Description-Based Design of Melodies

François Pachet

Sony CSL-Paris

6, rue Amyot

75005 Paris

France

Email: pachet@csl.sony.fr

TEL: +33 1 44080516

FAX: +33 1 45878750

Description-Based Design of Monophonic Melodies

Abstract

We introduce a novel paradigm for creating musical objects, based on the combination of a machine-learning and a combinatorial algorithm. In this scheme users associate freely subjective descriptions (tags) to musical objects. A machine-learning component continuously learns a mapping between these tags and a set of technical features. Thanks to a combinatorial generator, the user can then reuse these tags to modify other objects, in an incremental manner. In this scheme, to the traditional construction or programming task is substituted a description task integrated with an interactive naming game. We describe the paradigm and its application to the construction of simple monophonic melodies. We show that the approach allows users to create "interesting" melodies of various types without requiring any form of explicit programming. This approach to music composition lessens the need for technical skills from the composer, while exploiting fully his capacity to express and manipulate consistent subjective judgments.

1. Background

Most of the current approaches in computer-aided composition (CAC) are based on an explicit construction paradigm: users build musical objects by assembling components using various construction tools. Virtually all the technologies developed by computer science and artificial intelligence have been applied to CAC; thereby progressively increasing the sophistication of music composition tools. Composers can choose between many programming paradigms to express the compositions they “have in mind”, from the now standard time-lined sequencers (e.g. Steinberg’s Cubase) to advanced programming languages or libraries (e.g. OpenMusic, Assayag et al. 1999).

Although these explicit constructions do benefit from abstractions of increasing sophistication (e.g. objects, constraints, rules, flow diagrams, etc.) CAC always remains based on an explicit construction paradigm: users must give the computer a clear and complete definition of their material. This approach has the enormous advantage of letting users control all dimensions of their work. However, it also requires from users a fine understanding of the technicalities at work. For instance, composing music with object-orientation requires the understanding of objects, classes and message passing. Using constraints requires the understanding of constraint satisfaction, filtering and of the basic constraint libraries, etc. An interesting attempt to escape these technical requirements is the Elody system (Letz et al., 1998) in which user can create arbitrary abstractions by selecting a musical material together with a specific dimension of music (e.g. pitch or rhythm). These abstractions can then be applied to other musical material to create yet more complex objects. But here again the user has to mentally maintain a model of the abstraction algorithm at work, a task which can be particularly difficult as the complexity of the composition grows. Other approaches propose construction tools that do not require explicit programming skills. For instance, (Hamanaka et al., 2008) propose a morphing metaphor in which melodies can be created as interpolations between two given melodies. But this approach is limited to the context of the generative theory of tonal music (Lerdahl & Jackendoff, 1983), and is not extensible to arbitrary categories as we will show here.

We propose here a novel approach to music composition called *Description-Based Design* which aims at removing the need for the user to understand anything technical related to his target objects. In this paper we focus on the creation of simple musical objects - monophonic melodies - as a working example, but our paradigm is general and may be applied to many other fields of design.

In Section 2 we introduce the general description-based design mechanism. In Section 3 we describe the type of melodies we target, and in Section 4 we describe experiments demonstrating the functioning of the algorithm and its potential.

2. Description-Based Design

Description-based design stems from the paradigm of Reflexive Interaction (Pachet, 2008). The idea is to let users manipulate images of themselves, produced by an interactive machine-learning component. The creation of objects (musical objects in our case) is performed as a side-effect of the interaction, as opposed to traditional interactive systems in which target objects are produced up-front, as the result of a controlled process. A typical example of reflexive interactive system is the Continuator (Pachet, 2004), a system which learns continuously stylistic information coming from the user's performance, and generates music "in the same style" in the form of real-time answers to, or continuations of, the music performed by the user. The Continuator was shown to trigger spectacular interactions with professional Jazz musicians (Pachet, 2004) as well as with children (Addessi & Pachet 2005) involved in free, unstructured improvisation. However, this type of interaction shows limitations when users want to structure their production, in other words, when they want to shift from improvisation to composition.

Description-based design adds a further component to the Continuator-like interaction by introducing an explicit linguistic construct, precisely aiming at addressing this "structure" problem inherent to free form improvisation systems. The idea is the following. In a first phase, the system generates objects, melodies in our case, randomly or according to specific generators. The user can then freely tag these objects with words, e.g. "jumpy", "flat", "tonal", "dissonant", etc. Each object can be tagged by several words, or by none. In a second phase, the user selects a starting object (say, a flat melody), and one of his tags (say "jumpy"). He or she can then ask the system to produce a new object which will be "close" to the selected one, but "more jumpy", or "less jumpy". More generally the user can reuse any of his tags to modify a given object *in the semantic direction* of the tag. The system will then attempt to generate a new object that optimally satisfies two conditions: 1) being "close" to the starting object and 2) increasing (or decreasing) the probability of being of a certain tag. The new object (in our case, a progressively jumpier melody) is then added to the palette of objects created by the system. It can, in turn, be tagged or refined at will. The design activity is therefore strictly restricted to 1) tagging objects and 2) creating variations using these tags. The implementation of this scheme requires a combination of components that we briefly describe here.

1. Object generator

We call the objects that the user wants to produce *target objects*. In our context these objects are defined by 1) a set of *technical features* that describe these objects and 2) a generator that produce sets of objects. The generator is a program that should be able to randomly generate every possible object of interest. The choice of the feature set will influence the capacity of the system to learn faithfully user tags, but identifying reasonable feature set is usually straightforward. These two ingredients are therefore easy to design. We give the details for the particular case of monophonic melodies below.

