Jamming With Plunderphonics: Interactive Concatenative Synthesis Of Music

Jean-Julien Aucouturier Francois Pachet

Abstract —
This paper proposes to use the techniques of Concatenative Sound Synthesis in the context of real-time Music Interaction. We describe a system that generates an audio track by concatenating audio segments extracted from pre-existing musical files. The track can be controlled in real-time by specifying high-level properties (or constraints) holding on metadata about the audio segments. A constraint-satisfaction mechanism, based on local search, selects audio segments that best match those constraints at any time. We describe the real-time aspects of the system, notably the asynchronous adding/removing of constraints, and report on several constraints and controllers designed for the system. We illustrate the system with several application examples, notably a virtual drummer able to interact with a human musician in real-time.

Index Terms —**musaicing, concatenative synthesis, interaction, real-time, constraint satisfaction**

I. INTRODUCTION

Most Text-To-Speech (TTS) systems today are able to synthesize text typed in by a user in real-time, through a grapheme-to-phoneme transcription of the sentences to utter ([5]). Such systems typically rely on Concatenative Sound Synthesis (CCS), a paradigm which uses a database of samples, called *units*, and a *unit selection* algorithm that finds the sequence of units that best match a target sound or phrase. TTS systems are completely *user-driven* in the sense that they only produce responses to the user input (text), without any possibility for a predetermined strategy.

Inspired by the success of TTS, CSS is gaining more and more attention in the field of music, as recently reviewed in [22]. While some commercial systems like Synful's RPM Synthesizer ([8]) also build up on the idea of purely user-driven synthesis, a number of experimental systems, like John Oswald's historical Plunderphonics effort ([10]) or more recent semi-automatic systems ([7], [21]), propose a *generative*, compositional approach where the system produces musical textures according to some prescribed target. Along these lines, we introduced in [24] the concept of musical mosaics ("Musaicing"), which reconstructs a given piece of music using sound samples extracted from other pieces. While musically more interesting than user-driven synthesis tools, such generative systems are intrinsically static, and unable to adapt to real-time user input.

This paper describes a real-time interactive music system based on concatenative synthesis, which is an attempt to find a middle point between the purely user-driven and purely generative approaches. We propose a system able to generate an audio track by concatenating audio segments, or *samples*, which can be controlled in real-time by high-level properties holding on their metadata (possibly automatically extracted). The typical sample used in the system is a few beats' audio extract from a given piece of music, which corresponds to a musical bar (or measure), and can therefore be looped while preserving a feeling of steady beat and metric. Figure 1 illustrates such a continuous concatenation of audio samples, which are being selected from a database based on metadata such as energy, presence of certain drum sounds, etc.

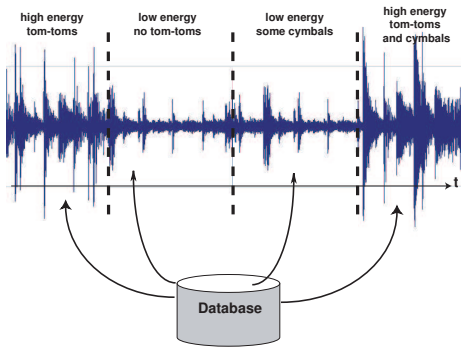

Fig. 1. Illustration of metadata-driven concatenative synthesis: an audio track is generated as a continuous concatenation of audio samples, which are selected in a database according to some criteria holding on their metadata.

We propose a constraint-satisfaction algorithm to control the high-level properties of the generated audio track, such as its energy or its continuity, and a real-time mechanism to allow constraints to be modified at any time. Constraint satisfaction programming (CSP) is a paradigm for solving difficult combinatorial problems, particularly in the finite domain. In this paradigm, problems are represented by variables having a finite set of possible values, and constraints represent properties that the values of variables should have in solutions. CSP is a powerful paradigm because it lets the user state problems declaratively by describing a priori the properties of its solutions and use general-purpose algorithms to find them. There have been numerous applications of CSP to music, e.g. for automatic generation of playlists of music titles [1], automatic harmonization [14] and spatialization [13]. For our "Musaicing" system [24], we introduced the idea of using CSP to generate audio sequences of sound samples, with high-level constraints holding on the metadata of the samples. The work presented in this paper is a real-time, interactive extension of Musaicing.

Figure 2 shows an overview of the principal components of the system. The concatenation engine is composed of a *CSP solver* component which is responsible for the continuous solving of the constraint problem, and a *player* component which is responsible for the rendering of the continuous concatenation of the successive solutions found by the solver. This concatenation engine can be controlled in real-time by a set of *controller* components, which can modify the constraint problem asynchronously (following the arrows labeled "1" in Figure 2), and may react to information received from both the solver ("2") and the player ("3").

The paper is organized as follows. Section II and III describe the concatenation engine. Section II presents the CSP solver which is at the core of the system: an object-oriented implementation of a local search constraint satisfaction technique, called adaptive search [4]. Section III then describes the specific extension of this framework, which we call *incremental CSP*, to handle real-time sequence building and asynchronous CSP modification. We notably examine the careful communication scheme between the solver and the player components. Section IV then presents several possible ways to interact with the concatenation en-

Jean-Julien Aucouturier is assistant researcher in Sony Computer Science Laboratory, Paris, France (Phone: (+33)144080514, Email: jj@csl.sony.fr).

Francois Pachet is researcher in Sony Computer Science Laboratory, Paris, France (Phone: (+33)144080516, Email: pachet@csl.sony.fr)

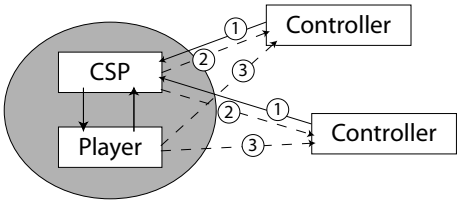

Fig. 2. Overview of the principal components of the system. The concatenation engine is composed of a CSP solver and a player. The CSP can be modified asynchronously (1) by several controllers, which are monitoring both the CSP (2) and the player (3)

gine, and describes a set of controller components that were designed in this purpose, notably controllers capturing information from an incoming MIDI or audio stream e.g. from a human musician interacting with the system. The final section (V) gives a number of usage examples of the system, with an emphasis on a real-time drumming machine able to interact with a human performer. We show that, contrary to more traditional mapping-based systems, the constraint satisfaction approach offers an effective and elegant way to handle the tradeoff between the reactivity and the autonomy of the system, which is a core issue when building interesting interactive systems ([12]).

II. CONSTRAINT-BASED CONCATENATIVE SYNTHESIS

A. Constraint Satisfaction

We define the selection of audio samples to build a concatenated audio sequence as a (finite-domain) constraint-satisfaction problem (CSP). A sequence of samples is modeled as a sequence of M *variables* V_1, V_2, ..., V_M whose *values* can be taken from a finite database of N samples, called their *domain*. Each variable V_i represents the i^{th} sample in the sequence. Figure 3 shows a possible CSP with 4 variables, each with their finite domain, which can be different from one variable to the other.

