts, we assign probabilities to b and a proportional to 2 and $2+1 = 3$ respectively (assuming neither transition is the direct transition in the factor oracle). Normalizing, we will pick b with probability $2/5$ and a with probability $3/5$.

The goal of this heuristic is produce improvisations whose feature distribution is more similar to that of the dataset than would be achieved with a purely random choice of transitions (see Section 7 for qualitative experiments assessing this). Many other heuristics are possible, and could give better results in some circumstances. For example, under the simple heuristic above the pattern a $\mathbf{T}$ b contributes equally to the probabilities of transitions on a and b, thereby prioritizing short 'Till patterns. A more sophisticated heuristic could incorporate the number of repetitions of a inside each instance of the pattern, adjusting the transition probabilities accordingly. We also note that random transition heuristics of this kind can be thought of as attempts to enforce a type of divergence criterion similar to the one used in the control improvisation problem, but where divergence is measured against a dataset of multiple songs instead of a single reference word.

In summary, the overall process for generating an improvisation of length at least n is:

1. Maintain a sequence $(q_0, s_0)(q_1, s_0) \dots (q_k, s_k)$ of pairs of states of $\mathcal{A}^p$ and $\mathcal{A}^s$, and a word $w_k = \sigma_0\sigma_1 \dots \sigma_k \in \Sigma^k$.
2. If $k \geq n$ and s_k is accepting, return w_k.
3. If there are no outgoing transitions from s_k, restart the improvisation process.
4. Let C be the set of transitions from q_k that are compatible with s_k (i.e. such that s_k has a transition on the same input symbol).
5. If C is empty, we need to add a transition to $\mathcal{A}^p$. Pick a random transition from q_k using the distribution described above; let σ be the input symbol triggering it and set q_{k+1} to be its destination state. Among all input symbols for which there is a transition from s_k, find the one, say τ, most similar to σ (e.g. among pitches, the nearest in terms of intervals), and set s_{k+1} to be the corresponding destination state. Add a transition from q_k to q_{k+1} on input τ, and set $\sigma_{k+1} = \tau$.
6. Otherwise C is nonempty. Pick a random transition from C using the distribution described above; set σ_{k+1} to the input symbol triggering it and q_{k+1} to its destination state. Set s_{k+1} to the destination state of the corresponding transition from s_k.
7. Repeat from step 1.

6. MUSIC SPECIFICATION MINING

In this section we describe some features that can be used to describe expected musical behaviors or properties and, therefore, are appropriate to mine specifications from. We start by formally defining the components involved.

We abstract and formalize a song into a sequence of *melodies*, where a *melody* is defined as a string of pitched notes and rests, aligned with an *accompaniment*, a sequence of chords with given durations[7]. The time unit is the *beat*, including respective integer subdivisions, and the piece is divided into measures, which are sequences of k beats. We assume that the accompaniment is fixed and our goal is to define an improviser for the melody. Hence, the plant will model the behavior of the accompaniment, without constraining the melody, and the specification FSA will set constraints on acceptable melodies played together with the accompaniment. To encode all events in a score, we use an alphabet which is the product of four alphabets: $\Sigma = \Sigma_p \times \Sigma_d \times \Sigma_c \times \Sigma_b$, where

- Σ_p is the *pitches* alphabet, i.e. $\Sigma_p = \{$𝄽, `a0`, `a#0`, `b0`, `c0`, $\cdots\}$;
- Σ_d is the *durations* alphabet, i.e. $\Sigma_d = \{$♪, ♩, 𝅗𝅥, $\ldots\}$ with ♩ $= 1$ beat. Note that Σ_d also includes fractional durations, e.g., for triplets, as discussed below;
- Σ_c is the *chords* alphabet, i.e. $\Sigma_c = \{$C, C7, G, Emaj, Adim, $\ldots\}$;
- Σ_b is the *beat* alphabet. For example, if the smallest duration (excluding fractional durations) is the eighth note, i.e. half a beat, then $\Sigma_b = \{0, 0.5, 1, 1.5, 2, 2.5, 3, 3.5\}$, where 0 represents the beginning of the first beat in the measure.