2. A Machine-learning Tagging System

The second component is a tagging system, associated with a machine-learning algorithm that learns a mapping between user tags and the feature set. In our case, this machine-learning component is a Support Vector Machine (SVM). SVMs are automatic classifiers routinely used in many data mining applications (Burges, 1998). In the training phase, a SVM builds an optimal hyper-plane that separate two classes, so as to maximize the so-called “margin” between the classes. This margin can then be used to classify new points automatically using a geometrical distance as illustrated below. SVMs have been used extensively to learn music information, in particular in the audio domain, typically using spectral features (Mandel et al., 2006). In our case, we use them to learn classes from symbolic features, as described below.

The tags are entered by the users as free text. To each tag is associated a SVM classifier, which is retrained each time a user adds or removes a tag for an object. To avoid undesirable effects such as over-fitting, a *feature selection* algorithm is applied prior to the training phase. We use the IGR (Information Gain Ratio) algorithm (Quinlan, 93) by which only a limited number of features are kept, maximizing the “information gain” of each retained feature. Once trained, this classifier can compute a probability for that tag to be true for any object. Similarly, the classifier computes the probabilities, for a selected melody, of all learned tags, according to the current state of the system in the session. More precisely, for a given tag, we use Boolean classifiers, trained on positive and negative examples. Positive examples are all the objects having been tagged by the user with this tag. Negative examples are chosen automatically by the system, according to various heuristics (notably, it chooses approximately the same number of negative examples than positive ones and chooses only objects which have been tagged, obviously with other tags than the tag considered).

Of course, the accuracy of this prediction depends on many factors including the feature set, but also the number of examples, i.e. objects having been tagged by the user. In the experiment described in this paper we show empirically that the predictions are satisfactory after about 70 examples for each category, but this result is not general.

A crucial aspect of the classifier to work in our context it that it should yield, for a given item, not only a class membership, but a *probability* of membership. Support Vector Machines (SVM) are an appropriate framework in this case as they precisely transform a classification problem into a geometrical distance problem. Once trained to classify between two classes, say A and B, a SVM identifies a set of support vectors, which define an optimal margin between A and B, as illustrated in Figure 1. The classification decision for an item I is then made on the basis of the distance of I to the hyperplane defined by the support vectors. As a consequence, one can interpret this distance as the inverse of the probability for I to belong to A (resp. to B). This is particularly true for items which are “outside” the hyperplane, i.e. who are not classified as belonging to the class under consideration.

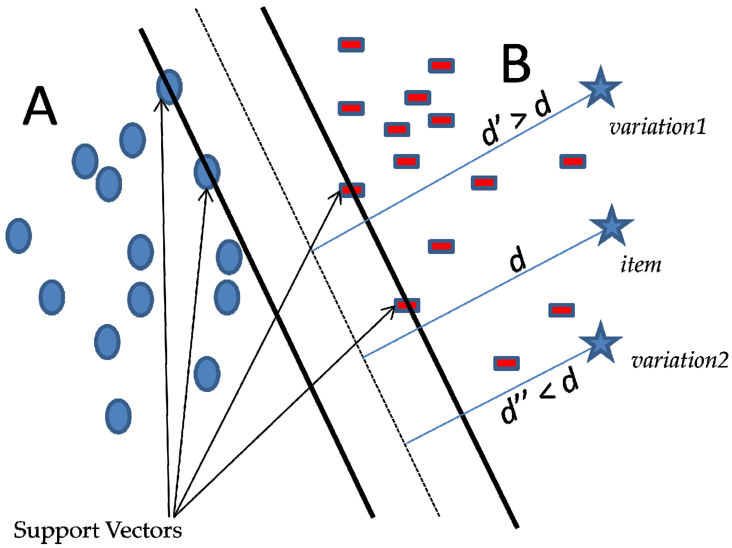

Figure 1. A Support Vector Machine defines a class separation in a *n-dimensional* space as an hyperplane of dimension *n-1*, dividing the space into two regions. In this figure, the heavy lines are defined by particular “margin” points called *support vectors* and define the *margin*. The lighter line, in the middle of the margin, represents the class separation itself. The distance between a point and the hyperplane is traditionally interpreted as the inverse of the probability for the item to belong to the class outside the boundary. In this case, *variation2* is closer to A than *variation1*, so its probability to belong to A is greater. Note that the distance between *item* and its variations (*variation1* and *variation2*) is not necessarily meaningful.

3. A combinatorial generator

The task of the combinatorial generator is to generate variations of an object that maximize two properties: 1) being as close as possible to the initial object and 2) increasing (resp. decreasing) the probability of a given tag/classifier by a given ratio. A naïve, combinatorial version of this algorithm is given in Figure 2:

FindMoreOfTag (Source, Tag, ratio)

```
1.  initial_Tag_Prob := probability that Source is classified as Tag;
2.  Variations := N randomly generated variations of Source;
3.  Sorted_Variations := Variations sorted according to distance to Source
4.  For all V in Sorted_Variations do
5.      Prob_V := classify V according to Tag.
6.      If (prob_V > (initial_Tag_Prob * (1 + ratio)) return V.
7.  If no object was found with a probability of Tag being greater than initial_Tag_Prob then
    report failure, or restart with greater value of N.
```

Figure 2. The combinatorial search algorithm. The "FindLess" is similar. Extension to compound commands is straightforward. N is a predetermined number of variations to be generated.