The problem is to assign values to each variable so that the resulting sequence satisfies a set of constraints defined by the user. Each constraint may hold on a subset of the problem's variables. In the example in Figure 3, constraint C_1 only holds on two variables V_1 and V_3, while C_2 holds on the four variables of the problem. The constraints typically hold on metadata of the assigned samples, which can be either editorial metadata (e.g. the title of the song from which the samples are extracted should be different from one variable to the next), or automatically extracted acoustic metadata (e.g. the energy of each sample in the sequence should be higher than 0.5). The issue of extracting such metadata is not addressed in this paper, however we will give a few examples in Section V.

B. Adaptive Search

As we described in [24], assigning values to variables under an arbitrary set of constraints is a difficult combinatorial problem. The technique we propose here is based on an adaptation of local search techniques to constraint satisfaction, called adaptive search [4]. Adaptive search formulates constraints as simple cost functions, which is well suited for our problem which is clearly over-constrained : it is likely that the constraints cannot all be satisfied at the same time, especially when dealing with numeric acoustic metadata. A constraint forcing samples to have, say, their energy equal to 0.8 will never be exactly satisfied if there is no sample in the domain of the constrained variables with this exact energy value. The cost of a constraint represents "how badly" the constraint is satisfied, for a given assignment of variables. In the previous example, a 0.78 energy sample will be a low-cost candidate for the constraint.

More precisely, we define:

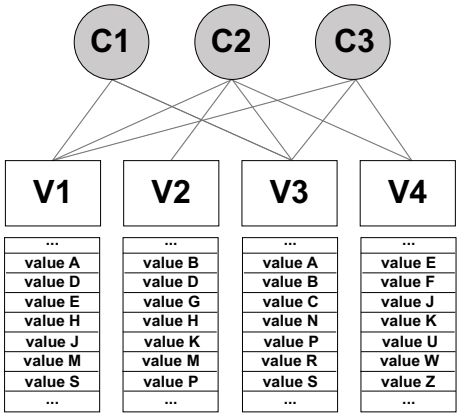

Fig. 3. A CSP with 4 variables and 3 constraints.

- the cost $F(V_i,C)$ of a given variable V_i with value X_i, with respect to a given constraint C, which represents "how badly" X_i satisfies C
- the cost $F(V_i)$ of a given variable V_i with value X_i, which is the weighted sum of its costs $F(V_i,C)$ with respect to each constraint holding on V_i. Each constraint has a weight, which enables to balance the importance of some constraints over some others. Section III.F will illustrate the importance of constraint weighting.
- the global problem cost $F(CSP)$, which is the sum of the $F(V_i)$ for all V_i in the problem.

Assigning a new value to the a variable V_0 modifies the costs $F(V_0,C_i)$ of all the constraints C_i holding on V_0, in turn possibly modifies all the costs $F(V_i)$ of all the variables within the scope of one of several constraints C_i, and finally the global problem cost $F(CSP)$.

The algorithm works as follows:

- Start with a random assignment of values to variable (i.e. a random sequence of samples)
- Compute $F(CSP)$, the total cost of the sequence.
- Repeat until $F(CSP)$ is below a given threshold :
  - For each variable V_i, compute $F(V_i)$
  - Find V_w the worst variable in the sequence, i.e. whose cost is the highest.
  - Find its best possible new value by successively trying all the values in its domain, and selects the value that minimizes the global cost $F(CSP)$ (note that it does not necessarily minimize $F(V_w)$).
  - Assign this value to V_w.

Additionally, there is a provision for handling local minima, through the use of a *tabu* list: worst variables for which no value can be found to decrease the total cost are labeled as "tabu" for a given number of iterations. This trick, along with a technique inspired by simulated annealing, forces the algorithm to explore other regions of the search space.

C. Constraints as Cost functions

The main interest of this algorithm is that constraints are simply seen as cost functions, and hence are very easy to define. For instance, the "all different" constraint stating that all variables should have different values is defined as follows:

```
  AllDifferentCt.cost ()
Return 1 - the number of different values in the problem
divided by the size of the sequence.
```

More complex constraints can be defined as easily. For instance, the "distance" constraint forces each variable V_i in scope to have a value X_i for which a given numerical metadata $p(X_i)$ is as close as possible as a target value p_t (e.g. "all these variables should have an energy of

0.1"). The corresponding cost function is defined as follows :

`DistanceCt.cost()`
Return the mean distance between the $p(X_i)$ and p_t, i.e. $\frac{1}{M}\sum_{i=0}^{M-1}|p(X_i)-p_t|^2$

The "continuity" constraint holds on a set of variable $V_i, i = 1..s$. It forces each duple of successive variables $\{V_i, V_{i+1}\}$ to have values $\{X_i, X_{i+1}\}$ for which a given numerical metadata p has similar values $\{p(X_i), p(X_{i+1})\}$. The corresponding cost function can be defined as follows :

`ContinuityCt.cost()`
Return the mean distance between all $p(X_i)$ and $p(X_{i+1})$, i.e. $\frac{1}{M}\sum_{i=0}^{M-2}|p(X_i)-p(X_{i+1})|^2$

In practice, the cost functions are implemented more efficiently, by passing the lastly modified variable as argument to the cost functions. This information is used to compute only the differential cost, instead of the whole cost.

For instance, the cost function of the distance constraint can be defined in such a differential way :

`DistanceCt.cost(Variable v)`
Returns $\frac{1}{M}(oldcost*(M-1)+|p(X_n)-p_t|^2)$

This saves $M-1$ database accesses to compute the $p(X_i)$, and $M-1$ substractions and multiplications. Such optimizations are crucially important, as the cost function of each constraint is called N times at each iteration, where N is the size of the current variable's domain.

D. A note on continuous metadata

Note that the formulation of constraints as cost function is a natural way to handle continuous metadata such as the energy of a piece of audio, without any need for quantization. Each constraint computes a floating-point "cost" which describes how badly they are satisfied for a given instanciation of the problem's variables. A distance constraint such that a variable should have an energy of 0.1 will be perfectly realised for an audio segment with energy 0.1 (cost = 0), slightly less for a segment with energy 0.12 (cost $|0.1-0.12|^2 = 0.0004$), and even less for energy 0.125 (cost 0.0006). Cost functions are optionally rescaled between 0 and 100 using minimum and maximum bounds on metadata values, so that constraints can be compared and weighted (see Section V.C.5).

E. Object-Oriented Implementation

The previous algorithm is implemented in a Java framework for constraint-satisfaction, which reifies such objects as variable, constraint, problem and solver. The implementation capitalizes on several object-oriented constraint satisfaction frameworks previously built in our research team, such as BackTalk and BackJava ([19], [18].

E.1 Variable

A variable is represented by a `Variable` object, which contains a `domain` (a `Collection` of values, generically Java Objects) and a `value` (a Java object). The `value` of the variable is chosen among the values in its domain, by the solver. Typical objects found in a variable's `domain` encapsulate multimedia items in a database, which can query the database to access their metadata values. In [11], we have described a Java framework, MCM, which offers such functionalities, notably a sophisticated query language where metadata can be described by a path from the object (i.e. an audio sample), e.g. `sample.song.artist.name` designates the metadata corresponding to the name of the artist of the song from which is extracted a given audio sample.

