Ircam documentation

- Research reports
- Musical works
- **Software**

PatchWork

PWConstraints

First English Edition, October 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 Mikael Laurson, 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 PWConstraints library was conceived and programmed by Mikael Laurson.

First English edition of the documentation, October 1996.

This documentation corresponds to version 1.0 of the library.

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

Ircam

1, place Igor-Stravinsky

F-75004 Paris

Tel. 01 44 78 49 62

Fax 01 44 78 47 44

E-mail ircam-doc@ircam.fr

IRCAM Users' group

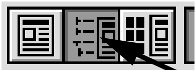
The use of this software and its documentation is restricted to members of the Ircam software users' group. For any supplementary information, contact:

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

Send comments or suggestions to the editor:

E-mail: 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

1. Introduction 1

2. About this documentation 2

3. PWConstraints Programming 3
   3.1 Examples Folder 3
   3.2 PW Extensions..4

4. PWConstraints Reference 8
   4.1 PWCs Menu 9
      4.1.1 PMC Box 9
      4.1.2 score-PMC Box 10
   4.2 Utilities Menu..11
   4.3 Debug Menu...11
   4.4 PC-set-theory Menu 12
      4.4.1 SC-name midis..13
      4.4.2 SC+off midis...13
      4.4.3 SCs/card card...13
      4.4.4 SC-info function SC-name.......................13
      4.4.5 sub/supersets sc-name card 14
      4.4.6 all-subs sc-name 14
   4.5 Music Analysis Menu 14
   4.6 DOC Menu 16
      4.6.1 PMC Sub Menu 16
      4.6.2 Lisp Sub Menu 17
      4.6.3 PC-set-theory Sub Menu.........................19
      4.6.4 Example Functions Sub Menu 20
      4.6.5 Score-PMC Sub Menu 22
      4.6.6 Misc Sub Menu..25
      4.6.7 Counterpoint Sub Menu 27
      4.6.8 Groups Sub Menu 29

5. Appendix 30
   5.1 Introduction 30
      5.1.1 PWConstraints in Perspective 31
         5.1.1.1 Background 31
         5.1.1.2 PWConstraints as a Constraint Satisfaction Problem Language 32
         5.1.1.3 Related Work.................................... 32
      5.1.2 Defining a Search-Space 33
         5.1.2.1 Random Ordering............................. 35
         5.1.2.2 Preference Ordering......................... 35
         5.1.2.3 Constraining Individual Notes............ 35
         5.1.2.4 Moulding a Search-Space 36
      5.1.3 Writing Rules 36
         5.1.3.1 Partial Solutions 37
         5.1.3.2 Pattern-Matching 40
         5.1.3.3 Pattern-Matching Syntax 40
         5.1.3.4 PWConstraints Pattern-Matching Compared with Other Pattern-Matchers 43
         5.1.3.5 Rule Examples 43
      5.1.4 Classical Constraint Examples................44
         5.1.4.1 A Cartesian Product Example 45
         5.1.4.2 Subset Indices.................................. 45
         5.1.4.3 All Permutations 46
      5.1.5 Musical Examples 47
         5.1.5.1 Constraining Chords......................... 47
         5.1.5.2 A PC-Set-Theoretical Example 49
         5.1.5.3 Subsets 54
         5.1.5.4 Statistical Distribution and Automatic Rules 56
         5.1.5.5 The "At Least" Property.................... 60
         5.1.5.6 Splitter 61
   5.2 Implementation details and extensions of PWConstraints...65
      5.2.1 Implementation of the Search-Engine 65
         5.2.1.1 Search-Variable................................ 65
         5.2.1.2 Search-Engine.................................. 66
      5.2.2 Pattern-Matching Compiler......................70
      5.2.3 Objects in a Partial Solution 73
      5.2.4 Heuristic Rules 74
      5.2.5 SCREAMER Interface 76
      5.2.6 PW Interface...77
      5.2.7 Efficiency Issues - Forward Checking 79
         5.2.7.1 Graph Representation of a CSP........ 79
         5.2.7.2 Choice of Consistency Inference Techniques 81
         5.2.7.3 N-arity Constraints and Forward Checking 82
         5.2.7.4 Rule Translator................................. 83
   5.3 First Case Study: Counterpoint Rules................86
      5.3.1 PWConstraints Compared to CHORAL...87
      5.3.2 Score Representation.............................87
         5.3.2.1 Score-Sort 88
         5.3.2.2 Restoring Musical Information........... 90
         5.3.2.3 "Musical Map".................................. 90
         5.3.2.4 Melodic-Context 92
         5.3.2.5 Harmonic-Context 92
         5.3.2.6 Harmonic-Slice 93
         5.3.2.7 Metric-Context 95
         5.3.2.8 Completing the Score Representation Scheme 97
         5.3.2.9 Utility Functions and Macros 100
         5.3.2.10 Melodic Pattern-Matching.............. 102
         5.3.2.11 Non-Attack Search-Variables 102
      5.3.3 Translating Counterpoint Rules 103

