


- Research reports
- Musical works
- Software

# PatchWork

## Situation

*Library for musical constraints programming*

*First English Edition, February 1996*

IRCAM  Centre Georges Pompidou

© 1996, Ircam. All rights reserved.

This manual may not be copied, in whole or in part, without written consent of Ircam.

This manual was written by Antoine Bonnet and Camilo Rueda, translated into English by J. Fineberg, and was produced under the editorial responsibility of Marc Battier, Marketing Office, Ircam.

PatchWork was conceived and programmed by Mikael Laurson, Camilo Rueda, and Jacques Duthen.

The Situation library was conceived and programmed by Antoine Bonnet and Camilo Rueda.

First English edition of the documentation, February 1996.

This documentation corresponds to version 1.0 of the library, and to version 2.1 or higher of PatchWork.

Apple Macintosh is a trademark of Apple Computer, Inc.  
PatchWork is a trademark of Ircam.

**Ircam**  
**1, place Igor-Stravinsky**  
**F-75004 Paris**  
**Tel. (33) (1) 44 78 49 62**  
**Fax (33) (1) 42 77 29 47**  
**E-mail [ircam-doc@ircam.fr](mailto:ircam-doc@ircam.fr)**

---

# IRCAM Users' group

The use of this software and its documentation is restricted to members of the Ir-cam software users' group. For any supplementary information, contact:

Département de la Valorisation  
Ircam  
Place Stravinsky, F-75004 Paris

Tel. (1) 44 78 49 62  
Fax (1) 42 77 29 47  
E-mail: [bousac@ircam.fr](mailto:bousac@ircam.fr)

Send comments or suggestions to the editor:  
E-mail: [bam@ircam.fr](mailto:bam@ircam.fr)  
Mail: Marc Battier,  
Ircam, Département de la Valorisation  
Place Stravinsky, F-75004 Paris

---



To see the table of contents of this manual, click on the **Bookmark Button** located in the **Viewing** section of the **Adobe Acrobat Reader** toolbar.

# Contents

Résumé .....	6
Introduction .....	8
General Principals .....	9
Constraints and the instantiation of musical objects .....	9
Notion of "weak" constraints .....	10
The Situation Library: Reference .....	11
Harmonic-constraints .....	12
n-solutions .....	12
n-accords .....	13
ambitus .....	13
densité .....	13
int-vert-filt .....	14
vs-filt-hori .....	14
vi-filt-hori .....	14
vs-seuil-rep .....	14
fixed-notes .....	15
rep-par-dens .....	15
renouv-haut .....	15
filt-pas-band .....	15
int//forbid .....	15
nb-mouv//ok .....	15
nb-dir=ok .....	16
User-constraints .....	16
Prev-Solutions .....	16
paramètres .....	16
set-param .....	17
construct-sols .....	17
part-sols .....	18
rhythm-constraints .....	18
construct-rhythm .....	18
total-notes .....	19
dist-into-units .....	19
groups-per-units .....	19
notes-per-units .....	20
notes-per-group .....	20
constraint .....	20
solve .....	21
funarg .....	22
parameter .....	22
form-pattern .....	23
interval-interp .....	23
funct-and .....	24
funct-or .....	24
funct-not .....	24
Menus of the Situation library .....	25
A selection of examples .....	27
Index .....	40

---

# Résumé

Situation est un moteur de satisfaction de contraintes écrit en Common-Lisp-CLOS et muni d'un interface graphique en PatchWork adapté à la construction de séquences harmoniques et rythmiques.

Situation est particulièrement pertinent pour la génération des objets musicaux lorsque :

- chaque objet dans une séquence est une entité décrite par un ensemble donné de propriétés reliant ses éléments;
- des objets dans des positions sélectionnées de la séquence doivent satisfaire un certain nombre de contraintes;
- les relations entre les objets établissent des contraintes souhaitables mais pas forcément obligatoires.

Situation permet ainsi la construction de séquences musicales à partir de la description de leurs propriétés - contrairement à l'approche fonctionnelle qui nécessite la spécification précise de l'algorithme de construction.

Certains modules de Situation exigent de posséder Common Lisp, ainsi qu'une image de PatchWork contenant le compilateur. Common Lisp peut être obtenu dans le commerce; l'image de PatchWork avec compilateur est distribuée par l'Ircam. Les objets réclamant cet environnement sont<sup>1</sup> :

- constraint
- funarg
- parameter
- solve

La construction et la manipulation d'objets musicaux se trouvent au centre de l'activité compositionnelle. Ces objets se conçoivent généralement en terme de relations entre leurs composantes. Dans les cas les plus simples, la construction d'objets musicaux se fait de façon convenable au moyen de l'élaboration d'un algorithme spécifique, c'est-à-dire qui sache placer chaque composante selon la relation à satisfaire.

Quand la relation est complexe ou quand un nombre important de relations doit être satisfait, la définition de l'algorithme adéquat est lourde et difficile. La meilleure solution est alors de décrire les relations souhaitées afin de laisser la machine construire le bon objet. Les systèmes de résolution de contraintes, tel que Situation, ont cette approche.

Situation est conçu pour un utilisateur non expérimenté en informatique (si l'on fait exception des quatre modules cités plus haut, qui, eux, sont expérimentaux, et réclament une connaissance de la programmation en Common Lisp). L'inconvénient de cette exigence est le temps parfois très long nécessaire à l'obtention d'un bon résultat. Pour un environnement à vocation interactive, c'est un problème à prendre en considération.

---

1. La compréhension et l'emploi de ces modules expérimentaux requiert un niveau avancé en programmation.

Par souci d'efficacité, Situation a donc été optimisé en fonction des relations harmoniques et mélodiques les plus probables compte tenu du parti pris de départ de placer la notion d'accord au centre du projet. La possibilité de paramétrer les contraintes avec l'interface PatchWork en témoigne. Naturellement, les relations y figurant ne sont pas les seules possibles : leur présence est plus une question d'efficacité que de capacité.

Très généralement, on peut envisager le processus de résolution d'un problème de satisfaction de contraintes comme celui des raffinements successifs des solutions partielles. Pour trouver une séquence harmonique particulière, on pourrait partir d'une "solution" comportant toutes les séquences d'accords possibles et filtrer celles-ci par des contraintes permettant de sélectionner autant de sous-ensembles. Une solution valide serait ainsi l'intersection de tous les sous-ensembles.

L'utilisateur doit donc tenir compte du fait que si l'ensemble initial est plus restreint, le processus de résolution est plus efficace. Trouver par exemple des séquences particulières d'accords de sept hauteurs est beaucoup plus efficace si l'on impose un certain registre aux accords que si l'on considère au départ tous les accords possibles. Cet ensemble initial est appelé *Domaine* dans le jargon informatique.

### **Notion de contraintes "faibles"**

L'interaction des contraintes données en entrée du module harmonic-constraints peut dans certains cas définir un problème inconsistant, sans aucune solution possible. Le module rendra nil<sup>2</sup> en sortie mais après avoir exploré tout l'espace de recherche, ce qui peut entraîner un temps de calcul très important. Or, il n'est pas toujours évident de trouver la façon d'exprimer un problème en évitant cet écueil.

Situation propose donc de hiérarchiser les contraintes selon une échelle d'importance ou de nécessité. L'on peut associer à chaque contrainte un nombre entre zéro et 1 qui indique son importance relative. Une valeur de 1 (la valeur par défaut) signifie une contrainte obligatoire (à 100%); une valeur de zéro rend la contrainte inutile; une valeur intermédiaire donne son importance par rapport aux autres. Dans la figure 2, certaines contraintes s'inscrivent à l'intérieur d'une liste commençant par :

(n <degré> ...)