Note that the full alphabet enables the creation of data abstractions, such as melodic intervals and tone classes. A similar strategy is used in [Conklin and Witten 1995], where data abstractions (derived types) specific for chorales are implemented. In our current implementation, all pattern graphs implicitly use the full alphabet Σ. However, each component alphabet is meant to address one particular aspect of the music formalization, and we construct the specifications by composing many small specifications which operate only on some of the component alphabets. For example, a specification might constrain only the sequence of beats in the melody, without using the other information in each event, and so could be represented as a small pattern graph defined only over Σ_b.

6.1. Time Domain Features

— **Event Duration**: This feature describes the duration, given in beats, of silences and tones. The event duration feature imposes hard constraints on duration diversity but provides only weak guarantees on rhythmic complexity because it has no awareness of beat location. Figure 6 provides one example where specifications built on this feature fail to prevent incomplete tuplets. We can impose further constraints on rhythmic complexity by combining the features event duration and beat onset location.

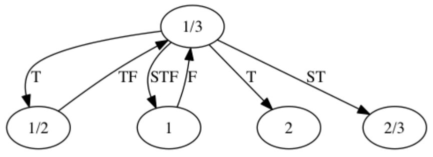


Fig. 6: Selection of event duration specifications learned from the training set. The pattern 1/3 **S** 1 (1/3, 1, 1/3) is allowed but can produce incomplete tuplets if placed on certain beats.

— **Beat onset location**: This feature describes where events happen within the beat, ignoring information about the length of the event. It is computed by taking the remainder modulo 1 of the beat feature, which describes the onset locations of each event. Cooperatively, event duration and beat onset location specifications impose hard constraints on rhythmic complexity that duration specifications alone do not guarantee, and allow for rhythmic diversity that beat onset location alone does not guarantee. These specifications extend our work in [Donzé et al. 2014] by replacing

[7]This is not canonical, and dynamics are not considered in this work, although they could easily be treated as another feature.

handmade specifications designed to ensure rhythmic tuplet completeness with mined specifications. Figure 7 shows an example of the patterns learned and respective patterns between beat onset locations.

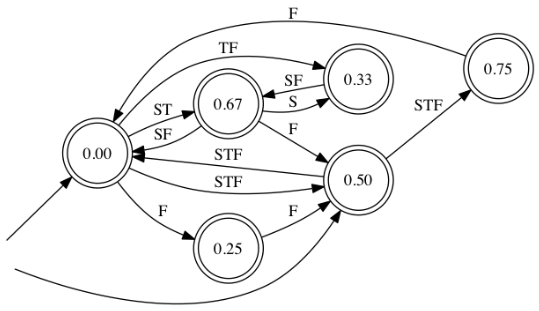


Fig. 7: Beat onset locations specifications learned from the training set.

6.2. Frequency Domain Features

— **Scale Degree**: The scale degree is the identification of a note disregarding its octave but regarding its distance from a reference tonality. We represent scale degree numerically, e.g. in a C scale $C = 0, C\# = 1, \ldots B = 11$. Songs usually impose soft constraints on the pitch space, defining the set of appropriate scale degrees and transitions thereof. The selection of specifications mined from scale degree shown in Figure 8 conform with the general consent that blues songs include the main key's major scale with the "flat seven" (scale degree 10) and the blue note (scale degree 3), excluding, for example flat ninths (scale degree 1) so common in jazz literature. In Figure 8, notice that sharp fourths (scale degree 6) are used as approach tones to scale degree 5 and 6. Since scale degree can only provide overall harmonic constraints to each tone over the scope of the entire song, we use another feature to provide harmonic constraints based on chord progression, therefore increasing the temporal granularity of the harmonic specifications.

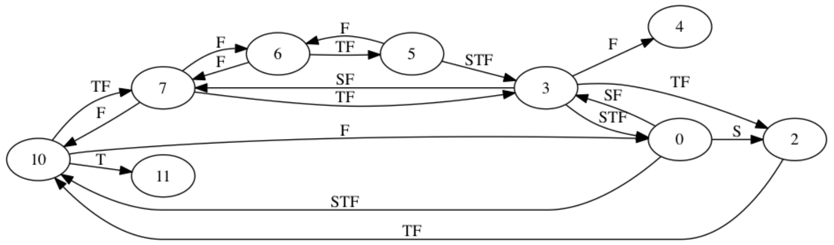


Fig. 8: Selection of scale degree specifications learned from the training set.