5.3.3.1 Melodic Rules................................. 104
5.3.3.2 Voice Leading Rules 107
5.3.3.3 Harmonic Rules.............................. 112
5.3.4 Defining Search-Space for a Polyphonic Search Problem120
5.3.4.1 Search-Space Score 120
5.3.4.2 Constraining Notes.......................... 121
5.3.5 PW Interface..122
6. Bibliography 124
. Index 126

1 Introduction

PWConstraints is a rule-based programming language used to solve complex musical problems. These problems typically comprise situations where the user wants to describe the end result from many different points of view. Writing counterpoint is a good example. The end result is described with the help of rules controlling melodic lines, harmony, voice-leading, etc.

When using a procedural programming language, such as C or Pascal, or a functional language, like Lisp or PW, the user has to solve a problem in a stepwise manner. In many cases this approach is an adequate one, but for many types of problem it may lead to programs that are difficult to design or understand. Descriptive or declarative languages, like Prolog, offer an alternative way to look at this problem: instead of trying to solve a problem step-by-step, the user describes a possible result with the help of a set of rules.

PWConstraints can be thought of as a descriptive language. When using it we do not formulate stepwise algorithms, but define first a *search-space*. Then we search for solutions with a *backtrack search-engine* that produces systematically potential results. In a typical case we are not interested in all possible results, but filter (or, rather, *constrain*) these with the help of rules describing an acceptable solution.

2 About this documentation

A detailed discussion on constraint programming and PWConstraints is found in the fifth chapter of my thesis (Laurson 96). The Appendix of this documentation gives the first three sections of this chapter (called henceforth "thesis text"). The thesis text is quite long and technical and it is not necessary for all users to read everything in order to be able to use PWConstraints effectively.

Probably section 5.1 should be read by all users, as it introduces the basic concepts of PWConstraints. Also it covers a detailed discussion of many concrete examples (ranging from simple to relatively complex).

Section 5.2 is mostly technical (especially sections 5.2.1, 5.2.2 and 5.2.5 can be skipped as they mostly describe how PWConstraints is implemented). Sections 5.2.3, 5.2.4 and 5.2.6, however, give useful information of heuristic rules, PW interface, etc. Section 5.2.7, dealing with forward checking, is quite arduous. It describes some of the efficiency problems that one typically encounters when using the system. Therefor it is useful for people who want to use PWConstraints more intensely.

Section 5.3 discusses how to write counterpoint rules in PWConstraints and should be read by anyone (especially sections 5.3.1-5.3.2) who wants to use the `score-PMC` box (the `score-PMC` box will be explained later in the documentation).

Besides this introductory text and the Appendix, the documentation also includes a short text about programming, examples and extensions (PWConstraints Programming) and a reference manual (PWConstraints Reference).

The section PWConstraints Programming will concentrate mostly on the aspects that are not covered in my thesis, like giving some practical programming hints, describing various PW extensions, etc.