The generation of variations is domain-specific. We describe a simple variation generator for melodies below. Increasing the probability of the variation to belong to the class represented by the tag is done in our case by exploiting the probability given by the SVM, as described in the previous section. Sorting generated variations according to the distance to the initial object is a crucial step, as it ensures that the resulting variation will be as close as possible to the starting object. There are several ways to implement such a distance. One is to use the feature set described in Section 2, which defines a natural distance between two objects (e.g. an Euclidean distance). The feature space can also be transformed, e.g. by the kernel used for classification. In our example, a standard Radial Basis Function (RBF) kernel was used. However, the distance between two items given by a SVM is not necessarily appropriate, as kernel transforms aim primarily at optimizing class separation, and not at defining a meaningful distance between items of the training set.

A better option is to introduce a domain-dependent distance. In the case of melodies, we use a Levenshtein distance (Levenshtein, 65) on the pitch sequence, as described below.

A simple extension of this algorithm is the use of "compound commands". Arbitrary Boolean expressions can be formed from basic "more" or "less" commands, such as "*more* T_1 AND *less* T_2 AND *as* T_3" (the tags T_n are typically adjectives). Such an extended Boolean expression can be easily substituted to the test of line 6 in the pseudo code of Figure 2. An example is given in Section 4 where we generate a "*more* long AND *as* tonal" melody.

We will now describe an application of our scheme to the construction of melodies.

3. The case of Monophonic Melodies

4. Five types of melodies

Melodies are a good example to illustrate our approach because they are both technical objects and subjective ones. There are many known technical features to describe melodies, notably related to pitch distribution, repetition or tonality. There are also many subjective appreciations one can think about to talk about melodies (simple, jumpy, linear, annoying, dissonant, etc.). There is furthermore no simple way to associate these subjective expressions to the technical features, especially for non musicians. Even for trained musicians, finding the “right melody” can sometimes be an extremely difficult task. So melodies are an ideal playground for description-based design.

In this experiment, we restrict ourselves to the composition of 4-bar monophonic melodies, with a maximum of 4 possible notes durations (quarter, half, dotted half and whole notes) and a pitch range of [60, 80] (in Midi pitch). Figure 4 gives an example of such a melody. Although these restrictions may appear drastic, compared to real melodies, these constraints still define a search space of more than 20^{16} possible items, huge enough to justify the use of our framework.

The aim of the experiment described here is to demonstrate that the algorithm proposed and described in Section 3 essentially works, i.e. does produce “close variations” that increase the probabilities of melodies to be of an arbitrary subjective category. To this aim, we chose not to consider arbitrary subjective categories, but limit the experiment to 5 “controlled” categories: *tonal*, *brown, serial* as well as *long* and *short*. The justification for this choice comes from the clarity of the definition of each of them, which allows us to test the results non-ambiguously. More precisely we introduce the following (possibly overlapping) categories:

> *Tonal* melodies are melodies having a clear tonal center. Although the notion of tonality has long been an object of debate in musicology as well as in cognitive

science (Temperley, 2007), it is quite easy to produce melodies with a clear tonal center and we give below a simple algorithm to do so. An example of a tonal melody is given below in Figure 3.

Figure 3. A typical tonal melody: "My rifle, my pony and me" (sung by Dean Martin and Ricky Nelson), here, stylized.

Brown melodies are melodies with only small intervals. The brown term is borrowed from the famous experiment of Voss and Clarke (1978), who compared random, Brownian and 1/F melodies. We give below the description of a simple algorithm that generates brown melodies. A typical example of a brown melody is the song "With a little help from my friends" by Lennon & McCartney (see Figure 4).

Figure 4. The melody of "with a little help from my friend" (here, stylized) is a typical brown melody.

Serial melodies are defined here to be melodies in which all pitches occur with equal frequency. Serial melodies are rarely used in popular music, but this category is useful for our demonstration as it bears a non-ambiguous definition.

> Typical serial melodies are dodecaphonic melodies, in which all 12 pitch classes are used.
>
> Additionally, we introduce two categories: *long* and *short*. These categories are simply related to the number of notes. Like the preceding ones, they will be defined only by a set of examples.

72 examples of each of the three categories (*tonal*, *serial* and *brown*) are generated and given to the system, with the corresponding tags. Examples of these generated melodies are given in Figure 5. The generators are defined as follows. For each generator, the basic operation described below is repeated *nb* times, where *nb* is a random number between 0 and 16.

The tonal generator draws randomly notes from a random scale (e.g. C major). Each note falling on a beat is chosen from the triad of the scale. The other notes are chosen randomly from the scale. The duration is each note is randomly selected between quarter and half notes.

The serial generator randomly draws a pitch from an initial list of all possible pitches. It then removes it from the list and repeats the operation. When the list is empty, the list is filled again. This ensures a "maximally" serial melody, given the pitch range.

The brown generator starts from a random pitch. It then randomly draws an interval in { -1, +1}, and adds it to the preceding pitch.

In all three cases, the notes' MIDI velocities values (corresponding roughly to loudness) are random integers taken in the range [70, 100].

Figure 5. Examples of (resp.) *tonal*, *serial* and *brown* melodies, generated by our three generators.

The variation generator we use is a deliberately simple and agnostic algorithm. Starting from a set containing only the initial melody, it generates a variation by picking up randomly one of the following three modifications:

1. Modify the pitch of a randomly selected note,
2. Insert a random note with a random pitch and velocity,
3. Remove a randomly selected note.