Ceci donne à la contrainte concernée un nombre <degré> correspondant à sa valeur de nécessité. Ce schéma induit aussi une hiérarchisation des solutions au problème. Une solution à valeur 1 est celle où toutes les contraintes ont été satisfaites. Une solution à valeur  $v$  inférieure à 1 est celle où n'a pas été satisfaite, à un degré de nécessité inférieur ou égal à  $1 - v$ , la contrainte non obligatoire la plus importante.

Les solutions sont calculées en ordre croissant de valeur.

---

2. En Common Lisp, nil représente une liste vide ou la valeur booléenne 'FAUX'.

# 1 Introduction

Situation is a calculation engine for satisfying constraints. It is written Common-Lisp-CLOS and makes use of a graphic interface within PatchWork that is well suited to constructing harmonic and rhythmic sequences.

Situation is particularly useful for generating musical objects under the following conditions :

- when each object within a sequence may be described by a group of characteristics which are, in fact, the same characteristics which grant cohesion between the constituent objects of that sequence;
- when objects at selected positions within a sequence must fit certain constraints;
- when the relations between objects should preferentially, but not obligatorily, satisfy certain constraints.

Thus Situation allows the construction of musical sequences based upon a description of their attributes. This approach is contrary to a functional method which requires a specific algorithm to generate material.

The use of certain modules contained in Situation requires the possession of Common Lisp, in addition to a PatchWork image (which contains the compiler). Common Lisp may be purchased commercially; the PatchWork image with compiler is distributed by Ircam. These experimental objects, included with the environment are the following<sup>3</sup> :

- **constraint**
- **funarg**
- **parameter**
- **solve**

---

3. Understanding and using these experimental modules requires an advanced level in programming.



## 2 General Principals

### Constraints and the instantiation of musical objects

Building and manipulating musical objects is central to the process of musical composition. These objects are generally conceived of in terms of the relations existing within their constituent parts. The simplest case is the construction of an object through a specific algorithm, capable of placing each constituent part in satisfactory relation to all others.

When the relations to be satisfied are complex, or numerous, defining a satisfactory algorithm is extremely difficult. The best solution in this case is to simply define the desired relationships in such a way as to allow the machine to find its own route to constructing the desired object. Constraint resolution systems, such as Situation, use this approach.

Situation is intended to be accessible to even novice computer-users (except for the four, above mentioned, experimental modules, which require experience in Common Lisp programming). This convenience of utilization has the unfortunate side-effect of sometimes requiring an extremely long time to find a good result. This is a problem to be considered in any deliberately interactive environment.

In the interest of making the constraint propagation as effective as possible, Situation has been optimized in function of the most likely harmonic and melodic relations (this takes into account the fact that from its origin we have placed the notion of a chord as being central to Situation). The possibility of controlling the parameters of the constraints testifies to this approach. Naturally these 'most likely' are not the only possibilities: their presence being a question of efficacy and not capacity.

In a general way, the problem of resolving a set of constraints can be seen as successive refinements of incomplete solutions. To find a particular harmonic sequence, it would be possible to take as a starting point a "solution" containing every possible chord sequence and then filter the parts of each solution that fail to meet any given constraint. The valid solution will be made up of the intersection of the remaining solution fragments.

The user must keep in mind that the smaller the initial set of possible solutions, the more effectively (rapidly) a solution can be found. For example, in order to generate a particular sequence of seven note chords, it is much more effective to limit the registers of those chords than to allow all possible chords. In computer terminology the initial set of possible solutions is called the domain.

# Notion of "weak" constraints

The interaction between various constraints placed as inputs to the module harmonic-constraints can sometimes provoke an inconsistent problem, one without any possible solution. In this case the module will output a result of *nil*<sup>4</sup>; however, this result will be determined only after the module has completely explored the domain, which may require a very long period of calculation. Avoiding this type of contradiction, and the resulting loss of time is not always easy.

As a solution to this problem, Situation allows constraints to be placed in a hierarchy of importance or necessity. A number between zero and 1 may be associated to each constraint to indicate its relative importance. A value of 1 (the default value) signifies an obligatory constrain (100%). A value of zero makes a constraint meaningless, while an intermediate value establishes the constraints importance relative to all other constraints. In figure 2, certain constraints have been given with a list that starts with :

(n <degree> ...)

This provides the constraint with a number <degree> corresponding to its necessity value. This same scheme also provides a hierarchy to the various problem solutions. A solution with a value of 1 has satisfied all the constraints. A solution with a value *v* of less than 1 is one in which a constraint was not satisfied (the unsatisfied constraint is the most important, non-obligatory, constraint with a necessity value less than or equal to 1 - *v*).

Solutions are calculated in order of increasing values of *v*.

---

4. In Common Lisp, *nil* represents either an empty list or the boolean value 'FALSE'.

# 3 The Situation Library: Reference

The situation library is organized in a hierarchical menu containing four parts:

- **Harmony** - Modules used in generating chord sequences.
- **Rhythm** - Modules used in constructing rhythmic sequences.
- **Constraints** - Modules used to define and solve general problems of constraint satisfaction.
- **Utilities** - Tools for managing the syntactic expressions used as constraint parameters or for the generation of predicates (functions).

The modules in each of these menus are detailed below.

Patches using Situation to generate chord sequences are shown in detail (figures 1, 2 and 3) in section 6. Certain of their inputs are used to specify the initial domain and others to establish the constraints to be satisfied from within that domain.

For the sake of clarity and completeness, we have included an example (figure 1) which uses all the possible constraints. It should go without saying that such exactitude does not always make sense musically and that accumulating large numbers of constraints slows the calculation of the results.

# Harmonic-constraints

n-solutions	n-accords
ambitus	densite
int-vert	int-hori
int-vert-filt	vs-filt-hori
vs-seuil-rep	vi-filt-hori
fixed-notes	rep-par-dens
renouv-haut	filt-pas-band
int//forbid	nb-mouv//ok
nb-dir=ok	
harmonic-constraints	
E	

n-solutions	n-accords
ambitus	densite
int-vert	int-hori
int-vert-filt	vs-filt-hori
vs-seuil-rep	vi-filt-hori
fixed-notes	rep-par-dens
renouv-haut	filt-pas-band
int//forbid	nb-mouv//ok
nb-dir=ok	user-constraint
prev-solution	paramètres
harmonic-constraints	
E	

Syntax: **harmonic-constraints** *n-solutions n-accords ambitus densite int-vert int-hori int-vert-filt vs-filt-hori vs-seuil-rep vi-filt-hori fixed-notes rep-par-dens renouv-haut filt-pas-band int//forbid nb-mouv//ok nb-dir=ok* & optional *user-constraint prev-solution parametres*

Defines chord sequences. The module has 20 inputs:

- 5 for defining the domain (1 to 5),
- 12 for defining the constraints placed on that domain (6 to 17),
- 3 special optional inputs (18 to 20); these inputs only appear by performing an option-click on the letter E for each optional input desired (the module is extensible).

Clicking on the module's bottom left toggles the display to show the input names.

The following is an explanation of the 17 principal and three optional inputs to the **harmonic-constraints** module.

## n-solutions

Number of solutions desired (drag<sup>5</sup> with the mouse).

## n-accords

Number of chords desired.

The other inputs (except 18-20) are programmable in two dimensions:

- vertical, range, measured in Midi ((60 66) = C4 to F#4);
- horizontal, position within the sequence (0= first chord, 1 = second, etc.).

## ambitus

Defines the range in which the chords will be placed. This definition may be dynamic in time; for this use, the **interval-interp** module allows the interpolation between two points.

In the example shown as figure 1 in section 6, the first chord will be between A2 and B5 [Ex: 0 (57 83)], the eighth between Bb2 and B5 [Ex: 7 (46 83)], the sixteenth between F3 and B5 [ Ex: 15 (53 83)]. Between these three points the computer will perform interpolations to determine the range for each chord.