PWConstraints Reference, in turn, describes the menu structure of PWConstraints and the PW-boxes, Lisp functions and macros included with this package.

3 PWConstraints Programming

PMC and score-PMC

PWConstraints provides two primary functions, PMC and score-PMC. Both functions are described in detail in my thesis text (sections 5.1, 5.2 and 5.3). As it is important to understand the difference between them, we recapitulate below their main features and differences.

PMC is a general purpose tool and it can solve both music-related and non-music-related constraint-based problems. PMC can be used either directly in a text-editor or inside PW (PW provides a graphical interface for constraint-based programming).

Compared with PMC, score-PMC is more specialised in solving musical problems. It requires a prepared rhythmic structure (a polyphonic PW RTM-editor) as input and aims at filling the input score with pitch information. score-PMC can be used only inside PW (i.e. the user has to work with a PW-patch). The score-PMC rules are different from those of PMC as the variable names mentioned in the pattern-matching part and the special variables l and rl refer to *objects* not to values (like PMC rules do). For instance the following PMC rule:

```
(* ?1 ?2 ?3 (?if (eq-SC? '(3-11a 3-11b) ?1 ?2 ?3))
  "only minor/major 3-member SCs")
```

has to be translated to the following score-PMC rule:

```
(* ?1 ?2 ?3 (?if (eq-SC? '(3-11a 3-11b) (m ?1) (m ?2) (m ?3)))
  "only minor/major 3-member SCs")
```

where we extract the midi-values (using the macro m) from the objects ?1, ?2 and ?3.

3.1 Examples Folder

The PWConstraints package includes a collection of examples found in the "Examples" folder. It contains a folder called "PMC-examples" that in turn has two folders called "Text examples" and "Patches". The former consists of text-based examples, while the latter contains PW patches. The text-based examples deal with various general constraint-based problems ("Basic-exs.lisp") and with the examples discussed in my thesis ("Section-5.1-exs.lisp"). There are also typical combinatorial problems in the files "All-interval-series.lisp" and "All-interval-chords.lisp".

The "Patches" folder, in turn, consists of patches that demonstrate how the PMC box is used inside PW. Running these examples is straightforward: just evaluate the PMC expression or the PMC box (sometimes it is necessary to evaluate some global variables or lisp help functions found in the same file before running the PMC function).

The "Examples" folder contains also `score-PMC` box examples ("`score-PMC`-examples"). As these typically contain already 10-20 rules it is often convenient to use for the rules the text-win module. The text-win box allows the rules to be edited, saved and loaded in a modular fashion. I normally put each `score-PMC` example in a separate folder that contains besides the patch-file, text-files for ordinary rules, forward checking rules, etc. For instance the example folder "2-part-canon" contains the patch-file "2-part-canon.pw" and two text-files: "2-part-canon-rules.lisp" (for the `rules` input) and "2-part-canon-fwc-rules.lisp" for the fwc-rules input). This example is run in four steps: (1) open the patch-file; (2) open the text-file for the `rules` input (use the popup-menu of the text-win box to open the file); (3) open the text-file for the `fwc-rules` input; and finally (4) evaluate the `score-PMC` box. After the search is finished (this can take a while, as the example is quite complex), the `score-PMC` box draws directly the solution into the polyphonic RTM-editor that is connected to the first input.

The music analysis examples (also found in the "Examples" folder) are run in a similar fashion: first open the patch, then the text-file containing the music-analysis rules and finally evaluate the music analysis-box. The `MA-PMC1` box prints messages into the Listener window, whereas the `MA-PMC2` box draws its result directly into the input score (these boxes are described in the Reference part of this manual). In the latter case it is necessary to open the RTM-editor in order to see the analysis information.

Note that the PWConstraints text-files should be in the package "PWCS". This means that first line of a text-file should be either:

`(in-package :pwcs)` or `(in-package "PWCS")`.