— **Interval Classification**: Expanding on [Donzé et al. 2014], we replace the hand-designed tone classification specifications, here called interval classification, with mined specifications. Our specifications include information about the size (diatonic or leap) and quality (consonant or dissonant) of the music interval that precedes each tone. Figure 9 illustrates the mined specifications. We use the symbols A, B, C, and D, to describe tones reached by consonant step, consonant leap, dissonant (non-chord tones) step, and dissonant leap respectively. Consonant and dissonant notes preceded by rests are described with the symbols I and O respectively. The symbol R represents rests. Although scale degree and interval classification specifications ensure desirable harmonic guarantees given key and chord context, they provide no guarantees over the contour of a melody.

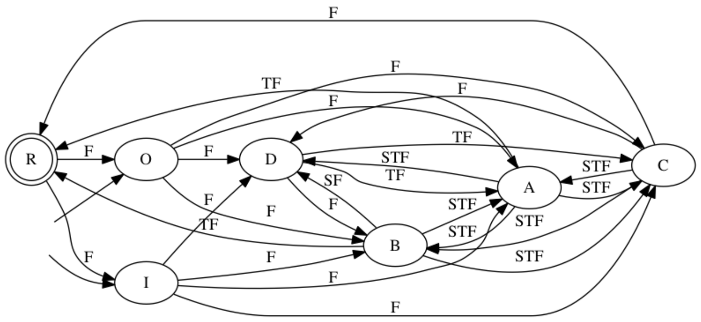


Fig. 9: Interval class specifications learned from the training set.

— **Chord Degree**: The chord degree is the identification of a note regarding its distance in semitones to the root of a chord. It adds harmonic specificity to the interval class, without enforcing a melodic contour.
— **Melodic Interval**: This feature operates on the first difference of pitch values and is associated with the contour of a melody. Combined with scale degree and interval classification, it provides harmonic and melodic constraints, including melodic contour.

	chord	dur	measure	phrase	...	pitch	mel_interval	beat	interval_class
0	F7	14/3	1	1	...	69	NA	1	I
1	F7	1/3	2	1	...	65	-4	5/3	B
2	F7	2/3	2	1	...	67	2	1	C
3	F7	1/3	2	1	...	65	-2	5/3	A
4	F7	1	2	1	...	68	3	1	D
...	...	...	...	...	...	...	...	......	
22	B-7	1	10	3	...	68	3	1	B
23	B-7	1	10	3	...	67	-1	1	C
24	F7	4	11	3	...	65	-2	1	A
25	F7	-4	12	3	...	NA	NA	1	R

Table I: Dataframe from Blues Stay Away From Me by Wayne Raney et al. NA represents a rest or a transition to or from a rest. *dur* is duration in beats and *mel_interval* is melodic interval

Table I provides the reader with a selection of features extracted from a blues song, including chord and phrase number annotations. The next section analyzes in detail the application of specification mining to the tasks of song validation and machine improvisation with formal specifications.

7. EXPERIMENTAL RESULTS

In all our experiments, we used the features above to mine specifications from a training set of 20 blues songs, $\mathfrak{D}^{train}$, digitized from the Real Book of Blues [Long 1999].

7.1. Specification Validation

To check whether our learned specifications are overfitting the training set, we measured the extent to which a disjoint set of songs from the same genre satisfied the specifications. We divide specification violations into two categories:

— the word has a pattern whose symbols exist in the alphabet but the pattern is not allowed by the specification.
— the starting/ending node does not exist in the specification.

An example of the first type is playing an interval that is not valid, although both notes exist in the scale; e.g. the augmented fourth is a forbidden interval in harmony or has to be appropriately resolved. The second type ensures that phrases will start with the proper notes.

We quantify how a test song violates the specification by computing the *violation ratio*, which is the fraction of patterns occurring in the song that are illegal. This quantity is computed by building the pattern graphs for the test song and comparing them with the corresponding graphs learned from the training set. Although the violation ratio is a quantitative measure of how badly the specifications are violated, its formulation is not based on human cognition and perception. Ideally, we would like a measure that takes into account how the violations perceptually differ from the behaviors in the specification. We have developed a prototype of such measure, but leave its evaluation to future work.