The resulting variation is then added to the set and the process is repeated by picking up randomly a new "seed" melody from the updated set. This procedure creates, by definition, variations of various "depths", i.e. very similar as well as very different melodies. In the experiments described here, the number of variations produced and explored is set to 20,000.

The set of features we use for describing melodies is the following table:

Number of notes
Mean value of the *pitch* sequence
Mean value of the "*pitch interval*" sequence
Mean value of the *velocity* (Midi information)
Tonal weight. This feature gives an indication of how tonal is a melody. It is computed using a "pitch profile" algorithm (Krumhansl, 1990): for each possible 12 major scales, it counts the number of notes of the melody which are in this scale. It returns the maximum value of this count.
Pitch compressibility ratio. This feature gives an indication of how repetitive is the melody. It uses a data compression algorithm as used in the Continuator (Pachet, 2004). Its value lies between 0 (no repetition at all) and 1 (a sequence with the same note repeated throughout).
Interval compressibility ratio. This feature is the same as the previous one, but applied to the sequence of intervals rather than pitches

Table 1. The set of features used to represent and learn melodies.

Many other melody features could be introduced but we restrict ourselves to this list in the context of this experiment. As we will see below the velocity feature is not used in this particular experiment and is just inserted here to show the robustness of the algorithm. As mentioned above, a feature selection algorithm is applied on the feature set prior to learning, to select the most meaningful features given the set of examples and counter examples for a given tag. This feature selection has the extra advantage of giving an indication about how the classifier has generalized from the examples.

4. Experiments

We first generate a set of examples using our three melody generators: *tonal*, *serial* and *brown*. We then train the corresponding classifiers on these examples. Finally we introduce the “long” and “short” categories by tagging the generated melodies accordingly, and also train the corresponding *long* and *short* classifiers. After this step, the system is able to predict each of these five categories for any, possibly untagged, melody. We then perform a series of experiments using these generated melodies as starting points and the classifiers as modifiers.

5. Training phase

After the training phase, each of the five categories has been trained on approximately 72 positive examples and 72 negative examples. The negative examples are chosen automatically for the system, for the *brown*, *tonal* and *serial* categories, by picking-up random melodies which are not tagged with the corresponding tag. In the case of the *long* and *short* categories, we help the system in telling it to use *lon*g examples as negative examples for *short*, and conversely. This trick is to avoid having “bad” counter-examples, as some generated melodies could turn out to be long (or short) without being tagged as such.

As a result, we give here the result of the feature selection process applied for each tag (only the first four most significant features are kept). This gives an indication of which features were selected by the classifier, and with which weight. These numbers indicate "how well" the classifiers have understood the semantics of each generator. The most important features for *brown*, *long* and *short* do fit with the corresponding semantics of the generator. It can be observed that the "tonal" classifier did use the feature "tonalWeight", but not in first position. This slight discrepancy is due to the limited number of examples given for training. It has a small incidence on the process, as shown in Figure 13.

	Tonal	**Brown**	**Serial**	**Long**	**Short**
1	0.791 meanPitch	1 meanPitchInterval	0.795 meanPitch	1 nbNotes	1 nbNotes
2	0.788 tonalWeight	0.948 intervalCompRatio	0.649 pitchCompRatio	0.67 meanPitch	0.75 intervalCompRatio
3	0.546 meanPitchInterval	0.437 meanPitch	0.489 meanPitchInterval	0.586 tonalWeight	0.723 meanPitchInterval
4	0.49 pitchCompRatio	0.139 tonalWeight	0.399 tonalWeight	0.264 pitchCompRatio	0.634 meanPitch

Table 2. The feature selection mechanism applied to our 5 categories.

In a second step we will now consider a series of use cases, illustrating the use of classifiers as melody constructors using the description-based algorithm. In particular we show that the algorithm is able to produce variations that would otherwise be found only by very specific programs.

6. Tonalizing a serial melody

The first example consists in starting from a serial melody (Figure 6) and making it progressively more tonal. Figures 8-11 illustrate the process step-by-step. At each step, a new melody is generated which as both close to the preceding one and slightly more tonal. The initial melody is generated with the serial generator. The last one is optimally tonal while being still “close” to the original. We indicate the probability of each classifier (*tonal* and *serial*) as well as the effective measure of *tonalness*. Note that such a measure is usually impossible to get with arbitrary categories, hence the use of control categories for this experiment. These figures and the resulting melodies indicate clearly that 1) the system has correctly learned the notion of *tonal* and *serial* and more importantly 2) that it is able to use these classifiers as melody generators controlled by the tags.

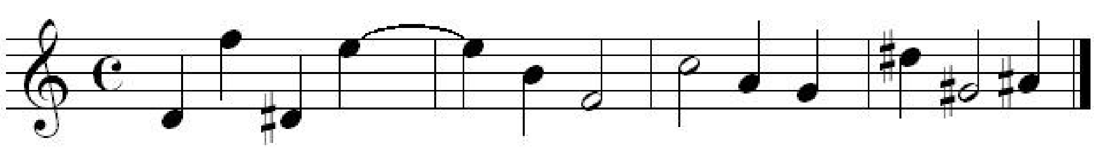

Figure 6. Initial melody created by the *serial* melody generator. The melody is perfectly serial according to our definition (all pitches are different, though here not all pitch classes). The “serial” classifier yields a probability of 1.0. The “tonal” classifier yields a probability of $8*10^{-3}$.