E.2 Constraint

The `constraint` object contains a list of `variables` on which it holds, a `weight` (a `double` precision floating-point number) and implements a cost function able to compute the cost of the constrained variables based on their current values. To decouple these abstractions from their implementation so both can vary independently and be re-used as building blocks in several constraints, we apply the *Bridge* design pattern [6], and reify two adapters :

- `CostAdapter`: implements the `computeCost(Constraint c)` method. Several adapters were implemented, such as `AllDifferent`, `Cardinality`, `Distance`, or `Continuity`.
- `VariableAdapter`: implements the method `selectVariables(Constraint c)`, which returns the set of constrained variables for a given constraint at a given iteration. This set of variables need not be static. A constraint may hold on the variables which match a given criteria at a given iteration, e.g. "All piano samples should be different" uses the `AllDifferent` cost adapter and a dynamic variable adapter selecting the samples which match the property "sample.timbre = piano". Moreover, a constraint's `VariableAdapter` can use the output of another constraint's `CostAdapter`, i.e. at any iteration, constraint C_2 holds on all the variables that have a bad cost according to constraint C_1.

E.3 Problem

CSP problems themselves are represented by instances of class `Problem`. The class defines two instance fields: `variables`, holding the set of variables and `constraints`, the set of constraints. The main method of the `Problem` class is the `solve(Solver solver)` method, which uses an instance of the `Solver` class described below. The `Problem` class implements a number of advanced features, such as a scheduler that carefully chooses the order in which constraints are treated at each iteration, based on their possible hierarchical dependencies described above. Reifying CSP problems offers a number of design advantages already described in [19]. Additionally, this provides a unified interface for modifying the problem's structure (variables, variables domains, constraints, constraint's variables, constraint's weights), as we will become clear in Section III.E.

E.4 Solver

Finally, a number of variants around the adaptive search algorithm presented above are implemented as subclasses of the `Solver` class, again an application of the *Bridge* design pattern. The main method in the `Solver` class is the `solve(Problem p)` method, which is called by `Problem.solve(Solver s)`. Other types of CSP algorithms, like filtering based on arc-consistency or back-tracking methods (see [19] for a review) could be implemented in this framework. However, we have shown on the example of constraint-based playlist generation ([1], [15]) that local-search approaches such as adaptive search generally allow a scale-up of the domain sizes (up to several tens of thousand values) unreachable by traditional complete CSP algorithms.

III. REAL-TIME CSP

This section examines the extension of the previous framework to handle the continuous building of a sequence of audio samples. We introduce the notion of *Incremental* CSP, and describe the communication between the CSP and a player thread, which is responsible for rendering the concatenation of the successive best solutions found by the CSP solver.

A. Incremental CSP

We represent the audio track being generated by a growing sequence of variables, with a finite memory M. At any time, the next sample to be selected as well as the M latest past samples (M arbitrary, can be as large as n the total

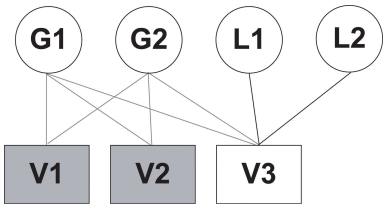

Fig. 4. An incremental CSP with 3 variables and 4 constraints. V_3 is the current variable, and V_1, V_2 are the past variables.

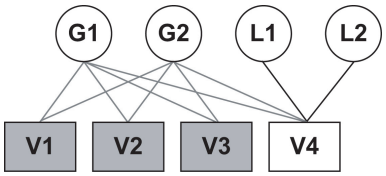

Fig. 5. The same incremental CSP than in Figure 4, after the increment operation.

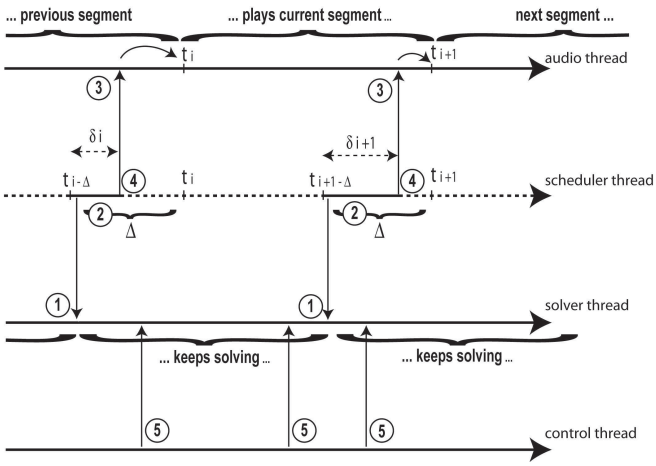

Fig. 6. The real-time implementation of the system, using 4 concurrent threads. See main text for explanation of steps 1 to 5.

number of samples played so far) constitute a sequence of *variables* V_{n-M}, V_{n-M+1}, ..., V_{n-1}, V_n, each with their domain. We call the variable V_n corresponding to the next sample to be selected the *current variable*, and the V_{n-i} for $i = 1..M$ the *past variables*.

The problem is to successively assign values to each variable so that the resulting sequence satisfies a set of constraints defined by the user. The constraints may change at any time, in an asynchronous manner. Obviously, at any time, the problem can only set the value of the current variable : once a variable is played, its value cannot be changed ("one can't modify the past"). However, the choice of the next sample is influenced by the past choices, as constraints holding on the current variable may also hold on the past variables.

To model the passing of time, we introduce the notion of *increment* operation. Each time a value is assigned to the current variable, the problem is incremented, i.e. a new variable is added to the problem. The former current variable becomes a past variable, and the new variable represents the next current variable.

Figures 4 and 5 explicates the structure of the problem, and illustrate the increment operation. In Figure 4, the CSP at a given iteration i includes $M = 3$ variables, one current and 2 past, with some constraints (the G and L circles) holding on them. A value is selected for V_3, and the corresponding audio is scheduled to be played. At the next iteration $i+1$, the CSP is incremented, i.e. a new variable V_4 is added, which becomes the new current variable. Note that the scope of the constraints is automatically modified to also hold on the newly added variable. We will make explicit two strategies for such a mechanism in Subsection C below.

B. Incremental Adaptive Search

In this context, since only one variable can be modified at a time (the current variable), there is no combinatorial explosion of the search space. A complete enumeration of all possible values for the current variable is only the size of the sample database. Adaptive search is mainly targeted at off-line problems where all values must be assigned simultaneously, and a complete N^M enumeration is intractable, e.g. in playlist generation [1]. However, adaptive search's formulation of constraints as simple cost functions is still well suited for our problem, notably to handle contradictory numeric constraints, as will be seen below. The algorithm described in Section II is applied as is, with the simple modification that at each increment operation, the past variables are labeled as "tabu", so that their value can't be modified. Note that it would also be possible to consider a set of several current variables, in which case the full power of adaptive search would be used. For instance, the current variable corresponding to the next measure to be played could be further divided into -say- 4 variables corresponding to a beat each.