In our experiments, we used a separate test set $\mathfrak{D}^{test}$ consisting of 10 blues songs, digitized from the Country Blues songbook [Grossman et al. 1973]. In total there were 975 patterns learned from $\mathfrak{D}^{test}$, of which 124 were violations. In particular, there were 51 chord degree violations, 47 melodic interval violations and 26 interval class violations, yielding a violation ratio of 0.12 for the $\mathfrak{D}^{test}$ dataset with 10 songs. All songs in $\mathfrak{D}^{test}$ had starting and ending nodes that existed in the specification.

Figure 10 provides histograms of violations obtained by using harmonic specifications based on chord degree and interval to validate each song in the test set.

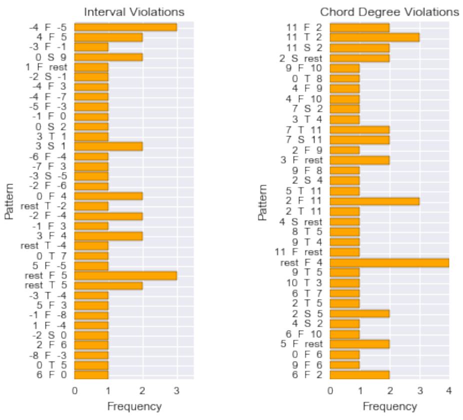
Given the small size of our training data for learning specifications, we assume that these violations would not occur on a larger training set. This validation can be exploited in style recognition and we foresee that more complex validations are possible by creating more elaborated metrics and using a combination of specifications from multiple styles.

Overall, there were many interval violations related to leaps. Although both training and test sets had licks that used chord arpeggiations, their starting notes were different, leading to invalid interval transitions. We provide a specific example in Figure 10 where the first three notes represent an arpeggiation over E7 that starts with an invalid interval. In addition to interval violations, this test set has chord degree violations that are mainly related to playing in sequence two notes that do not belong to the current chord. After analysis, we learned that these invalid transitions occur in blues songs where the second phrase is a repetition of the first phrase under a different harmony. The blues song *You Don't Mean No Good* in Figure 10 has a good example. In that song, the E7 arpeggio on the first phrase, measure 2, is valid under E7 but invalid on the second phrase, measure 6, under A7.

7.2. Machine Improvisation with hard and soft specifications

Using the 12-bar blues excerpt and its chord progression shown in Figure 12, we generated improvisations with and without specifications, generated from $\mathfrak{D}^{train}$, using the factor oracle with 75% replication probability. For this task, we used joint specifications, including duration, beat onset location, chord degree, interval class and melodic interval.

For the quantitative analysis, we computed the average melodic similarity between $\mathfrak{D}^{train}$ and other sets of improvisation, including: 50 factor oracle improvisations generated without specifications, 50 factor oracle improvisations generated with hard specifications and 50 factor oracle improvisations generated with soft and hard specifications. The melodic similarity is computed using



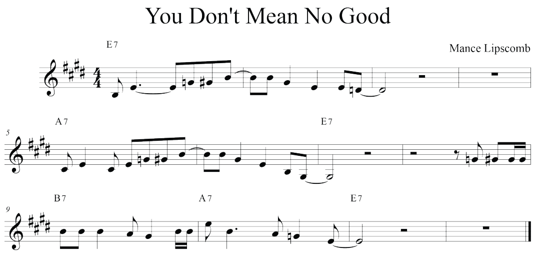


Fig. 10: Histogram of melodic interval and chord degree violations. The y-axis represents the patterns that do not exist in the specification and the x-axis represents their frequency. F and T represent the patterns Followed and 'Till respectively.

the algorithm described in [Valle and Freed 2015]. As baselines, we also computed the similarity of $\mathfrak{D}^{train}$ to the 12 Bar Blues reference word and to 50 songs with random notes and durations.

The results in Figure 11 show that the specifications are successful in making the improvisation generated by the factor oracle more similar to the melodies from which the specifications were mined. In the case of the 12-bar blues, the pitch distribution and the melodic similarity of the improvisations generated with hard specs and hard and soft specs to the training data is almost equal.

