Integrating Constraint Programming in Visual Musical Composition Languages

Camilo Rueda [1] and **Mikael Laurson** [2] and **Georges Bloch** [3] and **Gerard Assayag** [4]

Abstract. Computer aided composition is a domain where the integration of different programming paradigms finds special relevance. We describe the integration of constraint, functional and visual programming in two systems developped at IRCAM. Efficient constraint satisfaction techniques are coupled to iconic visual languages to ease programming of complex problems and to get computed results directly into standard music notation editors. Real composition examples taken from musical pieces actually created are discussed at some length to show the relevance of both systems to non trivial music composition tasks.

1 Introduction

Research on multi-paradigm languages has known increasing interest in the last years. The integration of what appears to be fundamentally different notions of programming has come up as a real need in some domains. One salient example is the realm of computer supported musical composition. Composers define complex hierarchical structures representing multiple musical dimensions which are to evolve according to either (or both) predetermined trajectories or to the satisfaction of a set of compositional rules supplying partial structural information. Both *Object-Oriented* and *Constraint programming* paradigms easily come to mind as relevant for devising music composition tools capable of effectively handling each of these forms of interaction.

In this decade a number of tools integrating objects and constraints for musical applications have been proposed. These are in general based on standard object-oriented or functional languages enriched with libraries for handling constraints. *Situation* ([BR98]) and *PWConstraints* ([Lau93]), both visual constraint programming application developed at IRCAM, are good examples of this approach. The visual language *PatchWork* ([LD89]) or *OpenMusic* ([AA96]) is supplied with a powerful constraint solving engine optimized to solve musical problems that can be represented in terms of a collection of possible objects and a set of properties they should satisfy.

[1] IRCAM, Paris, France and Pontificia Universidad Javeriana, Cali, Colombia. Supported in part by grant 1251-14-041-95 from Colciencias-BID. E-mail: crueda@ircam.fr and crueda@atlas.ujavcali.edu.co.

[2] Sibelius Academy, Helsinki, Finland. E-mail: laurson@siba.fi

[3] Universite des Sciences Humaines de Strasbourg, Strasbourg, France. E-mail:gbloch@monza.u-strasbg.fr

[4] IRCAM, Paris, France E-mail:assayag@ircam.fr.

Submitted to ECAI 98
April 23, 1998

Constraint programming can have far reaching applications in the field of computer assisted composition. In this paper two music composition systems, *PWConstraints* and *Situation* are described. Both are based on the notion of programming as searching for sets of elements satisfying a certain number of given properties or constraints. *PWConstraints* is a *PatchWork* library and *Situation* is an *OpenMusic* library (a version is also available for *PatchWork*). Both systems offer constraint satisfaction capabilities for constructing musical structures. They differ significantly in the strategies and algorithms used and also in their relative knowledge about the application domain. Much like *PatchWork*, PWConstraints is neutral in the sense that it makes no attempt to favor applications that fit into a particular notion of a musical object. In fact, as is also the spirit of *PatchWork*, objects are musical only if the user chooses to interpret them as such. The elements that the search engine of *PWConstraints* tries to find does not necessarily have any structure, although complex *PatchWork* musical structures can also be involved in the exploration.

Situation on the other hand, defines a very general structure for the objects it deals with. They should be expressible as sets of points and distances in a certain space. This structure is very general since it can easily accommodate musical objects such as chords, chord sequences, voices, rhythmic patterns, etc. But the fact that Situation knows already the structure of its objects allows it to greatly optimize solution searching in many practical applications. Obviously, not every problem can be conveniently expressed as that kind of structure, so Situation has a way to consider unstructured objects but in this case the optimizations do not apply. Each system is described in more detail in the following.

We give next a very summary description of *PatchWork* and *OpenMusic*. More details of these languages together with their place within the research program at IRCAM can be found in [CAR98].

1.1 PatchWork

PatchWork (PW) is a language for computer-assisted composition, although it can also be used as a general purpose programming tool. Written in Digitools Macintosh Common Lisp (MCL), PW provides a graphical interface to the Lisp language. Any Lisp function can be translated to a graphically operable box. Programming consists of making connections between boxes. PW is musically neutral in the sense that it does not make assumptions about what kind of music or musical raw material is to be produced or analyzed with it. The main aim is to give the user basic tools for visual programming and to provide a straightforward correspondence with the base language, Common Lisp.