3.2 PW Extensions

The PWConstraints package adds to the RTM-editor the possibility to group any continuous succession of chords (these selections are called henceforth "groups"). After loading PWConstraints the RTM-editor main-menu bar has a new menu called "Groups" containing the following menu-items:

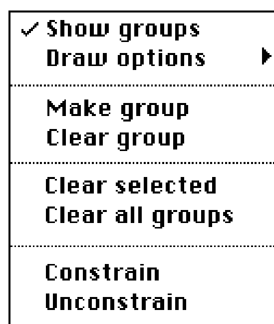


Fig.1: "Groups" menu.

The "Show groups" menu is a flag indicating whether the groups are displayed or not. The "Draw options" menu has two sub menus controlling the way the grouping information is drawn (we will explain these menu-items later).

The "Make group" menu allows the user to make groups. Before selecting this menu one must make a selection in the RTM-editor. The grouping scheme requires that the selected items must be at the chord leaves of the beat structure (i.e. select only the dotted-line rectangles that are drawn at the lowest level). For instance, in figure 2 below the user has selected a range of three chords having the notes C4, B4 and F#4. (This selection is accomplished by shift-clicking the end-points of the selection.) The selection is indicated by the black rectangles (only the first and the last rectangle of the selection are drawn black).

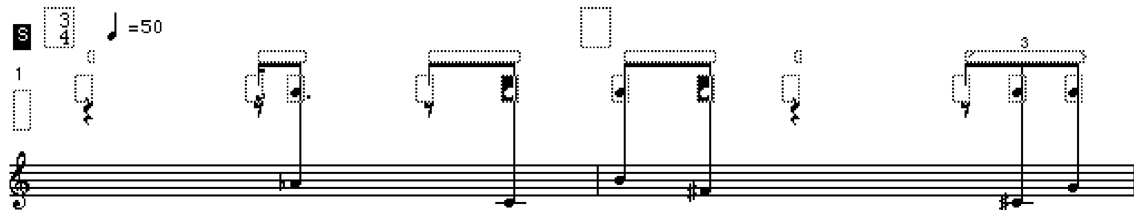


Fig.2: The user selects a range of chords.

After this the user selects the "Make group" menu (or types the character "g" from the keyboard). A dialogue is opened asking for the name of the group (figure 3):

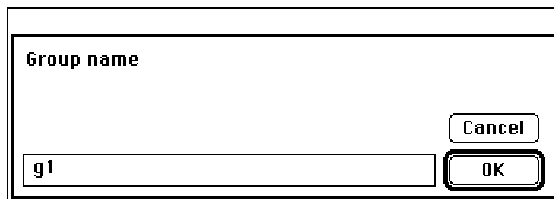


Fig.3: The group name dialogue.

Finally figure 4 shows the new group (named "g1") drawn below the selected chords:

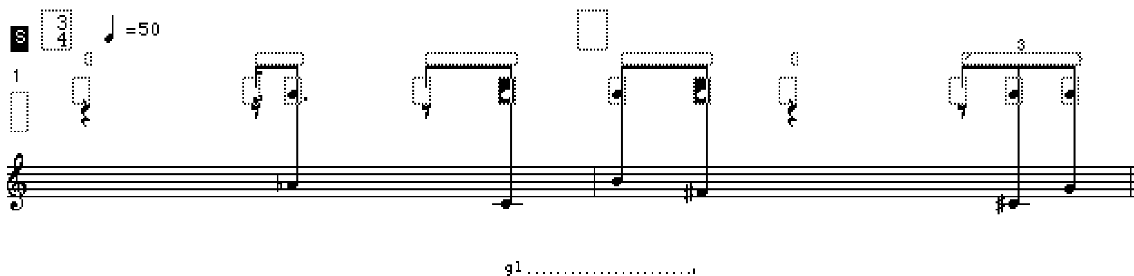


Fig.4: The score showing the group "g1".

To group isolated chords (i.e. to make a group containing only one chord), simply select the leaf rectangle of the respective chord and call the "Make group" menu.