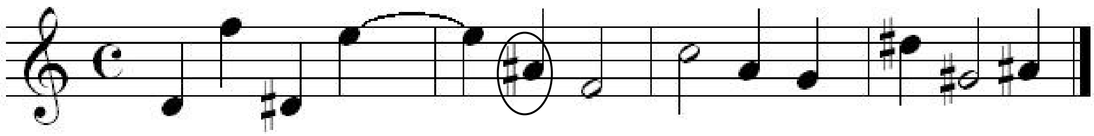

Figure 7. The same melody, a bit "more tonal". The differences are highlighted. The "tonal" classifier has increased its probability to 5.7 * 10^{-2}.

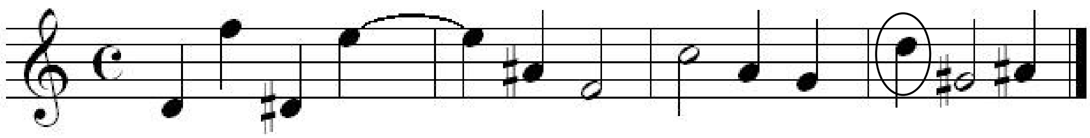

Figure 8. Still a bit "more tonal". Probability is now 1.4 * 10-1.

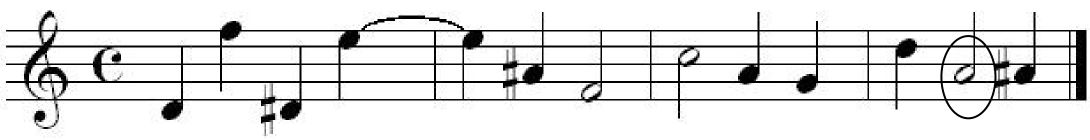

Figure 9. Again, more tonal. Probability is now 3.11 * 10^{-1}.

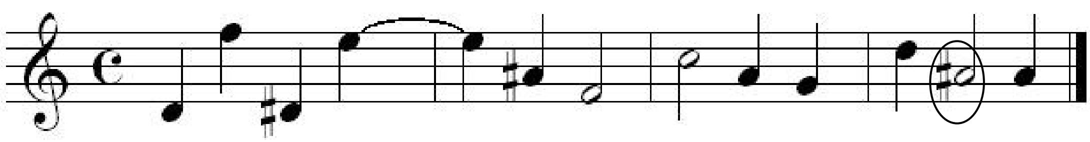

Figure 10. More tonal again. Probability is 5.3 * 10^{-1}.

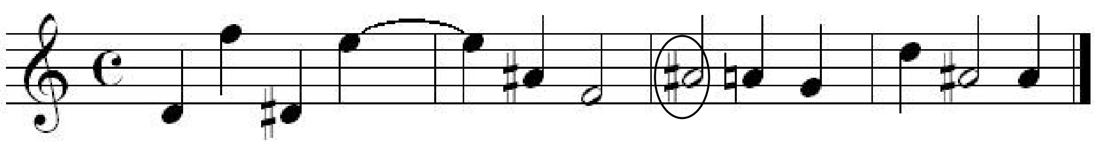

Figure 11. Probability is now 7.7* 10^{-1}.

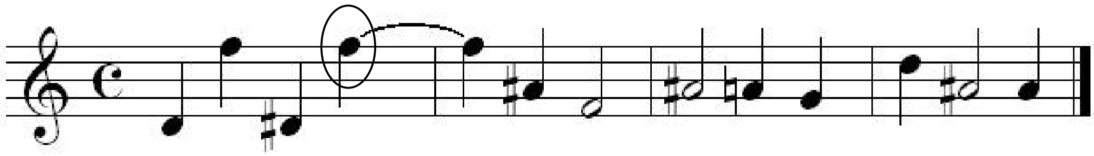

Figure 12. Final step, the melody is now perfectly *tonal* (in Bb, with a probability of the tonal classifier of 9.6 10^{-1}), yet "similar" to the initial serial melody. The accidentals are displayed without correction and thus do not reflect the tonality, which contains flats rather than sharps.

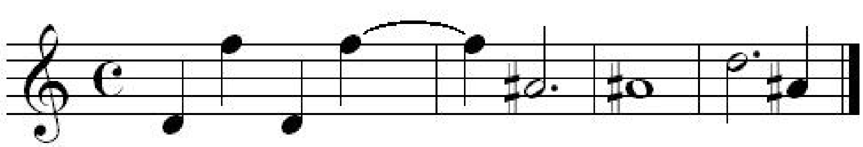

Figure 13. A slightly more "tonal" version of the melody. The melody is, strictly speaking not more tonal than the preceding one. However, because the notion of tonalness was not learnt perfectly by the classifier, the algorithm found an artificial way of improving its "understanding" of tonalness by reducing the number of notes.

Table 3 indicates the progressive increase in tonalness at each step of the process. This increase is confirmed by the increase of "real" tonalness of the melody, as computed by the *tonalWeight* feature (described in Table 1).

Melody versions	Serial classifier	Tonal Classifier	Tonalness
1 - Initial	1.0	$8 * 10^{-3}$	0.54
2 - More tonal	1.0	$5.7 * 10^{-2}$	0.625
3 - More tonal	1.0	$1.4 * 10^{-1}$	0.66
4 - More tonal	$9.95 * 10^{-1}$	$3.11 * 10^{-1}$	0.70
5 - More tonal	$8.15 * 10^{-1}$	$8.15 * 10^{-1}$	0.75
6 - More tonal	$1.82 * 10^{-2}$	$7.7 * 10^{-1}$	0.79
7 - More tonal	$3.87 * 10^{-5}$	$9.63 * 10^{-1}$	0.87

8 - More tonal	$1.72 * 10^{-7}$	$9.98 * 10^{-1}$	1.0

Table 3. The progressive increase in "tonalness" at each step of the process leading to sequences in Figures 7-13.