Much like in most graphical environments, PatchWork's interface is event-driven. It consists of windows in which different kinds of graphical items can be positioned, moved and edited in various ways according to actions triggered by external events such as mouse clicking or key pressing. PatchWork is a visual language based on the notion of *patch.* A patch is a layout of visual elements (or views) in a window. The views in a PatchWork patch are box frames enclosing one or more input rectangles and exactly one output rectangle. An output rectangle can be linked to one or more inputs of different boxes by connecting lines. Each box represents the invocation of a particular function. Lines linking boxes define functional composition (see figure 1). A visual patch is thus intended to define a very simple model of computation in which boxes take the role of function calls. Procedural abstraction is added by allowing a whole patch to be visually collapsed into a single abstraction box with suitably defined inputs and output. A powerful feature of PatchWork is the possibility of defining boxes that compute editable data structures. These can then be interpreted as musical structures that are amenable to visual manipulations in a variety of supplied editors. Figure 1 shows the same computed data visualized as a break point function, a musical structure displayed in standard music notation and an editable cell array.

1.2 Open Music

OpenMusic is a highly visual compositional environment on the Macintosh. Developed at IRCAM, it is intended as a superset of *PatchWork.* While drawing benefit from the huge amount of knowledge and experience gathered around the *PatchWork* project, *OpenMusic* implements a set of new features that makes it a second generation compositional software. Constructed on top of Digitool MCL, *OpenMusic* provides a visual programming interface to Lisp programming as well as to CLOS. *OpenMusic* is an Object Oriented visual programming environment. Objects are symbolized by icons that may be dragged and dropped all around. Most operations are performed by dragging an icon from a particular place and dropping it in an other. These places include the *OpenMusic workspace* as well as the Macintosh finder itself. A rich set of classes implementing musical data and behavior are provided. They are associated with graphical editors and may be visually subclassed by the user to meet specific needs. Different representations of a musical process are handled, including common notation, MIDI piano-roll, sound waveforms, breakpoint functions. High level in-time organization of the music material is available through the concept of *maquette*, which is a frame with an explicit representation of time where any data or computational object can be positioned. The music notation in *OpenMusic* is handled by *CMN*, a portable Common Lisp library designed by Bill Schottstaedt ([Sch95]) at CCRMA. CMN has been extended for Quickdraw compatibility and some interactive editing features have been added.

2 PWConstraints

2.0.1 Basic Environment

PWConstraints is a rule-based programming language in which the user writes rules to describe the end result from many different points of view. *PWConstraints* has been integrated into PW so that PW music editors can be used both for input and output. This increases the usefulness of the system as the results can be inspected and manipulated in standard music notation. A more complete description of the system can be found in [Lau93] and [Lau96].

A search problem is formalized in two steps. Firstly, a *search space* is defined as a set of *search variables.* Each search variable has a *domain* containing a list of values. Secondly, a set of rules are written. In a rule a pattern-matching language is used to extract relevant information from a potential solution. This information is given to a Lisp predicate that either accepts or rejects the current choice made by the search engine. The rules are compiled to efficient Lisp functions. The pattern-matching part of a rule uses a fairly standard pattern-matching syntax. It can contain variables (symbols beginning with the character "?": e.g. ?1, ?2, ?foo), anonymous variables (plain "?"), wild card ("*") and index variables (symbols that begin with the character "i" and end with an integer: e.g. i1, i2, i12). A variable extracts single values from a partial solution and is typically used in conjunction with a wild card. By contrast, an anonymous variable is never bound to a value, i.e. it only acts as a "place-holder" in the pattern. The wild card matches any continuous part of a partial solution. Finally, an index-variable extracts values from an absolute position. The pattern-matching part is followed by a Lisp code part that begins with the expression (?if ¡Lisp code¿). Besides the variables and index-variables the Lisp code part can refer to special reserved variables l and rl. l is the partial solution (including the current candidate) found so far by the search engine. rl is the same list but in reversed order. As an example, a rule disallowing all two adjacent equal values in a result is written as follows (this rule uses a wild card and two variables):

$(*?1?2(?if(/=?1?2))$ "no equal adjacent values")

The user can also define preferences by heuristic rules. These are similar to the ordinary *PWConstraints* rules but the Lisp code part of a heuristic rule returns a numerical value instead of a truth value (heuristic rules never reject candidates). This scheme allows the search engin to sort candidates that are accepted by the ordinary rules. The search engin tends to favor candidates with high numerical values. For instance a heuristic rule that prefers large intervals can be written as follows:

$(*?1?2(?if(abs(-?2?1)))$ "prefer large intervals")

Two classical constraint satisfaction algorithms (*forward checking* and *back jumping.* See [Nad89] and [Dec90]) are supplied to allow the user to optimize a difficult problem. Forward checking is used when some relation in the result is given in advance. By contrast, back jumping is more useful when the search reacts dynamically to the state of the partial solution. A typical case is for instance a melodic rule that allows only a given amount of repetitions within a subsequence notes (we don't know in advance the exact positions where the repetitions will occur).