The chord by chord results of the interpolation may be displayed in the listener window by evaluating the **interval-interp** module (this is also true for all other modules). Evaluation is performed by option-clicking on the small rectangle in the bottom center of the module.

If no interpolation were desired between, for example, the first seven chords, it would be necessary to program the following:

( 57 83 ) 6 ( 57 83 ) 7 ( 46 83 ) 15 ( 53 83 ) ) .

## densité

Use two numbers (min./max.) to define the number of notes in each chord.

In the example shown as figure 1 in section 6, the first chord will have exactly three notes [Ex: 0 (3 3)], the second either 3 or 4, the eighth exactly 7, the fourteenth 3, 4 or 5, etc.

The densities of the other chords will be determined by interpolation.

### 3.1.5. int-vert

Defines the domain of allowed vertical intervals (the numbers represent an interval as a number of semi-tones; e.g. : a diminished fifth = 6).

In the example shown as figure 1 in section 6, the first chord contains only major thirds, perfect fifths and major sevenths [Ex: 0 (4 4 7 11)] etc.

the major third (4) is given twice since the departure and arrival of the interpolation must have the same number of values.

The **filt-int** module (which is connected to the module **interval-interp**) is connected in turn by its input filtre, to a list within a module const. This list is used to forbid certain intervals from being included in the interpolated results (in this case : 3, or a minor third).

### 3.1.6. int-hori

- 
5. To drag with the mouse; place the pointer on the selected object, press the mouse button and move the mouse up or down without releasing the button. Release the button when the desired value is shown.

Defines the domain of allowed horizontal intervals. By default, only the interval between successive highest and lowest notes in each adjacent chord pair are considered on a pair, by pair, basis.

In the example shown as figure 1 in section 6, the highest and lowest voice formed by the sequence from chord 5 to 12 will contain only seconds and thirds, minor and major, perfect fourths and fifths, tritones and major sevenths.

Only the **expand-1st** module accepts the syntax '\_' (for example: 1\_7 for 1, 2, 3, 4, 5, 6 and 7).

It is also possible to control each voice formed by the succession of chords.

If the density of chords is both known and fixed for the entire sequence or for the part of the sequence to be constrained, the voices may be numbered from the bottom up: v 0, 1, 2, etc. Example for 10 chords with a density of 5 :

```
9 (v 0 (1 6 9) 1 (2 7).....5 (1 3 4 8))).
```

If the density of the chords is either unknown or variable, it will be possible to control only the *voix inférieure* (lowest voice), the *milieu* (middle) and the *supérieure* (upper) these voices will be designated i, m and s.

## int-vert-filt

Defines specific constraints in the domain of vertical intervals.

In addition to standard syntax Lisp [not, or, and], there is a special syntax for vertical intervals :

- when a number is preceded by ints, the corresponding interval will take into account over-projections;
- when a number is preceded by t, the corresponding interval will take into account octave doublings.

Finally, the system uses the standard syntax: '\*' (any quantity of any character, including nothing) and '?' (any single character).

In the example shown as figure 1 in section 6, throughout the entire sequence [Ex: 0 15], the following is forbidden : octaves in all their forms (direct, redoubled or over-projecting of other intervals), major seconds, perfect fifths and conjoint major sevenths regardless of what is above and below (in other words at any location within the chord).

```
[Ex: (not (or (ints 12 t) (* 2 2 *) (* 7 7 *) (* 11 11 *)))]
```

As a reminder of Lisp syntax, If not is replaced by and, the restrictions become obligatory; and if or is replaced by and, only a configuration where all four cases were combined would be forbidden (clearly impossible in this case).

## vs-filt-hori

Defines specific constraints in the domain of horizontal intervals for the highest voice, in French voix supérieure (vs).

See *int-vert-filt* (§ 3.1.6); however, for this constraint as well as for *vi-filt-hori* the syntaxes '\*/?' and 'ints/t' do not apply.

## vi-filt-hori

Defines specific constraints in the domain of horizontal intervals for the lowest voice, in French voix inférieure (vi).

See corresponding paragraph for the previous input *vs-filt-int*..

## vs-seuil-rep

Defines a threshold of repetition for the notes of the highest voice. In the example shown as figure 1 in section 6, a note must not be repeated in the highest voice of a chord sequence, until there have been seven different pitches represented, this takes into account octave doublings (in other words a C4 and a C5, for example, are considered as the same pitch).

[Ex: (0 15 (7 t))]

## **fixed-notes**

Defines in code Midi obligatory pitches which must be present in the chords.

In the example shown as figure 1 in section 6, A3, Db4 and C54 for the first chord.

[Ex: (57 61 72 (0))]

## **rep-par-dens**

Defines the number of repetitions of an interval that may be present within a chord, in function of that chords density.

In the example shown as figure 1 in section 6, if the chords density is 2 or 3, no interval repetition is allowed (any interval appears at most once); and if the chords density is 4, 5, 6 or 7 a single repetition is allowed (any interval appears at most twice).

[Ex: (0 15 ((2 1)(3 1)(4 2)(5 2)(6 2)(7 2)))]

## **renouv-haut**

Defines the number of different notes represented between a number of adjacent chords.

In the example shown as figure 1 in section 6, for the entire sequence, each pair of chords must have eight different pitches, this takes into account octave doublings.

[Ex: (0 15 (2 8 t))]

## **filt-pas-band**

This input is equivalent to *int-vert-filt* but allows the user to specify constraints in function of range and placement in the sequence.

In the example shown as figure 1 in section 6, for the entire sequence, it is forbidden to have a major seventh in the range below C4 :

[Ex: 0 15 (0 60 (not (\* 11 \*)))]

Also, it is required that the eighth and ninth chords will have a fourth as the second interval from the bottom and a major seventh anywhere above.

[Ex: 7 8 (and (? 5 \* 11 \*))]

## **int//forbid**

This input allows the restriction of parallel intervals between the highest and lowest voices of successive chords. Exceptionally, the number of times a number is given corresponds to the number of parallel intervals that are forbidden.

In the example shown as figure 1 in section 6, it is forbidden to have two parallel perfect fifths between the highest and lowest notes of two successive chord pairs .

[Ex: (0 15 (not (7 7)))].

## **nb-mouv//ok**

Defines the number of parallel movements allowed between the outlying voices of the chords. A single threshold value is designated.

In the example shown as figure 1 in section 6, for the first twelve chords, there may be no more than two consecutive parallel movement between the outer voices of the chords. For the last four, three are permitted.

[Ex: (0 11 (2) 12 15 (3))]

## **nb-dir=ok**

Defines the number of consecutive movements in the same direction that are allowed in the upper voice of each chord. A single threshold value is designated.

In the example shown as figure 1 in section 6, for the first twelve chords, there may be no more than three consecutive chords in which the upper voice moves in the same direction. For the last four, four are permitted.

[Ex: (0 11 (3) 12 15 (4))]

## **User-constraints**

Allows the user to define arbitrary constraints. This input may receive either a constraint or a list of constraints [Ex: Figure 3]. Programming these constraints is done with the module **constraint** which is described later in the manual.

## **Prev-Solutions**

Receives a list of lists specifying the initial domain for each chord. The sub-list at position *k* corresponds to the domain of the *k*-th chord. Each sub-list contains chords expressed as lists of midicents. If there are more variables (*n-accords*) than domains (length of the input list connected to *Prev-Solutions*), the last domain is used by default.

This module allows the use, as an initial domain, of the results from a previous calculation by the **harmonic-constraint** module.

## **paramètres**

This input is connected to the output of the **set-param** module which is described later in the manual.

Note on using variables in standard constraints :

The user may define, in Lisp or in PatchWork, a particular constraint; giving it a name that may be called for one of the modules (**const**, **interval-interp**, ...) connected to **harmonic-constraints**.