7. Stretching a tonal melody

The second experiment consists in using description-based design to stretch a melody, i.e. "adding more notes". Of course, an easy solution to this problem consists in programming explicitly a function to add notes to a given melody. But we can again avoid the use of such an explicit programming. To this aim, we can reuse the two tags: "long" and "short", trained with examples generated with the other three generators. By convention, we tag melodies with less than 8 notes as *short*, and melodies with more than 12 notes as *long*. As a consequence, the system automatically learns *long* and *short*, with examples coming from all three generators. We can check that the system has correctly learned the tags *long* and *short* by observing the selected features, as indicated in Table 2: The feature "number of notes" was selected as a primary feature for "long" and "short". This feature selection process also shows incidentally that the system has however not simply associated *long* and *short* to the number of notes, but to a more complex configuration of features. For instance, it turns out that most of the *long* melodies also have, by definition, more repetition in their interval sequence. The system has no way to generalize better in our context ("better" would be to only consider the feature "*nbNotes*"). But as we will see, this approximation does not prevent it to produce "meaningful" variations.

We now consider a melody which is tagged both as *tonal* and *short*, illustrated in Figure 14 (first melody). In a first step we will make it longer as in the previous experiment, i.e. through the command *more long*. As we can observe, this command indeed results in a similar melody, with more notes. The sequence of "more long" commands is illustrated in Figure 14, and it can be noted that the melodies have all indeed progressively more notes (from 7 initially to 8, 9 and 10).

Figure 14. A *short* and *tonal* melody as a starting point for repeated "stretching" operations. The probability of being "long" is initially 1.06* 10^{-7}. Successive probabilities of being "long" are 1.25 * 10^{-3}, 9,77 * 10^{-1} and 1.0. The latest melody cannot be stretched any further as its probability of being long is 1.0.

However, it can also be noted that the notes which have been added to the melody by the combinatorial algorithm make it not tonal any longer: the initial melody is in C, but added notes (D#, C# and F#) are not in C. This is highlighted by the fact that the corresponding probabilities of being *tonal* have shifted from .99 to .07 (see Table 4). This phenomenon is normal, as the system has just been asked to make the melody "more long", but was not given any constraint on tonality or on any other property.

A natural way to address this problem consists in issuing a compound conjunctive command of the form "more long AND as tonal". Such a query is made by selecting the tags and the corresponding modifiers through a specific interface (Figure 16). In this case, starting from the same melody, we obtain the melodies in Figure 15. We can observe that the algorithm has now progressively added only tonal notes (F and G). Most importantly, these added notes have been chosen as a "side-effect" of the compound command, and not through the introduction of an explicit representation of tonality in the program.

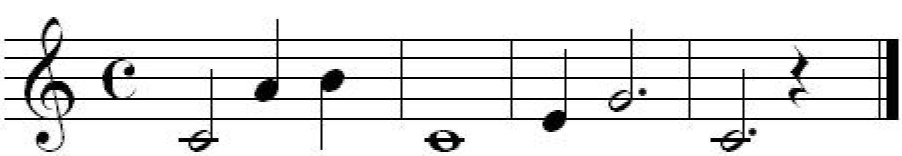

Figure 15. The short and tonal melody now progressively made "more long and as tonal". Respective probabilities of being "long" and "tonal" are given in Table 4. The final melody is 1) close to the original, 2) longer and 3) still as tonal as the starting one.

Melody versions	**long**	**tonal**	**Number of notes**
1 : Short and tonal	$1.0 * 10^{-7}$	$9.9 * 10^{-1}$	7
2 : 1 More long	$1.25 * 10^{-3}$	$9.2 * 10^{-1}$	8
3 : 2 More long	$9.77 * 10^{-1}$	$6.13 * 10^{-1}$	9
4 : 3 More long	1.0	$7. * 10^{-2}$	10
5 : 1 more long AND as tonal	$3.8 * 10^{-5}$	9.9	8
6 : 5 more long AND as tonal	$1.4 * 10^{-1}$	$9.96 * 10^{-1}$	9
7 : 6 more long AND as tonal	1.0	$9.87 * 10^{-1}$	10

Table 4. Variations on the starting "short and tonal" melody. Variations 2 to 4 are obtained by applying the "more long" command. Variations 5 to 7 are obtained by applying the "more long AND as tonal" command to the same initial melody.

Figure 16. An interface for specifying conjunctive compound queries holding on several tags simultaneously. For each tag the user can specify the type of action (*as, more, less* or *ignore*) and the corresponding ratios. Here, a query to produce an item which is "more long by 15%" AND "as tonal by 10%, while the other tags are ignored.

Note that this compound query triggers a non trivial search. Figure 17 illustrates the search process corresponding to "more long and as tonal" query. At each iteration (X axis) the probabilities for tags "long" (dashed line) and "tonal" (plain line) are displayed. A solution is found when the "tonal" plain line is within the two horizontal dashed lines (which represent the bounds +/- 10% of the starting tonalness) and simultaneously the "long" (dashed) line is above the horizontal large line (which represents more long by 15%). It can be seen that the system explores about 300 melodies before finding a solution.