The current version has also a new powerful tool that allows

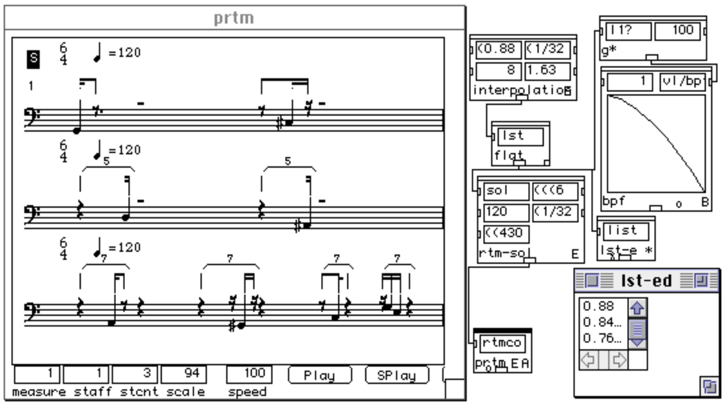

Figure 1. A PatchWork program

to run in parallel several independent copies of the same problem. In each copy the domains are ordered differently. This can be interesting in difficult search problems where out of a good number of trials only one or a few will produce a result in a reasonable time. This tool can be used in two modes. In the first one the user can interactively kill unsuccessful searches and restart new ones. In the second one the system is run under a supervisor process. It automates the task of killing and creation of searches according to a given deadline schedule. The deadline schedule gives time limits that specify the targets that must be reached. If a search does not meet these constraints it will be killed and a new one will replace it. The supervisor process will stop when the desired number of solutions has been found.

2.0.2 Score-PMC

Besides these basic tools *PWConstraints* contains several extensions. The most important and complex one is used to solve polyphonic search problems. This is accomplished with a function called *Score-PMC*. Like in a traditional polyphonic score, the user operates with several layers (parts, voices) of events (notes). The rhythmic structure of a search problem is prepared in advance in a standard PW music notation editor. This input score (which can be arbitrary complex) is given as an argument to the search engine. The search, in turn, aims at filling the input score with pitch information according to the given rules.

The search-engine works internally only with a flat queue structure. Due to the complexity of the input score *Score-PMC* has to make several preparation steps before starting the search. Firstly, *Score-PMC* collapses a given polyphonic score into a flat list of search variables. Each note of the input score is represented in the search engine by a search variable. The critical point is to determine the exact position of each note in the final flat queue structure. This ordering is done by a score sort algorithm. Score sorting works as follows. The score is read from left to right and notes are taken as they areencountered. If two or more notes share the same attack time, they are sorted so that longer notes are placed before shorter ones. If two or more notes have the same attack time and the same duration the order is arbitrary. The default convention is that notes having the highest part number are considered first. Secondly, *Score-PMC* has to preserve the musical information of the original input score. Thus each note must have knowledge of its melodic line (e.g. predecessor, successor, metric context, part, measure, beat), of its harmonic context (i.e. which notes are currently sounding and have already a solution), of its harmonic slice (i.e. which notes are sounding with the current note), etc.

In Score-PMC the pattern-matching part refers to note successions of a melodic line. In a sense Score-PMC can be seen as a multi-layered search problem where each melodic line represents one queue structure similar to the one used by a simple search problem. Also, the variables refer in this case to complex objects and not simply to values as before. This scheme allows relevant information to be extracted from the objects. A typical example is the expression $(m?1)$ that returns the current *midi* value of a variable having the name ?1. Thus the "no equal adjacent values" rule given above can be rewritten in Score-PMC as (this rule is run for each melodic note pair):

$(*?1?2(?if(/=(m?1)(m?2)))$ "no equal adjacent melodic pitches")

A harmonic rule disallowing unisons and octaves can be written as follows (this rule is run for each note):

$(*?1(?if(not(member(m?1)(hc-midis?1) : key\#'mod12)))$ "no unisons or octaves")

Besides the function m this rule contains some useful utility functions (currently *PWConstraints* has a library of over 200 predefined user functions): *hc-midis* returns the midi values of the harmonic context of ?1 and *mod12* returns the modulo 12 of its argument.