Note on figure 1

The **harmonic-constraints** module is connected to a **buffer** module and then to a **construct-sols** module. This configuration is used so that the complete set of calculated solutions may be stored in the **buffer** (by locking it after calculation is complete). The solutions are presented as a list:

(<Solution 1> <Solution 2> ... <Solution n>)

where each term <Solution *i*> is itself also a list, in the form :

( <value> ( <Note 1> <Intervals 1> ... <Note n> <Intervals n>))

Here, <value> is a number between 0 and 1 which gives a measure of value to each solution in relation to the relative weights of the various constraints. This mechanism of weights is explained above. The elements <Note *i*> <Intervals *i*> determine a chord. <Note *i*> is its bass note in code Midi and <Intervals *i*> are the intervals between consecutive notes.



The **construct-sols** module is used to transform this representation into a list of chords expressed in midicents. In the input <base-notes>, one enters in midicents the bass notes on which the solution is to be constructed. If one leaves the value 0, the solutions calculated by the module **harmonic-constraints** will be constructed without alteration.

The module **posn-match** allows the user to chose which solution is to be displayed or edited (0 = the first, etc.).

The final module of this menu, **part-sols** allows partial results to be recovered when a constraint resolution is interrupted (if, for example, the calculation is going on for too long). To recover the partial result, after having aborted the evaluation, simply connect **part-sols** to the input of the buffer.

## set-param



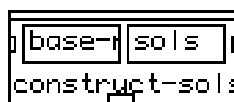
Syntax: **set-param** *maxtime* &optional *min-nec* *opt-fun* *test-fun* *maxval* *minval*

This module is to be connected to the optional input paramètres of the **harmonic-constraints** module. It allows the control of certain general parameter which control the system of exploration. *maxtime* is a positive whole number which indicates the maximum allowable calculation time, in minutes. When the time is finished, the system stops its exploration and returns the value that have been found up until that point.

*min-nec* must be a number between 0 and 1. This parameter is in relation to the weights given to the constraints. It indicates the minimum value that a solution may have. For example, if the value is 0.4, the weight of the strongest non-satisfied constraint in any returned solution is less than 0.6.

*opt-fun* *test-fun* *maxval* *minval* refer to the problem of finding a solution which at once satisfies a set of constraints and globally minimizes a given function. The description of these parameters goes beyond the scope of this document; it will be treated in a future users-manual.

## construct-sols



Syntax: **construct-sols** *base-n* *sols*

Receives in the input *sols* the output of an **harmonic-constraints** module (list of pairs [bass-note, intervals]) and outputs a list of chords in midicents. Each chord is transposed according to the values connected to the input *base-n*.

# part-sols



Syntax: **part-sols**

Returns as many objects as possible (formatted as pairs [bass-note, intervals]) that have satisfied all the constraints applied to an **harmonic-constraints** module. These objects are recovered after the **harmonic-constraints** module's calculation has been interrupted by an Abort, Break or by exceeding the time limit for the calculation (see the module **set-param** ).

## rhythm-constraints



Syntax: **rhythm-constraints** *n-solutions n-units units valid-divisions*

Another possible type of construction is a rhythmic structure, using the basic notions of a note (a duration that is not a rest), a unit (measure) and of groupings (succession of at least two individual durations, not separated by a silence). Note that this system understands only two types of values; durations and rests.

**Rhythm-constraints** has four inputs.

*n-solutions* Number of desired solutions

*n-units* Number of desired measures

*units* Defines the measures using standard PatchWork syntax

*valid-division* Defines the domain of acceptable durations; this takes into account the unit of reference selected in the input units.

For example : eighth note, triplet eighth note and quintuplet sixteenth note.

[Ex: (1/2, 1/3, 1/5)]

## construct-rhythm



Syntax : **construct-rhythm** *rhythm*

*rhythm* is connected to a list of lists of proportions. Each sub-list contains positive or negative rational numbers (negative numbers are used for rests). The module transforms each sub-list of proportions into a hierarchical structure representing symbolically the rhythm expressed by the proportions. This hierarchical structure is the input format for *beats* input of the standard PatchWork rhythmic notation editor: **RTM** .

## total-notes



Syntax: **total-notes** *list*

Calculates the total number of non-negative elements in the list *list*.. This module is used to calculate the number of notes in a sequence of proportions.

## dist-into-units



Syntax: **dist-into-units** *units list*

Receives in its input *list* the output of the **rhythm-constraints** module. It separates the elements (durational proportions) of this list into sub-lists so that the sum the elements will correspond to the values given in the input *units* .

## groups-per-units



Syntax : **groups-per-unit** *groups ranges*

Defines the number of groupings per measure

## notes-per-units



Syntax : **notes-per-unit** *groupings ranges*

Defines the total number of notes per measure

## notes-per-group



Syntax : **notes-per-group** *list max-notes*

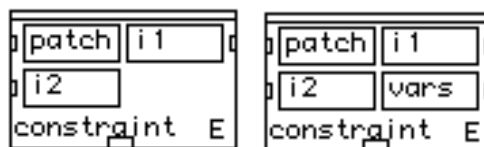
Defines the number of notes per grouping

These three modules function similarly to the module **interval-interp** in **harmonic-constraints**. Take for example the module **notes-per-units**, in order to have a first measure with a minimum of four notes and a maximum of seven; then in the seventh and eighth measures a minimum of 12 notes and a maximum of 18. An interpolation will be performed to determine the minimum and maximum for the intermediary measures.

[Ex: (0 (4 7) 6 (12 18) 7 (12 18))]

Memory buffers (*buffer*) are placed at each step allowing the conservation of whatever previously calculated results one wants to preserve before re-launching new calculations.

## constraint<sup>6</sup>



Syntax: **constraint** *patch i1 i2 &optional vars*

---

6. This module requires the possession of Common Lisp, in addition to a PatchWork image containing the compiler.

Defines a constraint which will be applied to one or more elements of an implied sequence. An implied sequence [see the module **solve**] is a list of variables where each variable is represented by the totality of its possible instantiations (the domain). The constraint is a Lisp or PatchWork predicate which test values from the elements of the corresponding domain of variables.

The input *patch* is connected to a patch which returns a function (the predicate) with a single argument. While researching solutions [see the module **solve**] that argument [see the module **parameter** in figure 5] of this function is instantiated as an object (structure) with five fields (*slots*):

slot *i1* : Position of one of the variables to be tested in the sequence

slot *i2* : Position of another of the variables to be tested in the sequence. It must always be true that :  $i1 \leq i2$

slot *item1* : current value of the variable in position *i1*

slot *item2* : current value of the variable in position *i2*

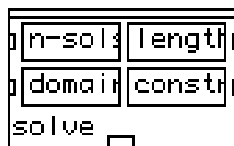
slot *other* : list of current values of the variables in positions (in order) *i1*-1, *i1*-2, *i1*-3, ..., 1

The patch defining the predicate accesses these fields through the intermediary of the PatchWork module **get-slot** [Ex: figure 5].

The module **constraint** has three other inputs *i1*, *i2*, *vars*. The first two, *i1* and *i2*, are indexes which specify the positions in the sequence which are to be tested by the constraint. The input *vars* is used to select one of the three possible interpretations of indexes *i1* and *i2*. The option *fix* interprets the indexes as being absolute positions in the sequence. If *i1* is equal to 2 and *i2* is equal to 5, for example, will only be used to test the domains of positions 2 and 5 in the sequence. If the option *seq* is selected, it is the distance between the indexes which is considered. The option *all* indicates that the constraint should be used for all pairs of positions in the sequence (thus the values of *i1* and *i2* are ignored).