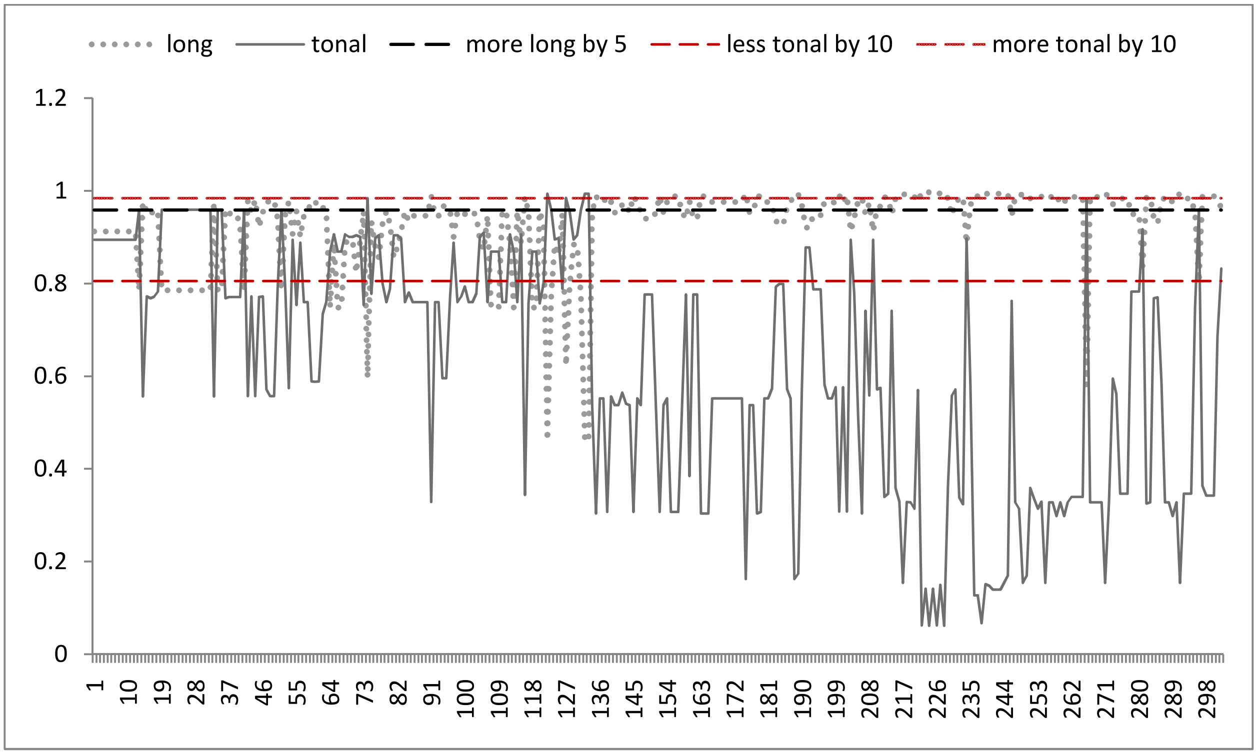

Figure 17. The evolution of the search process during the "more long by 15%" AND "as tonal by 10%" query.

8. Making a tonal melody more brown

The last example consists in starting from a tonal melody and making it progressively more "brown". We illustrate again the process step-by-step in Figure 18 as well as the increase in brownness in Table 5.

Figure 18. The various steps in making a tonal melody "more brown". Note that at step 6 the algorithm finds no other solution to increase brownness than to remove a note, to later add another note back (step 8). At the last step, the only way to improve (slightly brownness) is to remove a note (step 12).

Melody versions	brown
1 : tonal	0.0
2 : 1 More brown	0.0
3 : 2 More brown	$6.65 * 10^{-35}$
4 : 3 More brown	$4.67 * 10^{-33}$
5 : 4 More brown	$4.14 * 10^{-32}$
6 : 5 More brown	$1.15 * 10^{-29}$
7 : 6 More brown	$2.88 * 10^{-28}$

8 : 7 More brown	$4.71 * 10^{-22}$
9 : 8 More brown	$3.31 * 10^{-20}$
10 : 9 More brown	$1.41 * 10^{-8}$
11 : 10 More brown	$9.95 * 10^{-1}$
12 : 11 More brown	1.0

Table 5. The progressive increase in brownness, starting from an initial tonal melody.

This experiment shows again that the probability of a given tag (here, brown) does increase after each modification query. It can be observed that the resulting melody is indeed more Brownian in the sense that intervals are getting smaller in average. There is a limit obtained by the system, which cannot increase brownness further after step 12 as the probability reaches 1.0, although one could imagine further small modifications of the melody to make it more Brownian (e.g. lowering the initial G#). This may be explained by the fact that the brown classifier has either not enough examples (and counter-examples), or that the features chosen in this experiment are not able to fully grasp the notion of brownness. However, the classifier learns enough to produce meaningful "small variations". Furthermore the user has the possibility at any step to tag the resulting new melodies, and retrain the classifiers to continuously fine-tune the system.

5. Discussion

We have introduced description-based design as a novel way of building musical objects, which does not require any form of programming knowledge from the user. The only programming constraint lies in the variation generators: they have to be designed in such a way that they generate, at least, the target objects (together with possibly many unwanted objects). However, this is a relatively weak constraint as these generators can be designed once for all, for a particular domain (here, melodies). Of course, more or less efficient generators could be considered, but default naïve generators are easy to design.