C. Local and Global Constraints

In the context of incremental CSP, and for the clarity of further discussions, we distinguish 2 types of constraints :

- Local Constraints only hold on the current variable. They influence the selection of the next sample by only looking at its intrinsic properties, without taking past values into account. Typically, a distance constraint as defined in Section II.C is a local constraint.
- Global Constraints hold on the current variable plus some or all of the past variables. They influence the selection of the best drum sample by also accounting for the values of the past variables. Typically, a continuity constraint as defined in Section II.C is a global constraint, trying to select new values so that they are continuous with the past, already selected values.

Upon increment of the CSP, local and global constraints have a different behavior. All local constraints update their scope by removing the previous current variable (now v_{n-1}), and adding the new current variable v_n. All global constraints simply add the newly added variable to their scope. This mechanism is illustrated in Figures 4 and 5: before the increment, the global constraints G_1 and G_2 hold on $\{V_1, V_2, V_3\}$ and the local constraints L_1 and L_2 hold only on V_3, which is the current variable. After increment, G_1 and G_2 modify their scope to also include the new current variable V_4, while L_1 and L_2 now only hold on V_4.

D. Real-time Implementation

The audio track built by the system corresponds to the concatenation of the values of the successive current variables. This requires a careful synchronization between the Incremental CSP component and an audio player, which is in charge of the continuous playback of the track. The implementation of this scheme uses a *scheduler* thread, which iteratively queries the solver for the best solution so far and schedules the corresponding audio for playback by the audio thread.

Figure 6 explicits the interactions between the 3 threads corresponding to the CSP solver, the audio thread and the scheduler. At any time, the audio corresponding to the latest selected sample is playing, and a new value must be scheduled to immediately start after it finishes at endtime t_i.

1. **At time $t_i - \Delta$, the scheduler wakes up, and asks the solver for the best value it has found so far for the current variable, given all the current constraints and the values of the past variables. The solver replies and increments immediately to start looking for the next sample.**
2. **The scheduler retrieves the audio corresponding to the sample found at step1. In the current implementation, this includes reading and decoding a .mp3 file between a start and end date through a local network, which may take a variable time δ_i.**
3. **At time $t_i - \Delta + \delta_i$, the scheduler thread schedules the decoded audio for the current sample for playback at the exact ending time t_i of the latest sample, which is currently being played. The choice of Δ is made a priori, to ensure that $t_i - \Delta + \delta_i < t_i$, i.e. $\Delta > \delta_i, \forall i$. In our current implementation, we chose $\Delta = 500ms$.**
4. **The scheduler sleeps until $t_{i+1} - \Delta$, having $t_{i+1} = t_i + d_i$, where d_i is the duration of the audio just scheduled. This mainly gives the priority back to the solver, which keeps scouring the database for the next value to be scheduled at t_{i+1}.**

E. Asynchronous CSP problem alteration

As can be seen in Figure 6 with the arrows labeled "5", one or several control threads may modify the CSP problem, at any time. This in turn modifies the values found by the solver, which enables the real-time high-level control of the output of the concatenation engine. Such changes can be done manually by a user via a GUI, be scheduled *a priori*, or be the result of the analysis of an interaction, as proposed below in Section IV. We describe here the mechanisms that allow such asynchronous changes to the CSP problem.

Several types of changes can be done on the CSP Problem, including:

- **Adding a variable: This is the case in the *increment* operation, which, while not requested by the control thread, may nevertheless occur at any time, since it depends on the timing of the player thread, and the duration of the audio samples.**
- **Changing the domain of the current variable: For instance, the sequence should now be composed of samples from "The Beatles - Yesterday", and not any longer from "Let it be" as was the case until now. The change is made on the current variable, and all future current variables created by the next increment operations will use the same domain.**
- **Adding/Removing a constraint: The constraint may be either a local constraint (e.g. the next drum sample should be a high energy sample, and not low energy as was the case until now) or a global constraint (e.g. from now on, the samples should have a continuous energy).**
- **Changing the variables on which a constraint hold: This is also a notable consequence of the *increment* operation, where global constraints are modified to include the new current variable, and local constraints to only hold on the new current variable.**
- **Changing the weight of a constraint: It will be shown in Section V that the ratio between the weights of local and global constraints influences the whole system's behaviour, which we may wish to modulate in real-time.**

Any of these changes, if done in an unsynchronized manner during CSP solving, has a potential for yielding unexpected behaviours. For instance, a typical solver implementation will process the problem's constraints sequentially, hence adding or removing a constraint during this process will create havoc. Also, we have described that an effective way to bookkeep the costs of the variables when very many values are tried successively is to update them incrementally. However, if the weight of a given constraint changes from one value evaluation to the next, the whole chain of differential costs will break down. In full generality, it is dangerous to alter the structure of the CSP problem during each iteration of the solving phase.

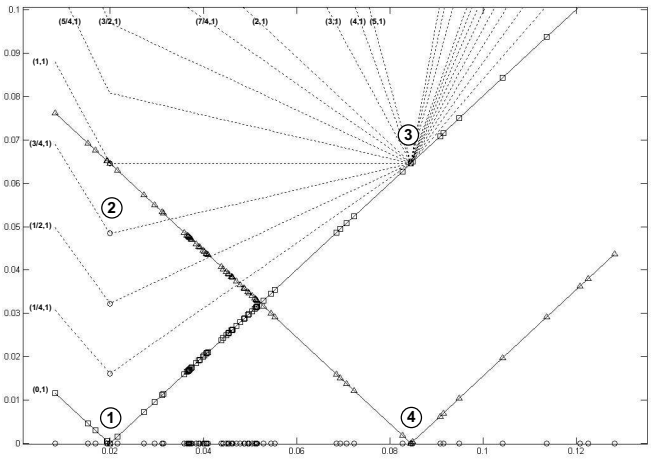

Fig. 7. Cumulated costs of two contradictory linear cost functions, for different combinations of their respective weights w_g and w_l. The solid curve with square markers corresponds to the cost function of the global continuity constraint (i.e. $(w_l, w_g) = (0, 1)$). The solid curve with upward pointing triangles correspond to the cost function of the local distance constraint (i.e. $(w_l, w_g) = (1, 0)$). The dotted curves correspond to intermediate weight ratios. Circle markers on the dotted curves represent local cost minima on the $[x_p, x_t]$ interval. See main text for explanation of labels 1 to 4.

Therefore, synchronization is enforced by logging any incoming change request that occurs during a given iteration into a FIFO stack (the *to-do list*), which is then committed (i.e. each request is processed in the order of arrival) at the beginning of the next iteration. If no control event is sent to the CSP problem while searching for the next sample to play, each iteration will yield the same best value result (and thus indeed only one iteration is needed after each increment). However, if a series of control events are sent to the CSP during a given iteration i, these changes are first stored in wait for the current iteration to finish, and then committed before the next iteration $i+1$. In this case, it is likely that the best values found by iterations i and $i+1$ will be different, since they simply don't correspond to the same problem. The more control events are sent to the CSP, the more iterations are needed to find the best solution between each increment operation.

F. Case of contradictory constraints

We give here an illustrative example of an incremental CSP with 2 typical constraints:

- **a global *continuity* constraint on a numeric metadata x, stating that successive samples should have x values that are as close as possible**
- **a local *distance* constraint that states that the next sample's x value should be as close as possible to a given target value x_t**

Additionally, we assume that the latest past variable had a x_p value, and that there are weights w_g and w_l on the global and local constraint respectively.