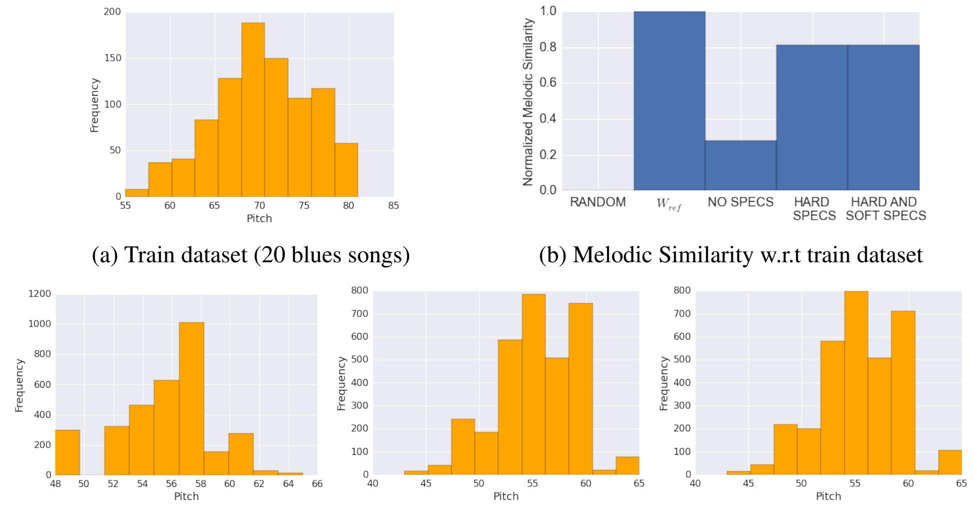
Qualitatively, the improvisation without specifications violates several specifications related to expected harmonic and melodic behavior, as Figure 12 confirms. For example, measure 4 in the improvisation without specifications has chord degrees that violate harmonic specifications. This is expected because the transitions taken by the unsupervised improvisation disregard harmonic context, thus commonly producing unprepared and uncommon dissonant notes. In addition, the melodic profile of the unsupervised improvisation is rather jumpy.

Both supervised improvisations are able to keep overall harmonic coherence despite the use of chromaticism. Their melodic contour is rather smooth and the improvisations include several occurrences of the 'Til and Surrounding patterns, as measures 5 and 1 of the improvisation with hard and with both hard and soft specifications respectively show. We noticed that the improvisations generated with the specifications are considerably similar, which implies that there are not many solutions to the constraints enforced by the specifications. This raises an interesting research question, namely how to ensure that there is enough diversity among the improvisations while still satisfying the constraints.

8. CONCLUSIONS

We proposed a solution to the problem of mining specifications from symbolic music for machine improvisation with formal specifications. This solution replaced our previous approach, which required manually inferring and encoding specifications, with an engine that automatically mines information from a dataset of songs. Our experiments show that the new approach is successful both in graphically and algorithmically describing characteristics of a music collection, and in guiding improvisations in the style of that music collection.

This paper is a first step towards music specification mining and we plan to investigate mechanisms to build more complex specifications, e.g. hierarchical specification schemes where the specification layers have hierarchical relationships such as section, phrase, motif, note.



(a) Train dataset (20 blues songs) (b) Melodic Similarity w.r.t train dataset

(c) 50 improvisations without specifications (d) 50 improvisations with hard specifications (e) 50 improvisations with hard and soft specifications

Fig. 11: Pitch Histograms for the training set and factor oracle improvisations with 0.75 replication probability. The melodic similarity with respect to the training set bargraph (b) has values normalized by the larger similarity value (W_{ref}).

With the purpose of extending specification mining to the audio domain, we are currently investigating the design of specifications based on audio features, such as Chroma and Mel-Frequency Cepstral Coefficients, and the use of neural networks for control.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation Graduate Research Fellowship Program under Grant No. DGE-1106400 and by TerraSwarm, one of six centers of STARnet, a Semiconductor Research Corporation program sponsored by MARCO and DARPA.

References

Rajeev Alur, Pavol Cernỳ, Parthasarathy Madhusudan, and Wonhong Nam. 2005. Synthesis of interface specifications for Java classes. *ACM SIGPLAN Notices* 40, 1 (2005), 98–109.

Glenn Ammons, Rastislav Bodík, and James R Larus. 2002. Mining specifications. *ACM Sigplan Notices* 37, 1 (2002), 4–16.

Gérard Assayag and Shlomo Dubnov. 2004. Using Factor Oracles for Machine Improvisation. *Soft Comput.* 8, 9 (2004), 604–610.

Giordano Cabral, Jean-Pierre Briot, and Francois Pachet. 2006. Incremental Parsing for Real-Time Accompaniment Systems. In *The 19th International FLAIRS Conference, Special Track: Artificial Intelligence in Music and Art*. Florida Artificial Intelligence Research Society, Melbourne Beach, USA.