This paper only aims at demonstrating the nature of the underlying algorithm using simple, well-understood examples. The experiment presented here used controlled categories to illustrate the algorithm and show its capacity to produce musical objects, without explicit programming or editing. The approach is, in essence, more suited to the use of subjective, non-controlled descriptions, and reaches its full potential when these descriptions are collected massively from social tagging systems. Such an experiment is currently under way, using tags collected from a *melody competition* web site. In this context, users can both produce melodies, tag the melodies of others, and reuse tags for modifying melodies. Another application of this paradigm is to use the system to control the generation of Jazz improvisations, as an extension of the

Continuator system (Pachet, 2004). In this latter case, subjective tags are used as control *handles* to influence, in real time, the quality of generated solos.

Note that the working example showed here is based on SVM, but other classifiers could be used. Decision Trees for instance, have been shown recently to exhibit better geometrical properties than SVMs (Alvarez et al., 2007). They could be substituted to SVMs without changing the framework.

The algorithm we propose is blind, bearing some similarity with other blind search algorithms like genetic algorithms (GAs), often used in music generation (Biles, 1994). GAs could indeed be used to build our variations, instead of specifically designed random generators. However GAs are more difficult to control than random generators. Most importantly, we believe that our currently naïve search algorithm can be optimized by exploiting information about the features used by the classifier, and this constitutes a current avenue of research. Such an optimization is not possible, by definition, in GAs, which operate on chromosomes, which are independent of the feature sets used by the classifiers.

Description-Based Design attempts to bridge the gap between description and construction, thereby reducing the need for users to learn the technical languages of the objects they have "in mind". This approach is well suited to domains in which 1) features to describe objects are known and accurate and 2) users have the capacity to easily express subjective judgments in a consistent way. This applies to most of the musical objects created in the context of computer-assisted composition. For instance description-based design is currently being applied to other types of musical objects, in particular chords, chord sequences, harmonized melodies.

Audio synthesis is also being investigated. For instance, programming FM sounds require notoriously complex knowledge of FM synthesis. Several approaches have attempted to provide users with more subjective means of programming sound synthesizers (Rolland & Pachet, 1996) (Sarkar et al., 2007). But these approaches are always based on a fixed, pre-programmed representation of supposedly universal subjective judgments. Description-based design allows users to express personal subjective judgments about sound textures and reuse these judgments to explore sound spaces in an intuitive and personal way. In the case of audio loops, our approach can benefit from two sets of technologies: 1) the large corpus of studies in the domain of audio *features* which yield efficient representations of audio objects and 2) the emerging technologies of concatenative sound synthesis (Schwartz, 2006) which provide us with the variation generators needed for description-based design.

6. References

Addessi, A.-R. and Pachet, F. (2005) Experiments with a Musical Machine: Musical Style Replication in 3/5 year old Children. British Journal of Music Education, 22(1):21-46, March.

Alvarez, I. Bernard, S. and Deffuant, G. (2007) Keep the Decision Tree and Estimate the Class Probabilities using its Decision Boundary, *Proc. of the 20th IJCAI*, pp. 654-659.

Assayag, G. Rueda, C. Laurson, M. Agon, C. Delerue, O. (1999) Computer Assisted Composition at Ircam: PatchWork & OpenMusic. *Computer Music Journal*, 23 (3):59-72.

Biles, J. (1994) GenJam: A Genetic Algorithm for Generating Jazz Solos, *Proc. of ICMC*, Aarhus, Denmark, ICMA, pp. 131-137.

Burges, Christopher J. C. (1998) "A Tutorial on Support Vector Machines for Pattern Recognition". Data Mining and Knowledge Discovery 2:121–167.

Hamanaka, M. Hirata, K. and Tojo, S. (2008) Melody Morphing Method Based on GTTM, In *Proc. of ISMIR 2008*, Philadelphia, PA, pp. 107-112.

Krumhansl, C. (1990) Cognitive Foundations of Musical Pitch, New York: Oxford University Press.

Lerdahl, F. and Jackendoff, R. (1983) *A Generative Theory of Tonal Music*, The MIT Press, Cambridge.

Letz, S. Orlarey, Y. Fober, D. (1998) The Role of Lambda-Abstraction in Elody. Proc. of the International Computer Music Conference, ICMA, pp. 377-384.

Levenshtein, V.I. (1965) Binary Codes Capable of Correcting Deletions, Insertions, and Reversals. In Cybernetics and Control Theory 10(8): 707-710.

Mandel, M. Poliner, G. Ellis, D. (2006) Support Vector Machine Active Learning for Music Retrieval, Multimedia Systems, special issue on Machine Learning Approaches to Multimedia Information Retrieval, 12(1), Aug., pp. 3-13.

Pachet, F. (2004) Beyond the Cybernetic Jam fantasy: The Continuator. IEEE Computer Graphics and Applications, 4(1):31-35, January/February 2004.

Pachet, F. (2008) The future of content is in ourselves. *ACM Computers in Entertainment*, 6(3).

Rolland, P.-Y. and Pachet, F. (1996) A Framework for Representing Knowledge about Synthesizer Programming, *Computer Music Journal* 20:3, pp. 47-58, Fall.

Sarkar, M., Vercoe, B. and Yang Y. 2007 (2007) Words that describe timbre: a study of auditory perception through language, presented at Language and Music as Cognitive Systems Conference (LMCS-2007), Cambridge, UK, May 11-13.

Schwartz, D. (2006) Concatenative synthesis: The Early Years. Journal of New Music research 35(1), March, pp. 3-22.

Temperley, D. (2007) Music and Probability. MIT Press.

Voss, Richard F. and Clarke, J. (1978) 1/f noise in Music: Music From 1/f Noise. J. Acoust. Soc. Am. 63(1): 258-261.