The example in figure 5 models the problem of positioning eight queens in a chess table in such a way as that no two of them may attack each other. Thus one must find a sequence of eight elements, the element in position *i* of the sequence gives the column number where the single queen on column *i* is located. The single constraint, which applies to all possible line pairs (*vars* is set to *all*), is a predicate that returns T only when two queens are not on the same diagonal (**g-oper** module on the left) or on the same column (**g-oper** module on the right). The **funarg** module, explained below, is used to transform the patch it receives as input into an equivalent Lisp function (the predicate) which may be connected to the input patch of the module **constraint**.

## solve<sup>7</sup>



Syntax: **solve** *n-sols length domain constraints*

---

7. See note 6.

Explores the *domain* in order to calculate *n-sols* sequences with a length of *length* that satisfy the constraints of *constraints*.

The input *domain* must be a list. Each element of the list is a set which specifies the possibilities of instantiation at that position in the sequence. This set may be presented as either a list or a series (list with a delayed evaluation). *constraints* is either an object constraint (the output from a **constraint** module) or a list of object constraints [Ex: figure 5].

## funarg<sup>8</sup>



Syntax: **funarg** *patch* &optional *name*

Constructs a LISP function equivalent to the patch connected to the input *patch*.

The arguments of this function are the modules found inside the patch. *name* is the name (optional) of the calculated.

Restrictions: the patch must not contain uncompiled abstractions or **absin** modules (for more see the PatchWork Reference Manual).

## parameter<sup>9</sup>



Syntax: **parameter** *order*

Defines a parameter in a patch connected to the module **funarg**.

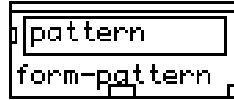
The input *order* gives the position of the argument in the list of arguments fro the function calculated by **funarg**. Within a single patch connected to a **funarg** module, each argument *order* must have a different number.

---

8. See note 6.

9. See note 6.

# form-pattern



Syntax: **form-pattern** *pattern*

Constructs a function with a single argument which tests the correspondence between its input and the given pattern.

The input *pattern* is either a list representing a regular expression, or a Boolean formula containing patterns as its arguments. Regular expression are lists of some sort which may have the symbols '\*' and '?'. The symbol '?' corresponds to any single element while the symbol '\*' corresponds to any sequence of zero or more elements. For example, the pattern (\* 4 7 \* 9 ? 11 \*) corresponds to (1 3 2 4 7 9 4 11) and (4 7 5 3 9 10 11 17) but does not correspond to (1 3 2 4 7 9 11) or (4 2 7 5 3 9 4 11). The expression of patterns may be within Boolean formulas. For example:

(and (not (\* 4 7 \*)) (or (\* 4 ? 11 \*) (\* 7 ? 11 \*)))

## interval-interp



Syntax: **interval-interp** *exp* &optional *curve*

*exp* is a list with the form (<index1> <liste1> <index2> <liste2> ... <indexn> <list n>) where the indexes represent positions in a sequence. The module returns a list with a length of <indexn> + 1, containing each <liste i> at the position given by <index i> and for the intermediate positions, between <index i-1> and <index i>, interpolations between <liste i-1> and <liste i>. For example :

*exp* = (0 (4 7) 10 (7 11)) gives

((4 7) (4 7) (5 8) (5 8) (5 9) (6 9) (6 9) (6 10) (6 10) (7 11) (7 11)).

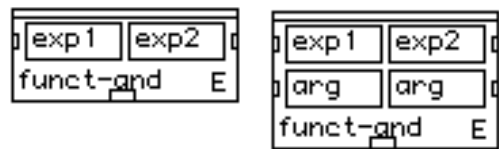
The optional input *curve* specifies the type of interpolation:

*curve* = 1 : linear,

: convex,

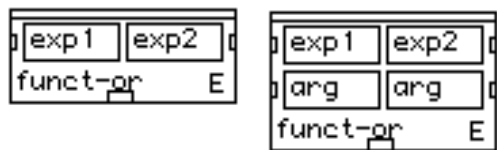
: concave

# funct-and



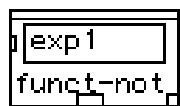
Syntax: **funct-and** *exp1 exp2 &rest exp\**  
Calculates the conjunction of the expressions *exp1 exp2 ...* In the above illustration the module on the left has two optional inputs *arg* .

# funct-or



Syntax: **funct-or** *exp1 exp2 &rest exp\**  
Calculates the disjunction of the expressions *exp1 exp2 ...* In the above illustration the module on the left has two optional inputs *arg* .

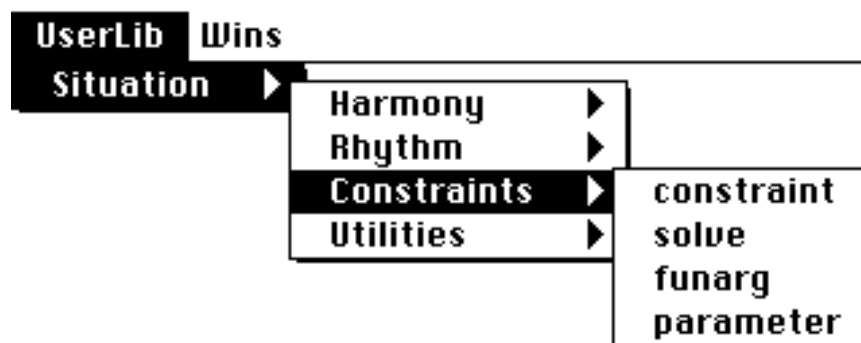
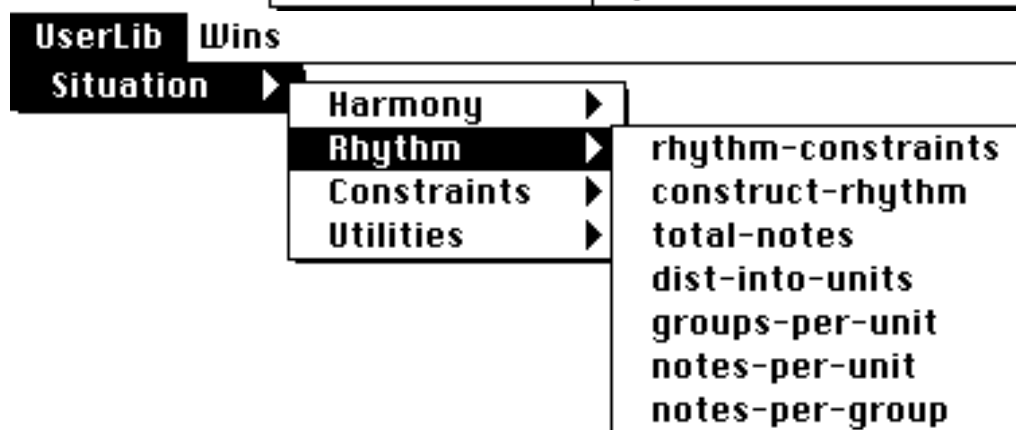
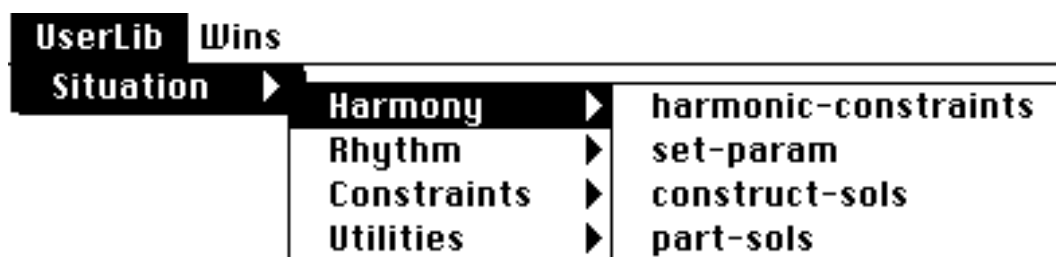
# funct-not