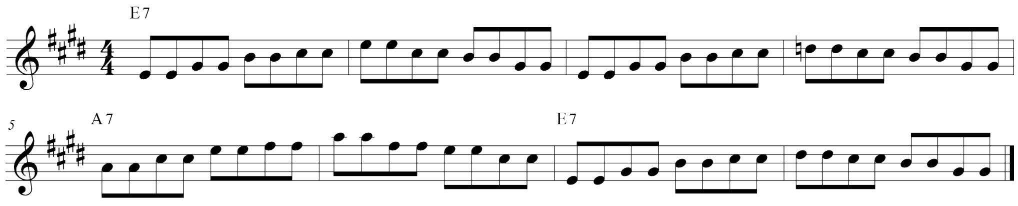
Michel Caplain. 1975. Finding invariant assertions for proving programs. In *ACM SIGPLAN Notices*, Vol. 10. ACM, ACM, 165–171.

Christos G. Cassandras and Stephane Lafortune. 2006. *Introduction to Discrete Event Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

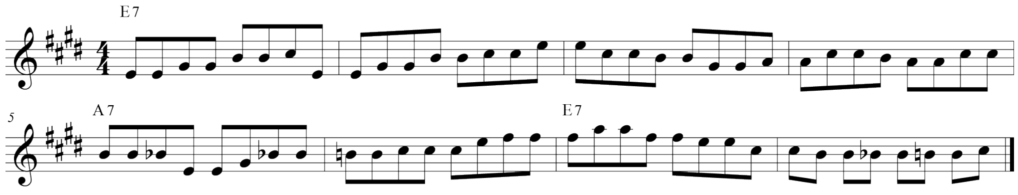
Loek Cleophas, Gerard Zwaan, and Bruce W. Watson. 2003. Constructing Factor Oracles. In *Proceedings of the 3rd Prague Stringology Conference*.

Darrell Conklin and Ian H Witten. 1995. Multiple viewpoint systems for music prediction. *Journal of New Music Research* 24, 1 (1995), 51–73.

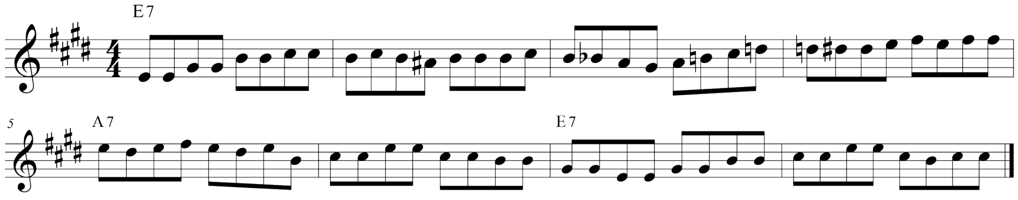
David Cope. 1991. *Computers and Musical Styles*. Oxford University Press.



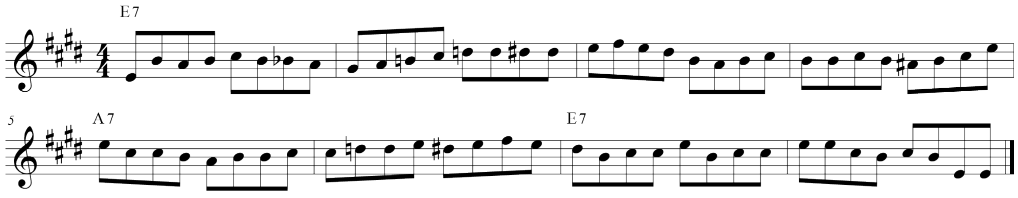
(a) Reference song



(b) Improvisation without specifications



(c) Improvisation with hard specifications



(d) Improvisation with hard and soft specifications

Fig. 12: Factor Oracle improvisations with 0.75 replication probability on a traditional instrumental blues lick

Alexandre Donzé, Ilge Akkaya, Sanjit A. Seshia, Edward A. Lee, and David Wessel. 2013a. Real-Time Control Improvisation for the SmartJukebox. (3 November 2013). http://chess.eecs.berkeley.edu/pubs/1065.html

Alexandre Donzé, Sophie Libkind, Sanjit A. Seshia, and David Wessel. 2013b. *Control Improvisation with Application to Music*. Technical Report UCB/EECS-2013-183. EECS Department, University of California, Berkeley. http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-183.html

Alexandre Donzé, Rafael Valle, Ilge Akkaya, Sophie Libkind, Sanjit A. Seshia, and David Wessel. 2014. Machine improvisation with formal specifications. In *Proceedings of the 40th International Computer Music Conference (ICMC)*.