Clearly, these 2 constraints are contradictory: if x_p is low and x_t is high, high x values will have a low cost according to the local distance constraint (being close to x_t), but a high cost according to the global continuity constraint (being far from x_p). One can manipulate the total cost of a given sample x_i by changing the weights on the local and global constraints:

$$cost(x_i) = w_l.cost_{local}(x_i) + w_g.cost_{global}(x_i) \quad (1)$$

Figure 7 shows the total cost profile for all values between x_p and x_t in the case where both costs are implemented as

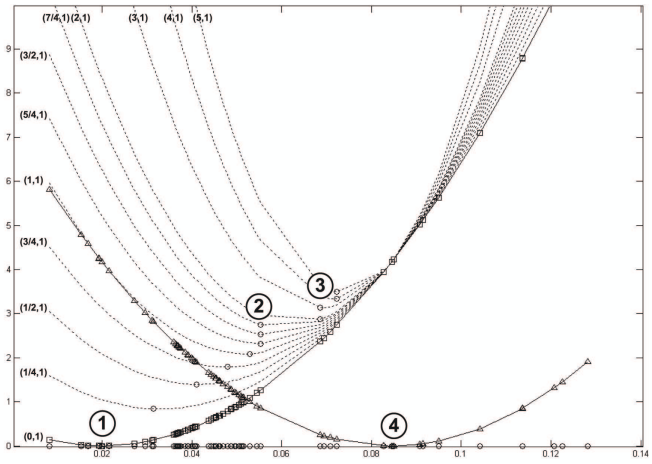

Fig. 8. Cumulated costs of two contradictory parabolic cost functions, for different combinations of their respective weights w_g and w_l, using the same conventions than in Figure 7. See main text for explanations.

linear functions, i.e.

$$cost_{local}(x) = \alpha.|x - x_t| \tag{2}$$

and

$$cost_{global}(x) = \beta.|x - x_p| \tag{3}$$

In Figure 7, the solid curve with square markers correspond to the cost of the global continuity constraint (using $\beta = 1$), which is zero at $x = x_p$ (label 1). The solid curve with upward-pointing triangles correspond to the cost of the local distance constraint (using $\alpha = 1$), which is zero at $x = x_t$ (label 4). The dotted curves correspond to the cumulated cost in equation 1, for different ratios between w_l and w_g. We observe that the cumulated cost is also linear on $[x_p, x_t]$, with a slope depending on the weight ratio, and therefore the behaviour is winner-take-all. When w_g is higher than w_l, the best cost value for x is x_p (label 2), while if $w_l > w_g$, the best value for x is x_t (label3). Intermediary values are never reached.

In many cases, we may wish a middle-point behaviour in which the sequence will gradually increase from x_p to x_t, with a rate depending on the weight ratio. While this cannot be achieved using linear costs, Figure 8 shows that it can with parabolic cost functions, such as

$$cost_{local}(x) = |x - x_t|^2 \tag{4}$$

and

$$cost_{global}(x) = |x - x_p|^2 \tag{5}$$

We observe that the cumulated costs (in dotted line) have local minima (marked as a circle on the dotted curve) on the interval $[x_p, x_t]$, which are all the more so close to x_t as the ratio $\frac{w_l}{w_g}$ increases. For a small weight ratio, the best x_1 value will be close from x_p. At the next increment step, the next selected value x_2 will be slightly higher than the new $x_p = x_1$, and so on until x reaches the target value x_t. If the ratio of the weights is higher, the speed of convergence to x_t will be higher. Labels 2 and 3 illustrate that since the domain is made of discrete values, the speed of convergence from x_p to x_t may not be uniform, but may reach a temporary potential barrier, where several successive exact cost minima are approximated by the same domain value.

IV. CONTROLLERS AND INTERACTION

In this section, we present several possible ways to interact with the concatenation engine, and describes a set of controller components that were designed in this purpose. A `controller` component is an abstract thread with a `start()` and a `stop` method. As represented in Figure 2, a `controller` is a listener of both the CSP problem's state (e.g. have the constraints changed, what is the cost of the best solution so far, etc.) and the player's state (e.g. what is the current sample being played).

A. One-shot Controller

The simplest type of controller is the one that immediately puts a new constraint on the problem when started, and removes it when stopped. A possible example of such a controller is the `Consecutivity` controller that enforces that successive samples be extracted from the same song, and that the original end position of a given sample be as close as possible to the original start position of the next sample. A concatenation engine controlled with this only controller will play samples song by song in their original order (starting at a random position).

B. Scheduled Controller

Another useful type of controller is the `ScheduledController`, with which CSP problem modifications can be scheduled to occur at given dates. For instance, the constraints put on the CSP problem by the above mentioned `Consecutivity` controller could be scheduled to be sent to the solver after 3 minutes of interaction. A more typical example of `ScheduledController` is the `WeightRamp` controller which puts a new constraint on the CSP problem at a given start date, and increases (or decreases) its weight linearly until a given end date. This could be used e.g. to force the selected samples to increasingly belong to the end of a given song, using a `Distance` constraint on the start position of the samples. When the constraint is first positioned, its low cost will only slightly bias the CSP solver towards selecting ending samples. As its weight increases however, the selected samples will converge to the end of the song, while still trying to respect the other constraints put on the CSP problem (e.g. it will converge to the end of the song by only using "piano" samples that are on the way).

C. Detector Controller

A `Detector` controller will issue a request for problem modification (typically add a new constraint) when a given property is matched either on the state of the CSP problem, or on the state of the Player. One example of this is the `Stopper` controller, which stops the concatenation engine when its detects that the last sample of a song has just been played.

D. Capture

A controller may analyse and react to information from an incoming stream of data, e.g. a MIDI or audio realtime acquisition from a human player. Figure 9 illustrates a possible scenario where a human musician plays music on a MIDI keyboard. The MIDI acquisition thread forwards any meaningful events (typically `note-on` and `note-off` events) to the controller thread (1). Such events can occur at a rapid rate and may not be meaningful individually, e.g. a chord will trigger several quasi simultaneous events, which need to be aggregated by post-processing on their onset date. Therefore, some buffering will typically happen in the controller (2). At given times, either after sufficient data has been received from the MIDI capture, or at a given fixed rate, the controller will analyse the buffer of data, and send a modification request on the CSP problem (3).

Several types of analysis can be done on MIDI data, notably

- energy : the mean velocity of the `note-on` MIDI messages, computed over 500 ms windows.
- onset density : the number of `note-on` MIDI messages received over 500 ms windows.
- pitch : the mean MIDI pitch of the `note-on` MIDI messages, computed over 500 ms windows.