In *Score-PMC* the pattern-matching has been extended so that the user can easily add restrictions to the application of a rule. For instance, the "no equal adjacent melodic pitches" rule can be rewritten so that it is only applied for parts 1, 2 and 3 (this rule uses the keyword :partnum that refers to the part number):

$(*?1?2 : partnum(123)(?if(/=(m?1)(m?2)))$
"no equal adjacent melodic pitches")

Other similar keywords (such as :measurenum, :beatnum, etc.) can be mixed in the pattern-matching part.

Lately the multi-layered search idea has been extended to cover parameters other than pitch. A new function called *Texture-PMC* uses internally the Score-PMC function and is capable of producing complex rhythms and textures. The input score consists of a dense pulse of attacks. Each input attack can be interpreted as either an attack, tie or rest defining the rhythmic structure of the result. Furthermore a resulting attack can consist of any PW chord object. Due to the flexible nature of the PW chord object an attack can thus have several textural interpretations: it can be a simple PW one note chord with various dynamic levels, it can be a multi-note chord resulting in attacks with varying note density, it can consist of a multi-note chord where the notes have varying offset times (relative to the start time of the chord) resulting in "grace chords", "clouds of notes", etc. A chord can also be attached a label (such as "pizz" "harm" "gliss") allowing for instance to define different modes of playing when working with instrumental parts.

2.0.3 Engine

Engine is a composition by Magnus Lindberg for chamber orchestra written for and commissioned by London Sinfonietta. It was premiered in 1996 and the duration is 16 minutes. Both the rhythmic and pitch structure of Engine was realized with the help of PatchWork. The piece is divided into 38 sections. Each section was translated into Enigma file format. The result was then read into *Finale*, where the final orchestration was done by hand. The rhythmic structure was accomplished with a PW user library called LeLisp ([Lin96]). Once a metric score done, the next step was to fill it with pitch information. This part of the task was in turn realized with *PWConstraints*. For realizing the pitch structure of Engine the main tool was naturally *Score-PMC* as the rhythmic structure for each section was calculated in advance. we give next an overview of the rules that were used during the search. The description is greatly simplified and does not cover all the rules used during this project (for instance, heuristic rules are not discussed at all).

In order to produce musically meaningful results a section needs typically from 30 to 40 rules. These rules can be divided roughly in four main categories: melodic, harmonic, voice-leading and crossing-ambitus. Melodic rules can be divided into two main groups: interval control and control of repetitions of melodic cells. The composer developed during the realization process a personal library of allowed interval successions. This was done in order to personalize the musical material and to treat problematic cases. Another type of interval rule verifies that in every group of adjacent notes (the length varies from three to six) the minimum and maximum pitches should not form octaves. Some interval rules are sensitive to the rhythmic context. Fast successions of notes have a different interval character than slow ones. The control of melodic cells consists of avoiding direct repetitions within a given window size. Harmonic rules forbid octaves (unisons are allowed) and require that vertical interval formations have a given *set-class* identity. Often chord formations having a long duration are controlled more strictly than shorter ones. Voice-leading rules are always applied for two voices (or parts) at a time. A cross-identity rule forms a 2*2 matrix where the first row consists of a two-note melodic succession of a given part and the second row of a simultaneous two-note succession of another part. The rule states that the diagonal pitch-class relations in this matrix should not be equal. A similar more strict version of this rule exists where the matrix size is 2*3. Another voice-leading rule states that two parts should result in an almost parallel movement. Voice-leading rules can easily be chained to control more than two parts. For instance, when the last mentioned rule is applied, say, to parts 1 and 2, and then to parts 2 and 3, the result is that the parts 1-3 follow the same rule. Like voice-leading rules crossing-ambitus rules are applied for two voices. They deal with cases where one wants to prevent two parts from crossing each other (voices can form unisons). This category contains also an ambitus rule that controls the minimum and maximum distance between two voices. One of the most important targets in this project was that the user should be able to "localize" rules easily for certain musical situations. For instance, a melodic rule may react differently to rhythmic two-note patterns (such as short-short, short-long, long-short, long-long), chord formations that have a long duration are controlled more strictly than short ones, parts may form different melodic and voice-leading textures (for instance parts 1-2 may use repetitions and large skips, parts 3-4 can be more static but avoid repetitions, parts 5-8 can be controlled by rules that force them to move in an almost parallel fashion), etc. An example of this kind of multi-layered texture is given in figure 2. It shows the beginning of section 2 (the score is written with usual transpositions).