Shlomo Dubnov and Gérard Assayag. 2002. Universal prediction applied to stylistic music generation. In *Mathematics and music*. Springer, 147–159.

Shlomo Dubnov, Gérard Assayag, Gill Bejerano, and Olivier Lartillot. 2003. A System for Computer Music Generation by Learning and Improvisation in a Particular Style. *IEEE Computer* 10, 38 (2003). http://articles.ircam.fr/textes/Dubnov03a/

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. *Bugs as deviant behavior: A general approach to inferring errors in systems code*. Vol. 35. ACM.

Daniel J. Fremont, Alexandre Donzé, Sanjit A. Seshia, and David Wessel. 2015. Control Improvisation. In *35th IARCS Annual Conference on Foundation of Software Technology and Theoretical Computer Science, FSTTCS 2015*. 463–474.

Mark Gabel and Zhendong Su. 2008a. Javert: fully automatic mining of general temporal properties from dynamic traces. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. ACM, 339–349.

Mark Gabel and Zhendong Su. 2008b. Symbolic mining of temporal specifications. In *Proceedings of the 30th international conference on Software engineering*. ACM, 51–60.

John Gillick, Kevin Tang, and Robert M. Keller. 2009. Learning Jazz Grammars. In *Proceedings of the SMC 2009 - 6th Sound and Music Computing Conference*. Porto, Portugal, 125–130.

Jonathan Reuven Gillick. 2009. *A Clustering Algorithm for Recombinant Jazz Improvisations*. Ph.D. Dissertation. Wesleyan University.

E. Mark Gold. 1967. Language identification in the limit. *Information and control* 10, 5 (1967), 447–474.

E. Mark Gold. 1978. Complexity of automaton identification from given data. *Information and control* 37, 3 (1978), 302–320.

Stefan Grossman, Stephen Calt, and Hal Grossman. 1973. *Country Blues Songbook*. Oak.

Robert M. Keller. 2012. *How to Improvise Jazz Melodies*. http://www.cs.hmc.edu/~keller/jazz/improvisor/HowToImproviseJazz.pdf

Robert M. Keller and David R. Morrison. 2007. A Grammatical Approach to Automatic Improvisation. In *Proceedings SMC'07, 4th Sound and Music Computing Conference*. Lefkada, Greece, 330 – 337.

Wenchao Li, Alessandro Forin, and Sanjit A. Seshia. 2010. Scalable specification mining for verification and diagnosis. In *Proceedings of the 47th design automation conference*. ACM, 755–760.

Jack Long. 1999. The real book of blues. Wise, Hal Leonard.

Jérôme Nika and Marc Chemillier. 2012. Improtek: integrating harmonic controls into improvisation in the filiation of OMax. In *International Computer Music Conference Proceedings*. 180–187.

Francois Pachet, Pierre Roy, Gabriele Barbieri, and Sony CSL Paris. 2001. Finite-length Markov processes with constraints. *transition* 6, 1/3 (2001), 635–642.

Jean-Francois Paiement, Samy Bengio, and Douglas Eck. 2009. Probabilistic models for melodic prediction. *Artificial Intelligence* 173, 14 (2009), 1266–1274.

Rafael Valle and Adrian Freed. 2015. Symbolic Music Similarity Using Neuronal Periodicity and Dynamic Programming. In *Mathematics and Computation in Music*. Springer, 199–204.

Ben Wegbreit. 1974. The synthesis of loop predicates. *Commun. ACM* 17, 2 (1974), 102–113.

Westley Weimer and George C Necula. 2005. Mining temporal specifications for error detection. In *Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 461–476.

Jinlin Yang, David Evans, Deepali Bhardwaj, Thirumalesh Bhat, and Manuvir Das. 2006. Perracotta: mining temporal API rules from imperfect traces. In *Proceedings of the 28th international conference on Software engineering*. ACM, 282–291.