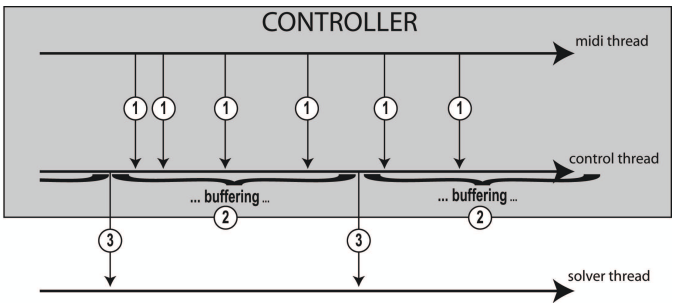

Fig. 9. Illustration of a controller analysing a MIDI capture

A wealth of other MIDI descriptors potentially useful for music interaction systems can be found e.g. in [17]. Similarly, audio capture can be analysed in real-time, with the added advantage the metadata extracted from the samples used by the CSP can usually be extracted with the same algorithms from the incoming audio buffers.

Several mappings of the analysed capture metadata to the samples' metadata can be used, among which:

- Direct control: the incoming metadata values from the captured are directly matched to the sample metadata using distance constraints. For instance, each time a new energy value is captured from the MIDI data, a new constraint is put on the energy of the samples, and any previous constraint holding on their energy is removed.
- Event-driven control: like a `Detector` controller, the `Capture` controller can wait until a given event occurs on the captured metadata, e.g. the energy grows higher than 0.5, to trigger a CSP problem change.

Arbitrary converters can be plugged in the `Capture` controller objects to e.g. adapt the range of the input to the output, or change the default linear mapping to any control curve. This can be used to build an `Inverter` controller which forces the selected samples to be of low energy when high energy is captured, and reciprocally.

In the next section, we notably describe an automatic drumming machine that uses a MIDI capture controller to detect parameters from a MIDI keyboard performance, and match them to specifically designed drum sounds parameters.

V. EXAMPLES

This section gives a few usage examples of the system, notably a real-time drumming machine able to interact with a human performer.

A. Rebuilding a normal player

An illustrative example of the system, if not particularly useful, is to recreate a simple song player, using constraints on the song's samples. Three controllers are needed for the player to have the following properties:

- The song starts at the beginning: This can be enforced by a `WeightRamp` controller putting a (local) distance constraint on the samples so that their original start position be close to 0.0. The constraint starts with weight 10, and quickly decreases to 0 in the first few seconds.
- The song is played in the original order: This is realized by a simple `OneShot` controller that puts in a (global) Consecutivity constraint on the sequence when it starts. This enforces that successively selected samples be from the same song, and that they correspond to successive samples in the original song.
- The song stops at the end: This is realized with a `Detector` controller monitoring the player component of the concatenation engine, which stops the concatenation engine when its detects that the last sample of a song has just been played.

It is possible to choose the song to be played by setting the variables' domain to only contain samples from the song. Additionally, it is possible to "speed up" the playback of the song by adding an additional `WeightRamp` controller that forces the samples to increasingly belong to the end of the song (using a distance constraint on the original start position of the samples, and an increasing weight from 0 to 10). The controller can be scheduled to start at a given start date, say 1 minute from the start. The song will play normally for the first minute, and then successively skip to samples that are increasingly near the end.

B. Concurrent playback strategies

The previous strategy can be combined with a concurrent strategy, e.g. ordering the successive samples by increasing energy. By manipulating the weight ratio between both (sets of) constraints (w_t the weight of the time-order constraints, and w_e the weight of the energy-order constraints), different playback strategies can be combined. If $w_t >> w_e$, the song will playback normally, from begin to end. If $w_e >> w_t$, the song will be played in order of increasing energy. For intermediate weight ratios, the song can be made e.g. to skip to the next high energy section, then proceed in normal time-order to a low energy section, and then rewind to a previous section of similar energy, etc. We are currently experimenting with such multi-criteria browsing into songs, notably as a means to explore different instrumental sections using distance constraints on timbre descriptors.

C. Interactive Drumming Machine

We now describe an advanced example of interactive concatenative synthesis: an automatic drumming machine able to interact with a human musician playing on a MIDI instrument. The system, nicknamed "RingoMatic"[1], uses drum samples automatically extracted from existing music titles to build an interactive drum track, using specifically designed drum sound descriptors. A complete description of the system, notably metadata extraction details can be found in [2]. Figure 10 shows a screenshot of the system, which is integrated as a plugin of the MusicBrowser, our content-based EMD system [11]. We summarize here the different steps.

C.1 Drum Solo Detection

In order to extract drum samples from an arbitrary song, we first need the ability to detect drum solo parts, i.e. sections in music where only a drumkit is playing. This is typically a drum solo in the middle of a jazz piece or shorter drum breaks in a rock or funk song. We model the problem as a 2 class classification problem, and build a labeled database of 100 5-second music extracts, the first 50 being pure drum solo, and the other 50 various extracts of popular music, encompassing many different genres (jazz, rock, heavy metal, classical, folk, electronic), with or without drums. We use EDS ([25]), a genetic-programming scheme for extracting arbitrary high-level audio descriptors from audio signals, to produce a classifier (see [2] for more details). The best feature found by EDS is a correlate of the SpectralKurtosis MP7 descriptor:

```
min (spectralKurtosis (hann (split (bartlett (bpFilter
(triangle (square (normalize(x))),223,2456)),134))))
```

The best classifier found by EDS is a k-nn classifier using 2 inverse-distance weighted nearest neighbors, which achieves 0.99 precision on the training database. We apply the drum detector on sliding 3-second windows on full songs to segment drum solo parts. For robustness, we only look for segments corresponding to at least 3 successive windows classified as drums.

[1] "an automatic Ringo Starr"

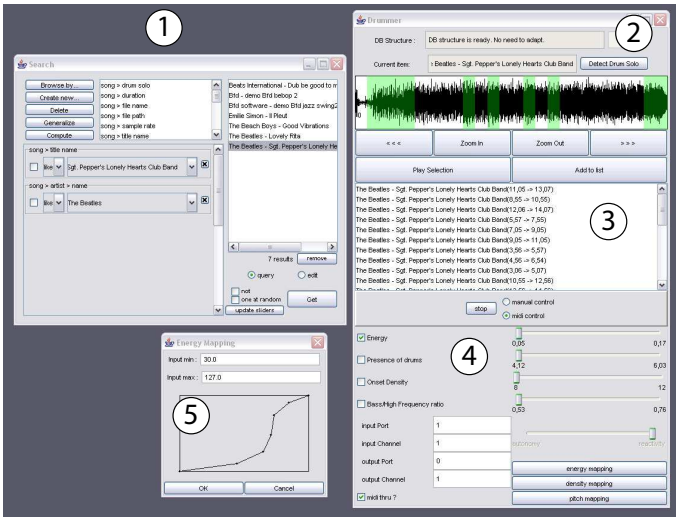

Fig. 10. Screenshot of the RingoMatic system. An arbitrary song is selected using queries on its metadata (1). All sections of drum solo are automatically detected (2), segmented (3) and indexed with automatically extracted drum sound descriptors such as energy or presence of cymbals. These samples are then used as variable domains in an interactive concatenation engine, which is controlled with metadata extracted from a live MIDI performance(4). The mappings between the input MIDI metadata and the metadata of the drum samples can be modified in real-time (5).