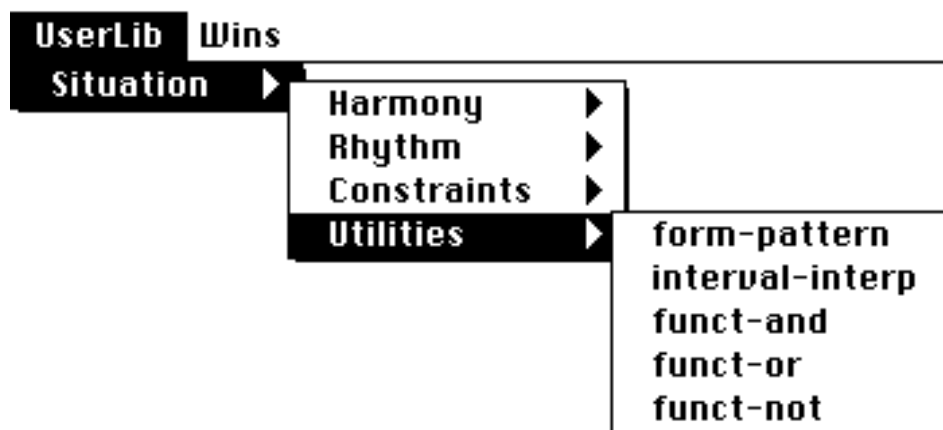


Syntax: **funct-not** *exp1*  
Calculates the negation of *exp1*.