3 Situation

situation allows the construction of objects out of two notions: *point* and *distance*. An object contains one or more points, spaced according to distances which are evaluated in a given unit of measure and with respect to a user supplied distance function. Distances can be *internal* or *external* depending on (respectively) whether they involve points contained in the same object or in several different objects. Intervals in a chord, for example, are internal distances whereas melodic intervals are external. The unit of measure can in this case be the semitone, the eighth tone or any other. By default, points are considered to have integer coordinates in a standard cartesian space. The sequence [60,64,67], for example, could represent an object (chord) consisting of three points (notes, pitch expressed in Midics) separated by distances of 4 and 3 semitones. The objects could also be rhythmic plans concerning a set of articulation points spaced according to a given set of temporal distances. An eighth note, quintuplet or other could be the unit of measure in this case. The sequence [31/4,65/8,67/8], for example, might represent the articulation points of a *voice* object. The unit of measure is 1/8 (an eighth note). The first articulation point is located at position 31/4. 3/8 and 1/4 are the distances separating consecutive points in this voice.

The specification of a problem for *situation* consists in supplying the number of objects wanted, the space of possibilities (region) for points, the number of distances in each object and the set of possible values for these distances. Built-in constraints establish the allowed configurations of each object and also that of object combinations. Any point or distance of any

Figure 2. *Engine*. Beginning of section 2.

object or set of objects can be related by a constraint. Built in constraints conveniently define : general profiles that should follow the objects ; patterns that a given set of distances should match ; predetermined points that should belong to every solution ; equality or difference of points or distances (or simultaneity, in the case of rhythms), etc. Figure 3 shows a visual program for the construction of a sequence of chords having some harmonic and melodic properties. The solver engine is represented by a pair of dice icon. In this example, constraints control the constitution of each chord as well as melodic and harmonic properties of chord subsequences. Each constraint icon defines a particular aspect of this control (integers represent number of semitones). These include, among others:

- Patterns of interval content in each chord (icon *vint-filt*). The expression $(not(or(ints12t)(*\ 6\ *)(*\ 2\ 2\ 2\ *)\ldots))$ constraints chords to not having octaves, augmented fourth or sequences of three major seconds, etc.
- Patterns of notes in each chord (icon *pitch/ch-filt*). The expression $(0(65\ 67\ 70\ 72)34(61\ *)\ldots)$ constrains the first chord to be (F,G,A#,C), the thirty-fifth to contain C#, etc.
- Single voice profiles, controlling the number of consecutive upward and downward movements (icon *v-prof*). The expression $(0\_27(max\ 3)\ldots)$ requires the upper voice for the first 28 chords to have a maximum three consecutive movements in the same direction.
- Relative movements of two voices (icon *v/v-prof*). Expression $(0\_27(0\ u\ (max\ 3)\ldots))$ imposes for the lower and upper voices of the first 28 chords a maximum of 3 consecutive parallel movements.
- Patterns of melodic intervals (icon *hint-filt*). The expression $(0(not(or(2\ 2\ 2)\ (3\ 3)\ldots)))$ constrains the lower voice to not contain three consecutive major seconds or two consecutive minor thirds.
- Ambitus evolution, defined by an interpolation forcing the chords gradually into an octave above middle C: $(interp(54\ 77)(61\ 77)))$

The visual program in figure 3 generates 96 chords. It is a simplified version of a patch used to generate an ostinato in the orchestral piece *Epitaph*, commissioned by IRCAM and written for the *Ensemble Intercontemporaine*, by the french composer Antoine Bonnet.

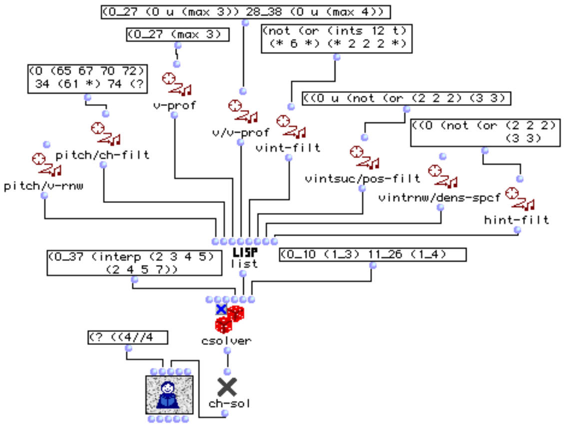

Figure 3. *Situation*: Harmonic problem definition

As mentioned before, *Situation* can also be used to generate sequences in other dimensions of a musical structure. In figure 4 each generated object contains the rhythmic pattern of one voice. Distances between points in each voice are here "measured" in a different unit: A 7-tuplet (1/28), a 6-tuplet(1/24) and a triplet(1/12). The general minimum approximation unit is set to the 7-tuplet. A constraint positions the first point of the first voice at less than 9/28 units of the beginning. Other constraint controls possible spacings between points within a voice. Each point in a voice is an articulation point where a suitable motive will be attached. A user-defined constraint contained in the patch "points-cnstr" requires than no two motives (motive size is constrained to lay within a given range) in the same or in different voices overlap. The icon *rtm-sol* associates articulation points with their motive notes. The latter are also computed (in an independent patch) by *Situation*. The score in figure 4 shows one measure of a computed solution.