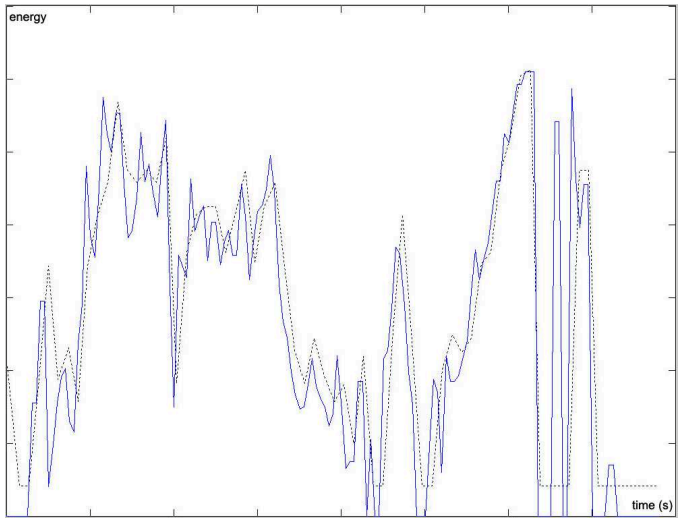

Fig. 11. Energy of the MIDI performance (solid line) and energy of the drum samples selected by the solver (dashed line) over time.

C.2 Segmentation

Once large sections of music which only includes drum solo are identified, we segment them in 4-beat drum samples using a stripped-down version of the method described in [20]. Since beat tracking on drumtracks is usually a lot easier than on arbitrarily complex polyphonic audio, we only consider one frequency band [0-400 Hz]. A first pass is done to compute the bpm on 3-second buffers, and a second pass is done with a beat-tracker tuned on the most represented tempo found during the first pass in order to localize the beats. Then, a 4-beat-long drum sample is extracted every beat (under the assumption that the music signature is 4/4).

C.3 Metadata Extraction

Once a database of drum samples has been gathered, we index each sample with perceptually-meaningful metadata. Again, we use the EDS system to find good specific signal processing features, and to optimize machine learning algorithms that use these features. We describe here 4 descriptors relevant for drum samples that we modeled with EDS. For each, we give the best feature found by EDS, and the classification results using 10-fold cross validation. Again, see [2] for more details.

- Energy : the perceptive energy of the drum sample, independent of the RMS volume (all drum samples are RMS-normalized). The best feature found by EDS

$$mean(log(var(split(deriv(square(X)), 1s)))) \quad (6)$$

  is related not with the absolute energy of the signal, but rather with the amplitude of its variations. This yields a precision of 0.89.
- Onset Density : the sensation of stroke density in the drum sample. Drum rolls typically include very many strokes, while some fills may include just a few kicks and crashes. The best feature found by EDS

$$length(peaks(rms(hamming(split(X, 4096))))) \quad (7)$$

  can be interpreted as a rough count of peaks of energy. The precision of the associated knn classifier is 0.92
- Presence of drums : the importance of tom and bass drum strokes as opposed to cymbals and snare drums. Jazz drummers typically use toms to give a ethnic groove to a song, rather than cymbals and ride which are typically used for swing. The best feature found by EDS

$$SpectralDecrease(deriv(square(norm(X)))) \quad (8)$$

  gives a classification precision of 0.84
- Presence of cymbals : the importance of high-frequency sounds like cymbals and ride. The best feature found with EDS

$$division(rms(lpfilter(X, 500, 44100)), rms(X)) \quad (9)$$

  is simply the ratio of high frequency energy over the total energy of the signal. This achieves 0.82 precision.

C.4 MIDI interaction

The resulting database of drum samples is used as the domain in a CSP problem, in order to generate a continuous drumtrack. We then control the high-level properties of the drumtrack using constraints holding on the metadata of the successive drum samples. We use a `Capture` controller as described in Section IV to analyse the MIDI performance of the human player in real-time and extract its energy, onset density and pitch.

The three streams of MIDI metadata are converted using a transfer function, and the controller sends new local distance constraints to the CSP problem accordingly. The concatenation engine thus generates a drumtrack by satisfying constraints created by analysing the MIDI performance. For instance, a new MIDI energy value modifies a local distance constraint holding on the drum samples' energy metadata, i.e. which forces the energy of the newly selected samples to be as close as possible to the input MIDI energy. Similarly, low MIDI pitch can be inverse converted, and mapped to a local constraint holding on the presence of cymbal metadata, so that melodies played on the lower octaves of the MIDI instrument trigger drum track that use a lot of high pitched sounds, and conversely, high pitched melodies trigger a lot of bass drum and tom sounds. Mappings between MIDI performance data and audio drumtrack metadata/constraints can be arbitrary complex and can be modified in real-time. This issue of mapping between MIDI performance and Machine generated music parameters is discussed at length in [16].

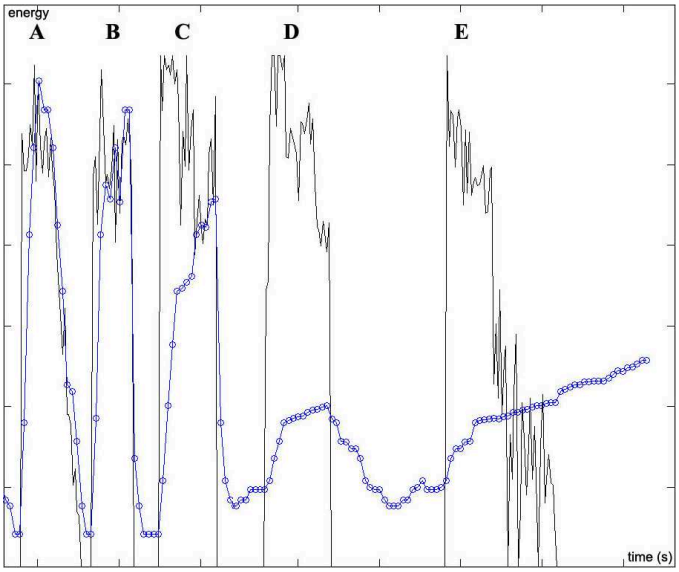

Fig. 12. System behaviour with different ratio r of local to global weight. (A) $r = \infty$: the drumming machine reacts immediately. (B) $r = 2$. (C) $r = 1$. (D) $r = \frac{1}{2}$. (E) $r = 0$: complete autonomy of the system.

Figure 11 shows both the energy of the MIDI performance and the energy of the drumtrack generated by the system over time. We observe that the CSP solver is able to follow the energy of the performance very finely, with very small latency (typically the duration of one drum sample), and a precision which depends on the available energy values in the database.

C.5 Autonomy/Reactivity Trade-off

While technically satisfying, such fully reactive behaviour is often not suitable in a music interaction context, in which one wants the interacting agent to have both musical realism and autonomy[2]. For instance, it may be unrealistic to instantly switch from very low to very high energy. Similarly to the example given in Section III.F, we use global continuity constraints to counter-balance the immediate reactivity created by the local distance constraints. If the performance energy suddenly increases, highly energetic drum samples will have a low cost according to the local distance constraint, but a high cost according to the global continuity constraint. One can manipulate the total cost of a drum sample by putting weights w_l and w_g on the local and global constraints respectively. A variety of behaviours can be achieved ranging between complete reactivity ($w_g = 0$) and complete autonomy ($w_l = 0$). Figure 12 shows the behaviour of the drummer subjected to a typical MIDI energy input, using different ratio between w_l and w_g. For $w_g = 0$, the behaviour is then same than in Figure 11. For $w_l = 0$, the system does not take any account of the MIDI interaction, and only obeys to the continuity constraint. With intermediate settings, the drumming machine follows the input energy while still preserving continuity, thus yielding a more musical output.