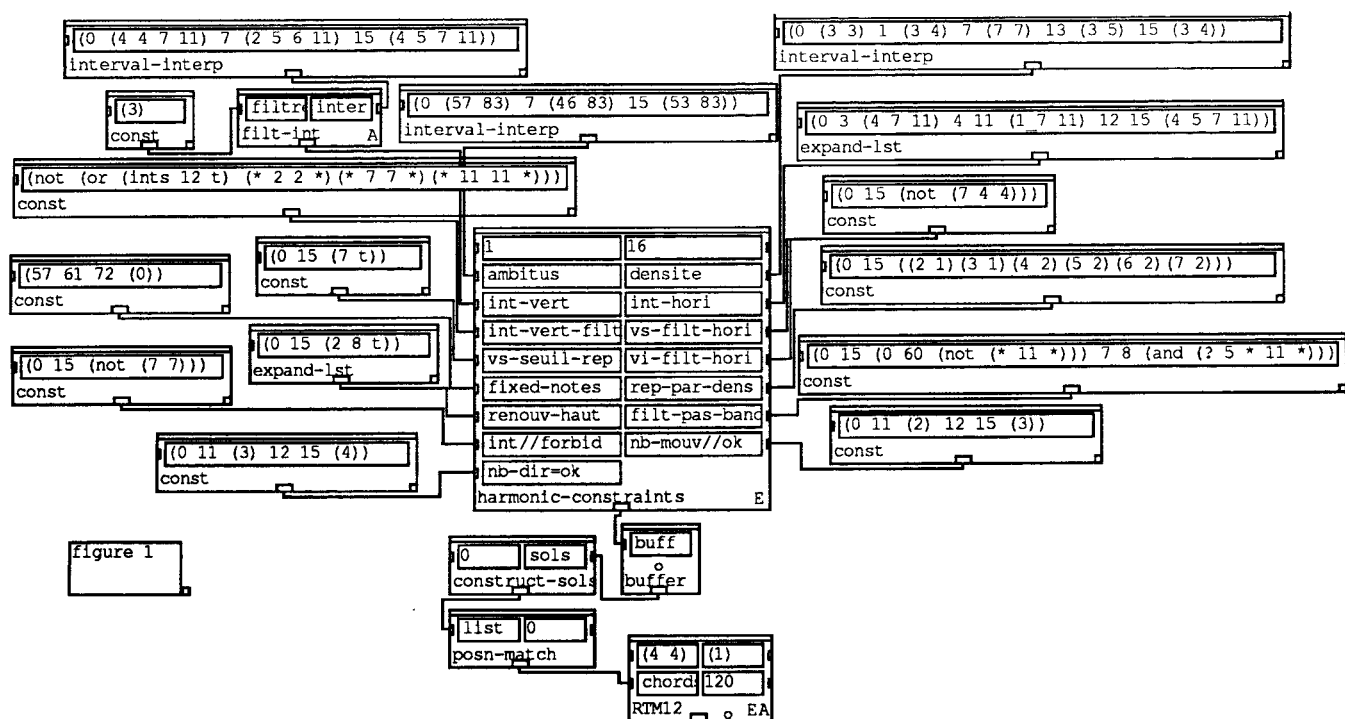


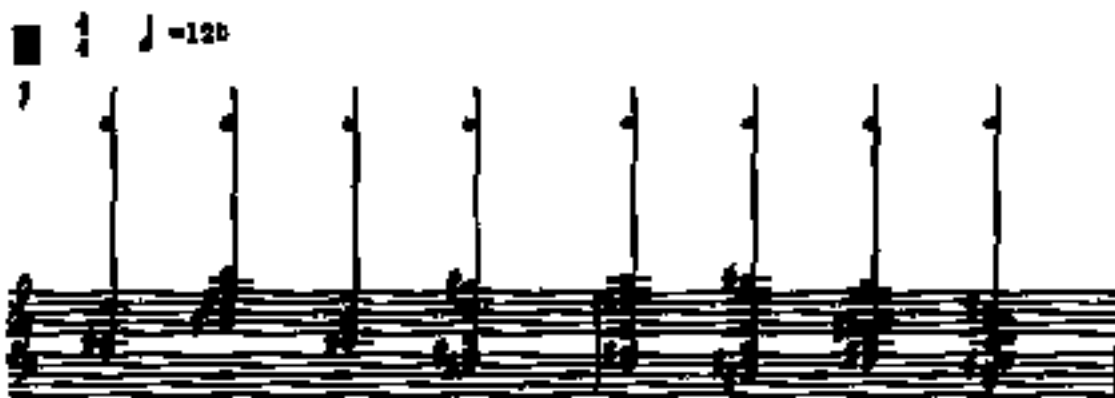
## 4 Menus of the Situation library

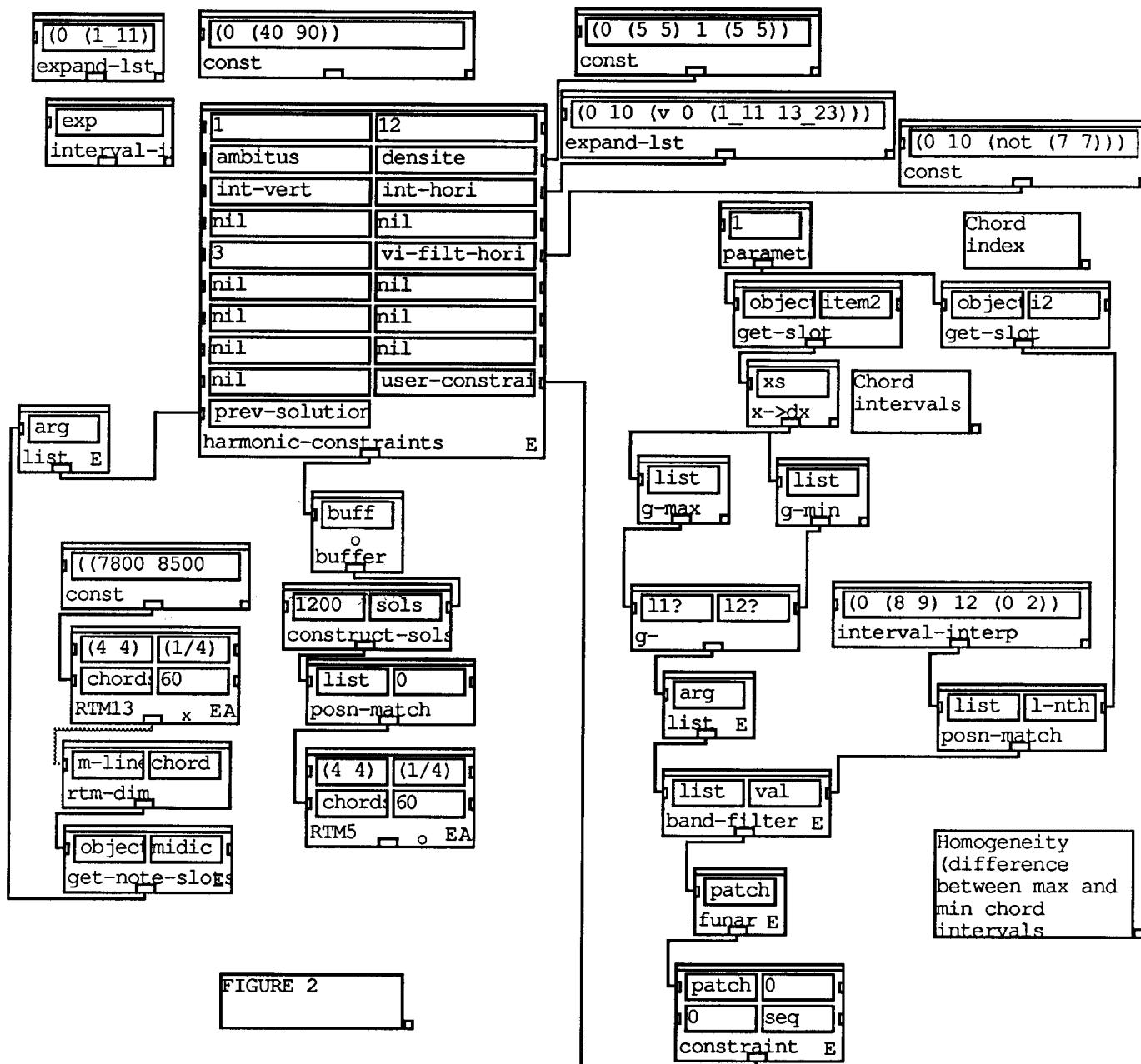




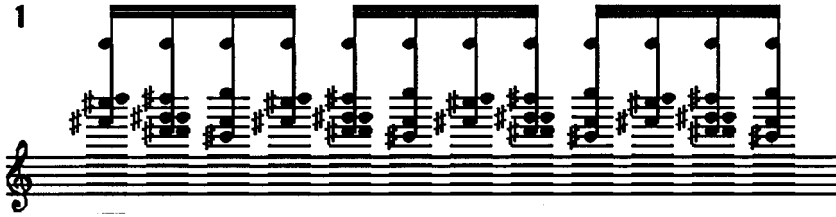
# 5 A selection of examples



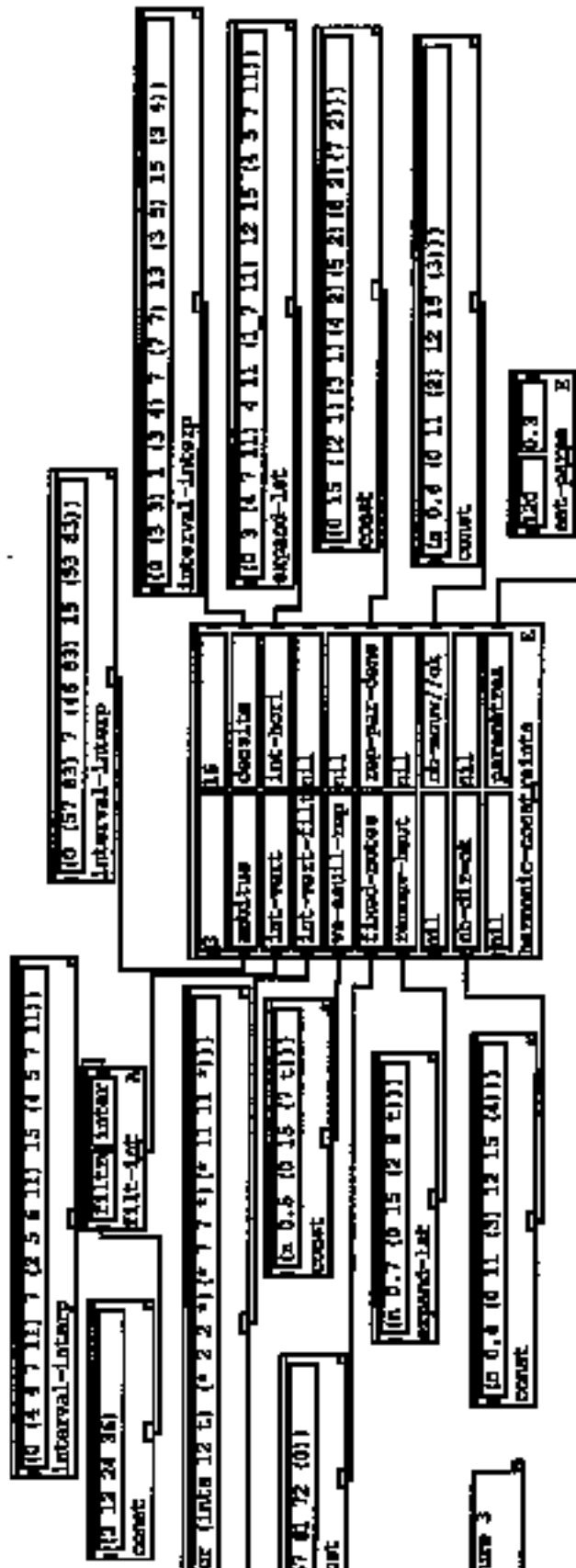




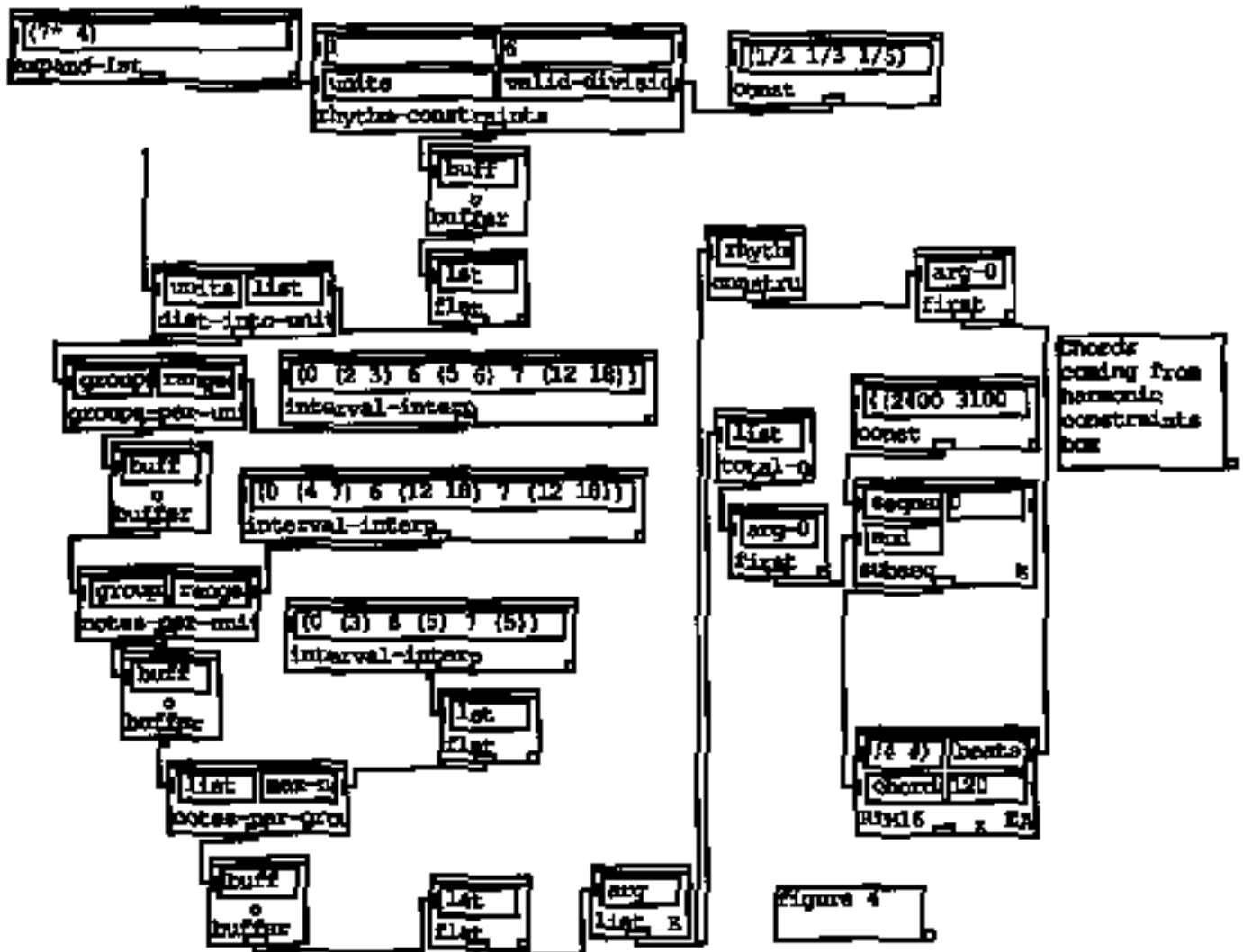
■  $\frac{3}{4}$  ♩ = 60











4/4  $\text{♩} = 120$   
 1

The first staff shows a sequence of chords, each with a 5th interval marked above the notes. The second staff continues this pattern, introducing a 3rd interval in some chords. The third staff shows a more complex arrangement, including a 3/4 time signature change and multiple 3rd intervals.

Example of figure 4 (RTM16)

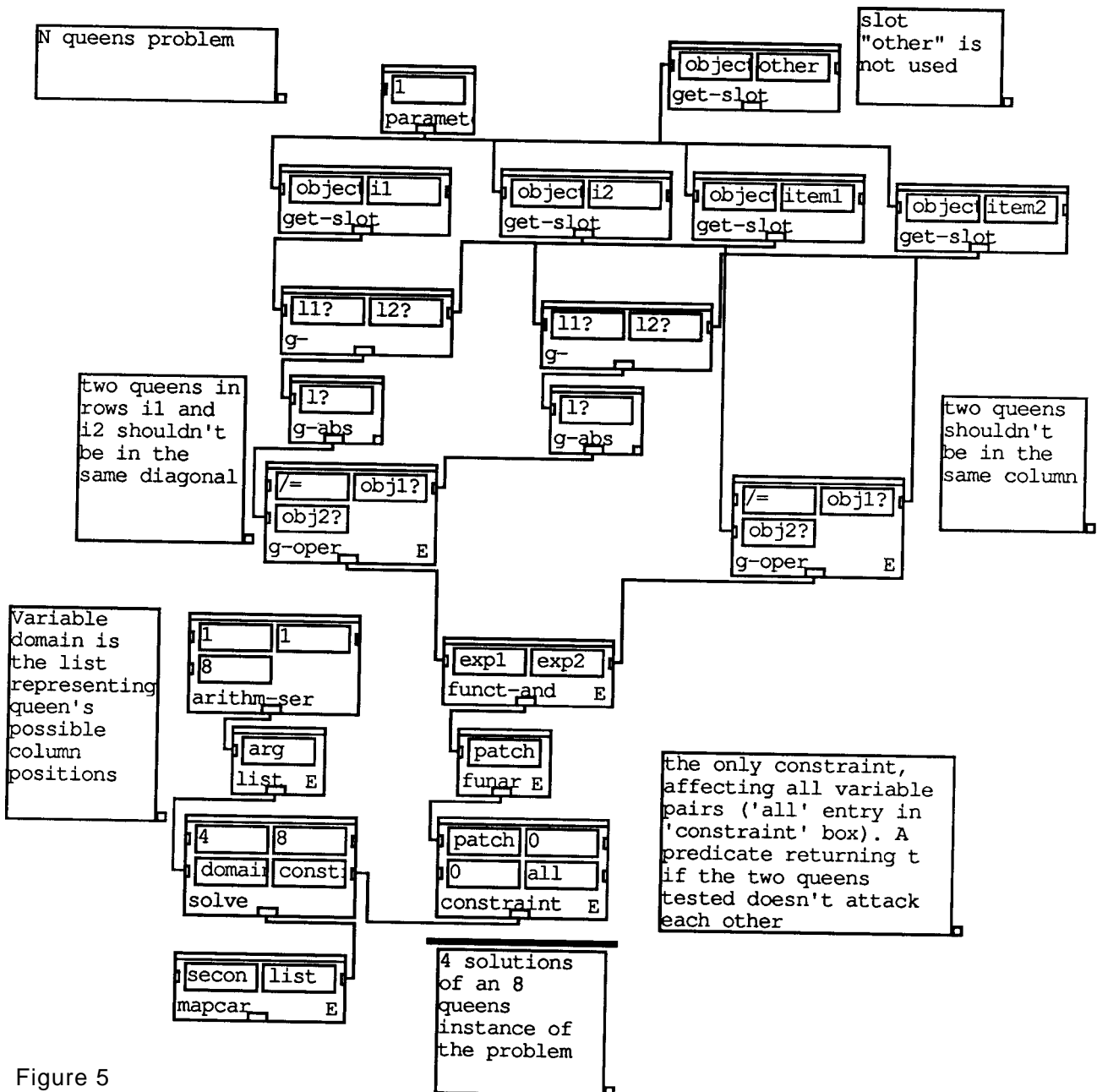


Figure 5



4/4 ♩ -120

1

This system contains measures 1 through 3 of a musical piece. It features a piano part with a complex, dense texture of chords and a vocal line with a melodic contour that rises and then falls. The tempo is marked as 120 beats per minute.

4/4 ♩ -120

2

This system contains measures 4 through 6 of the musical piece. The piano part continues with dense chordal textures, and the vocal line concludes with a final melodic phrase. The tempo remains at 120 beats per minute.





# Index

## B

Bonnet A. 2

## C

constraint? 20  
Constraints 11  
construct-rhythm 18  
construct-sols 17

## D

dist-into-units 19  
Duthen J. 2

## F

filt-pas-band 15  
Fineberg J. 2  
fixed-notes 15  
form-pattern 23  
funarg? 22  
funct-and 24  
funct-not 24  
funct-or 24

## G

groups-per-units 19

## H

Harmonic-constraints 12  
Harmony 11

## I

int//forbid 15  
interval-interp 23  
int-hori 13  
int-vert 13  
int-vert-filt 14

## L

Laurson M. 2

## N

nb-dir=ok 16  
notes-per-group 20  
notes-per-units 20

## P

parameter? 22  
paramètres 16  
part-sols 18

Prev-Solutions 16

## R

renouv-haut 15  
rep-par-dens 15  
Rhythm 11  
rhythm-constraints 18  
Rueda C. 2

## S

set-param 17, 18  
solve? 21

## T

total-notes 19

## U

Utilities 11