3.0.4 The constraint engine

Situation is a finite domains system. Each computed object is by default represented with two domain variables. The first variable defines the position of the first point of the object and the second variable defines the sequence of distances separating each consecutive point in the object. The domain of the first variable is any finite set of numbers. The domain of the second is a finite set of *sequences* of numbers. The latter domain is usually large.*Situation* allows it to be structured in a tree hierarchy. A subset of the sequences sharing a given property can be collected into a subtree in this hierarchy. At the level of the root of this subtree, the whole subsequence is represented by a domain of just one value: The shared property. A collection of (user supplied) functions compute properties to be used to abstract the sequences domain at a specified level. By default, *Situation* uses two levels, with the sum of the object's distances as the abstracting property for the highest level. This is very convenient for harmonic problems, where the upper and lower voices are usually more constrained than the others.

The notion of "distance" in *Situation* is not fixed. The composer can define her/his own by supplying the appropriate functions (normal, inverse) and neutral element. In musical applications this option can be very important. For example, some composers conceive harmonic material as aggregates of frequency *partials* related in precisely defined ways. Multiplicative distances are more relevant in this case.

The search engine of *Situation* uses *first-found forward checking* (F3C. [RV97]), a lazy-evaluation version ([DM94] of *forward checking* extended to hierarchical domains. Each domain level keeps track of the position of the current *consistent* value at that level. These are values known to satisfy all constraints referring to that level. A judicious choice of data structures allows *Situation* to efficiently update these current positions as new constraints are checked or when backtracking is needed.

No varaible reordering is used (although included as an option) since constraints for musical problems generally apply within short subsequences, with little dependencies across subsequences. All domain values are randomly permuted prior to exploration. The reason for this is that the musician is in most cases interested in obtaining few but widely different solutions to a given problem. Due to domain permutations, any new execution of the same problem is likely to give a solution with different values for many of the variables.

Constraints can relate any level of the domain of one variable to any level of the domain of any others. *Arc consistency* ([Mac77]), via AC-7+ ([VC96], an enhancement of the algorithm in ([CBU94]) can also be performed for binary constraints over upper domain levels. In the next subsection we develop in some detail a musical example involving interaction between rhythm and harmonic and melodic control.

3.0.5 Palm Sax*: A musical example*

Situation was used in the last movement of a piece called *Palm Sax* by the french composer Georges Bloch. *Palm Sax* is a composition for saxophone ensemble. Saxophones are organized in three groups, a soloist (tenor sax) and two groups of three saxophones. In the last movement these two groups are formed of an upper register group of soprano and alto and a lower register group of one alto and two barytone saxophones. This movement plays over two types of oppositions: A contrast between an improvised variation based on a given grid and a polyrhythmic structure. The grid rests on a rather traditional structure of subtil variations from a fixed tempo. The polyrhythmic part is based on a sort of inverted canon. Its structure is formed of "pages" of identical rhythmic pattern and variable tempo. The coexistence of page fragments of different speeds enforces an impression of "unfinishedness". At the harmonic level the opposition plays on the difference between "regular" and "fragmentary". The fragmentary part has a textural dominance. The combination of harmonic aggregates in this part creates a repetitive harmonic functionality. Different aggregates are formed by the repetition of chords in two variable temporal grids. The aggregates thus formed