D. Comparison with mapping-based systems

Most of the algorithmic elements provided by individual controllers and constraints (e.g. adapt the energy of the selected segments to the energy of the incoming performance) can be realized by standard direct mapping without utilizing the CSP formulation (as seen e.g. in [16]). However, the architecture proposed in this paper provides a number of advantages over this traditional approach.

- **Common framework for local mappings and global consistency: Typical interactive systems use a mapping formulation for coding reactive behaviours, and some hard-coded, unmodular rules for enforcing long-term consistency. The CSP formulation provides a common framework for specifying both types of behaviour, as local and global constraints. This ensures a greater modularity and clearer formulation of the properties of each system.**
- **Local and global weights: The fact that both local mappings and global behaviours be both represented by identical objects enables their balancing with the weight mecanism described in Section C.5.**
- **Expressive power: While "mapping" constraints (i.e. distance constraints as defined in Section II.C) clearly are important and natural elements of interactive systems, constraints can represent a variety of global properties which cannot easily be represented by mappings between input and output, like cardinality or distribution constraints (e.g. 2 piano segments should not occur in a row). Their representation as simple cost functions allows constraints to implement arbitrary complicated behaviours, such as database queries or histogram matching.**
- **Change in real-time: The modularity of both local and global constraints enables to switch them on and off in real-time, and to change their scope or parameters. In mapping-base systems, this is also possible for reactive behaviours ("rewiring"), however rarely possible for global behaviours. Such real-time modification of free-code interactive behaviours would otherwise require advanced concurrent programming architectures, as recently proposed in [23].**

VI. Conclusion

We have presented a real-time concatenative system, which uses a constraint-satisfaction algorithm to control the high-level properties of the generated audio track, such as its energy or its continuity. We described several extensions of traditional fixed-length sequence CSP to handle incremental sequences of variables, and presented a mechanism to change the constraints in real-time. Such a concatenation engine can be used to build real-time interactive systems, where constraints are controlled by the analysis of an incoming MIDI or audio stream from a human musician. The example of the "Ringomatic" system demonstrates that competitive local/global constraint satisfaction is both an effective and elegant way to control the autonomy/reactivity of the system, which is a core issue when building interesting interactive systems.

VII. Acknowledgements

This work gratefully inherits ideas, bits and pieces about adaptive search from Philippe Codognet, musaicing from Aymeric Zils & Anthony Beurivé and Object-Oriented CSP from Pierre Roy. It uses JSyn, a Java synthesis library [3], and MIDIShare, a real-time multi-task MIDI operating system developed by GRAME [9]. It has been partially funded by the SemanticHifi European IST project.

References

[1] **J.-J. Aucouturier and F. Pachet. Scaling up music playlist generation systems. In *Proceedings of The IEEE International Conference on Multimedia and Expo, Lausanne (Switzerland)*, August 2002.**

[2] **J.-J. Aucouturier and F. Pachet. Ringomatic: A real-time interactive drummer using constraint-satisfaction and drum sound descriptors. In *Proceedings of the International Conference on Music Information Retrieval*. London, September 2005.**

[2] **we define here "autonomy" not as an intrinsic self-motivation of the system - which lacks any kind of emerging behaviour -, but as its ability to preserve a predefined set of global constraints.**

[3] P. Burk. Jsyn, a real-time synthesis api for java. In *Proceedings of the International Computer Music Conference (ICMC)*. ICMA, 1998. available: http://www.softsynth.com/jsyn/.
[4] P. Codognet and D. Diaz. Yet another local search method for constraint solving. In *Proceedings of the AAAI Symposium, Cape Cod, MA*, November 2001.
[5] R. I. E. Damper. *Data-Driven Techniques in Speech Synthesis*. Springer Series on Telecommunications Technology & Applications, 2001.
[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series, 1st edition 1995.
[7] A. Lazier and P. Cook. Mosievius: Feature-driven interactive audio mosaicing. In *Proceedings of the COST-G6 Conference on Digital Audio*, London, UK, September 2003.
[8] E. Lindemann. Synful reconstructive phrase modelling (http://www.synful.com).
[9] Y. Orlarey and H. Lequay. Midishare: a real time multitasks software module for midi applications. In *Proceedings of the 1989 International Computer Music Conference, San Francisco*. Computer Music Association, 1989.
[10] J. Oswald. In J. Zorn, editor, *Arcana: Musicians on music*, pages 9–18. Granary Books / Hips Road, 2000.
[11] A. J.-J. L. B. A. Z. A. Pachet, F. and A. Beurive. The cuidado music browser : an end-to-end electronic music distribution system. *Multimedia Tools and Applications*, 2004. Special Issue on the CBMI03 Conference.
[12] F. Pachet. Music interaction with style. In *Proceedings of the International Computer Music Conference*, 2002.
[13] F. Pachet and O. Delerue. On-the-fly multi-track mixing. In *Proceedings of AES 109th Convention*, Los Angeles, USA., 2000.
[14] F. Pachet and P. Roy. Musical harmonization with constraints: A survey. *Constraints*, 6(1):7–19, 2001.
[15] F. Pachet, P. Roy, and D. Cazaly. A combinatorial approach to content-based music selection. In *Proc. of IEEE International Conference on Multimedia Computing and Systems, Firenze (It), Vol. 1 pp. 457-462*, 1999.
[16] R. Rowe. *Interactive Music Systems*. MIT Press, Cambridge, Massachusetts, 1993.
[17] R. Rowe. *Machine Musicianship*. The MIT Press, Cambridge, MA, 2001.
[18] P. Roy, A. Liret, and F. Pachet. The framework approach for constraint satisfaction. *ACM Computing Surveys*, 32(1es), 2000.
[19] P. Roy and F. Pachet. Reifying constraint satisfaction in smalltalk. *Journal of Object-Oriented Programming*, 10(4):43–51, 63, 1997.
[20] E. Scheirer. Tempo and beat analysis of acoustic musical signals. *Journal of the Acoustic Society of America*, 103(1):588–601, January 1998.
[21] D. Schwarz. The caterpillar system for data-driven concatenative sound synthesis. In *Proceedings of the COST-G6 Conference on Digital Audio Effects (DAFx)*, London, UK., September 2003.
[22] D. Schwarz. The first five years of concatenative sound synthesis. In *Proceedings of the International Computer Music Conference*. Barcelona, Spain, September 2005.
[23] A. M.-A. K. Wang, G. and P. R. Cook. Yeah chuck it! => dynamic controllable interface mapping. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME)*, June 2005.
[24] A. Zils and F. Pachet. Musical mosaicing. In *Proceedings of the COST-G6 Conference on Digital Audio Effects*, Limerick, Ireland, December 2001.
[25] A. Zils and F. Pachet. Automatic extraction of music descriptors from acoustic signals using eds. In *Proceedings of the 116th AES Convention, Berlin*, May 2004.