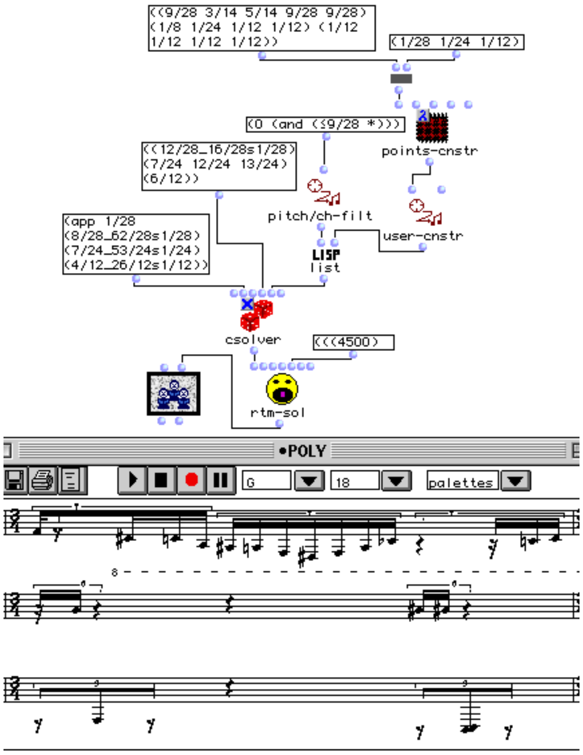

Figure 4. *Situation*: Rhythm generation

must follow a fixed given harmonic function which is itself repeated in a fixed temporal grid. Constraints are used to find the right harmonic aggregates.

There are two key concepts in the musical model: *Estrada distance* and *Hindemith fundamental*. A *Estrada chord* is one for which the sum of the intervals (modulo 12) between consecutive notes does not exceed an octave. All chords in the piece are *Estrada*. The *signature Sig* of a *Estrada* chord C is defined as

$Sig(C) = ints(C) \cup \{12 - \sum_{i \in ints(C)}(i \ mod 12)\}$, where $ints(C)$ is the set of intervals between consecutive notes in chord C.

The *Estrada distance* between two chords C_1, C_2 is defined thus

$distance(C_1, C_2) =$
$Max(\#Notes(C_1), \#Notes(C_2)) - |Sig(C_1) \cap Sig(C_2)| - 1$, where $\#Notes(C)$ is the number of notes in chord C.

The *Hindemith fundamental* of a chord C is defined to be the lowest note in C forming an interval of a fifth with any other note of the chord. If no such interval exists then the lowest note in C is chosen.

In the piece the upper voice is constrained by requiring *Estrada distance* equal to one, between every pair of chords in positions i and $i + 11$. Lower voice and upper voice chords whose "living time" intersect are mixed together into a single chord. Each different chord thus formed should be such as to have a particular *Hindemith fundamental*.

Figure 5 shows a patch implementing this model. Variables $v_0, \ldots, v_{47}$ correspond to the first 48 chords in the lower voice. Variables $v_{48}, \ldots, v_{167}$ refer to the first 120 chords of the upper voice. The entries to the constraint solver (the pair of dice) specify,

- The allowed subspace for the generated objects. In this case the pitch region allowed for the evolution of each voice. For the lower voice, this is defined by an interpolation from the register comprised between D2 and D3 for the first chord, to that comprised between C3 and C4 for the 47-th chord. For the upper voice, by an interpolation from the region comprised between C3 and C4 for the first chord, to that comprised between A4# and C5 for the 120-th chord.
- The set of allowed internal distances within an object. In this case the set of possible harmonic intervals within each chord. These are defined to be the set of possible intervals of *Estrada* chords consisting of one, two or three notes. They are computed in patch *vert-intervals*.
- A collection of constraints.

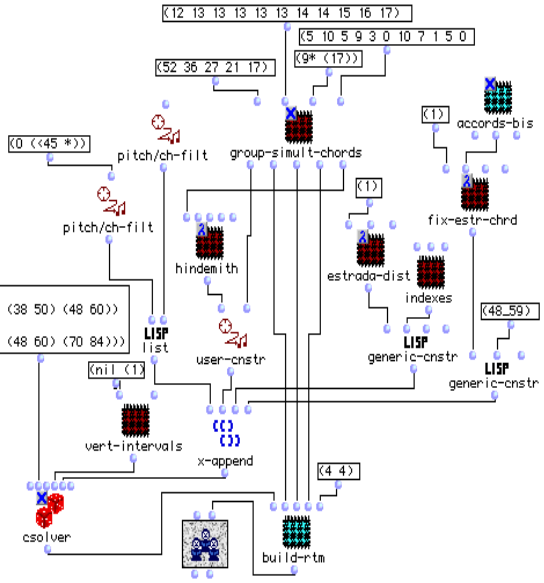

Figure 5. *Palm Sax*: Patch in *Situation*

The rhythm of *Palm Sax* is given by the composer. Each chord has an associated temporal position given as a distance from the beginning of the piece. The unit of measure is the sixteenth note. Each chord sounds until the beginning of the next chord in the same voice.

In figure 5 icons *pitch/ch-filt*, *user-cnstr* and *generic-cnstr* implement different types of constraint *schemes*. Each scheme defines a predicate to be used for constraining evry pair of consecutive chords within specified subsets of the variables. The actual subset is either given directly in one of the inputs to the icon, as in the expression (48_59), or computed by a patch, as for the input to *user-cnstr* comming down from patch *group-simult-chords*.

The subset of variables to be constrained by predicate *hindemith* is computed by patch *group-simult-chords* by considering rhythmic specifications for the voices supplied in its first (from the left) two entries. These rhythm data is used in the patch to compute the indexes of chords sounding together at each basic global "beat" of 17 time units. These chords should be constrained to mix (in pairs, one from each voice) into a chord having a hindemith base (modulo 12) equal to the one supplied for that beat (see the list linked to the fourth input to *group-simult-chords*). Predicate *Hindemith* implements this constraint. The corresponding patch icon has five inputs. The first one comes from *group-simult-chords* and contains the list of Hindemith bases for each pair of mixed chords. The rest of the inputs represent the variable index and the notes of each chord involved in this constraint. They are supplied by the search engine as new instanciations of chords are proposed.

Patch *estrada-dist* implements the above mentioned *Estrada*distance constraint for the upper voice. Icon *fix-estr-chord* defines an additional constraint, namely that the first twelve chords in the upper voice should be each at a *Estrada* distance equal to one from a supplied corresponding chord (computed in patch *accords-bis*).

The two icons *pitch/ch-filt* constraint the first chord of each voice to a restricted zone within the allowed pitch region. The expression $(0(< 45\ *))$ makes the base note of the first chord in the lower voice to be lower than 45 (i.e. A3).

Finally, patch *build-rtm* associates to each temporal position the corresponding chord computed by the constraint solver. It also performs a quantification to translate absolute temporal positions into standard OpenMusic rhythm notation.

The domain of each one of the 168 computed chords consists of about 800 possibilities. There are about 300 constraints in the problem definition. A section of the beginning of the last movement of the piece is given in figure 6.

Figure 6. *Palm Sax.* Computed seven measures.

4 Conclusions

REFERENCES

[AA96] G. Assayag and C. Agon. Openmusic architecture. In *Proceedings of ICMC'96*, Hong Kong, China, 1996. MIT press.

[BR98] A. Bonnet and C. Rueda. *Situation: Un Langage Visuel Basee sur les Contraintes pour la Composition Musicale.* HERMES, Paris, France, 1998. To appear in: Recherches et Applications en Informatique Musicale. M. Chemillier and F. Pachet (Eds).

[CAR98] M. Laurson C. Agon, G. Assayag and C. Rueda. Computer assisted composition at ircam: Patchwork and openmusic. *Computer Music Journal*, 1998. to appear.

[CBU94] Freuder E.C. C. Bessière and Regin J.C. U. Using inference to reduce arc consistency computation. In *Proceedings of ECAI'94*, Amsterdarm, The Netherlands, 1994.

[Dec90] R. Dechter. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence*, 41:273–312, 1990.

[DM94] M.J. Dent and R.E. Mercer. Minimal forward checking. In *6th International Conference on Tools with Artificial Intelligence*, New Orleans, USA, 1994.

[Lau93] M. Laurson. *PWConstraints*. Associazione di Informatica Musicale Italiana, Milano, Italia, 1993. in: X Colloquio di Informatica Musicale, G. Haus and I.Pighi (Eds).

[Lau96] M. Laurson. Pwconstraints reference manual, 1996. Available through IRCAM user's group.

[LD89] M. Laurson and J. Duthen. Patchwork, a graphical language in preform. In *Proceedings of ICMC'89*, San Francisco, USA, 1989. MIT press.

[Lin96] M. Lindberg. Lelisp library, 1996. A library of musical functions. unpublished.

[Mac77] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8:99–118, 1977.

[Nad89] B. Nadel. Constraint satisfaction algorithms. *Computational Intelligence*, pages 188–224, 1989.

[RV97] C. Rueda and F. Valencia. Improving forward checking with delayed evaluation. In *Proceedings of CLEI'97*, Santiago, Chile, 1997.

[Sch95] B. Schottstaedt. Cmn. A Common Lisp music notation editor. available at http://ccrma-www.stanford.edu/CCRMA/Software/cmn/cmn.html, 1995.

[VC96] F. Valencia and Rueda C. Uso de deducciones de no viabilidad en el calculo de arco-consitencia. In *Proceedings of CLEI96*, Bogota, Colombia, 1996.