The "Clear group" menu allows the user to clear a given named group. As in the previous example, select first the group (it is enough to select only the first chord of the group) and select the "Clear group" menu. A dialogue will appear asking the name of the group to be cleared. For instance, to clear the "g1" group in figure 4, select the first chord of the "g1" group, select the "Clear group" menu and type in the dialogue the string "g1".

The "Clear selected" menu is similar to the "Clear group" menu except it allows to clear all groups that are found within the selection. No dialogue will appear in this case as this menu clears all selected groups regardless of their names.

The "Clear all groups" menu clears all groups found in the score. This menu does not require any selection, as all groups will be cleared regardless of their position and name.

The "Constrain menu" is used to "constrain" a range of selected chords (the pitches of the constrained chords will be fixed during search). For instance, constraining the three selected chords of figure 2, will produce the following result (figure 5):

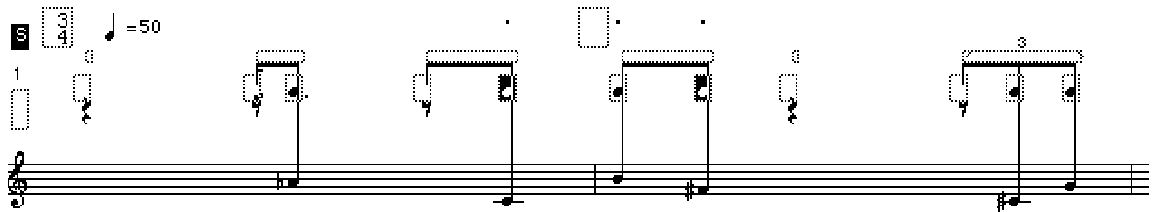


Fig.5: The user constrains a group of chords.

The constrained chords are indicated by the character `"'"` drawn above the chord.

The "Unconstrain menu", in turn, is used to remove the constraints from the selected chords.

The "Draw options" menu that was already mentioned above contains two sub menus called "Show two group names" and "Constant line drawing". The former is a flag indicating whether or not the group name is drawn twice (in the examples above this flag has been `nil`, i.e. the group name is drawn only once). If the flag is `true` then the name is repeated at the end of the dotted line showing the extent of the group (figure 6):

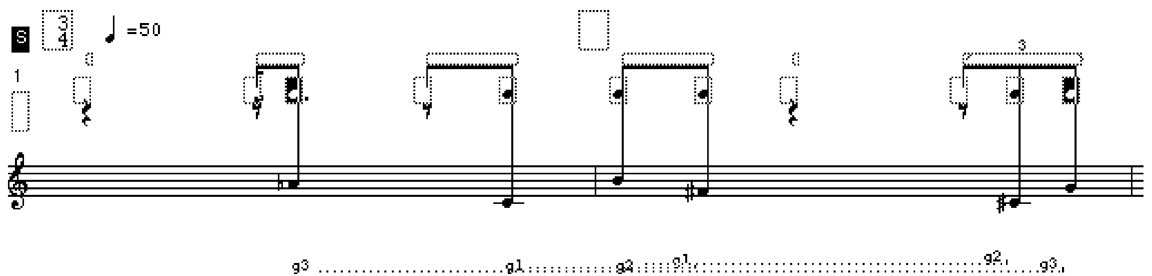


Fig.6: Overlapping groups.

This feature can be useful when different groups are overlapping (like in figure 6) and it becomes more difficult to identify where a given group ends.

Figure 6 shows also when the latter sub menu is useful. "Constant line drawing" indicates whether the dotted group lines are drawn at a constant y-position (the default case). If, however, this flag is `nil` then the algorithm draws the dotted lines with a slight offset added to the normal y-position. This makes it sometimes easier to distinguish the different groupings in a crowded situation (figure 6).

4 PWConstraints Reference

This part of the documentation serves mainly as a reference manual to the menus, PW-boxes, functions and macros used by PWConstraints. A more detailed discussion on constraint programming and PWConstraints is found in the Appendix.

Figure 1 below shows the menu structure of PWConstraints. We will next discuss each main menu and its sub menus in turn.

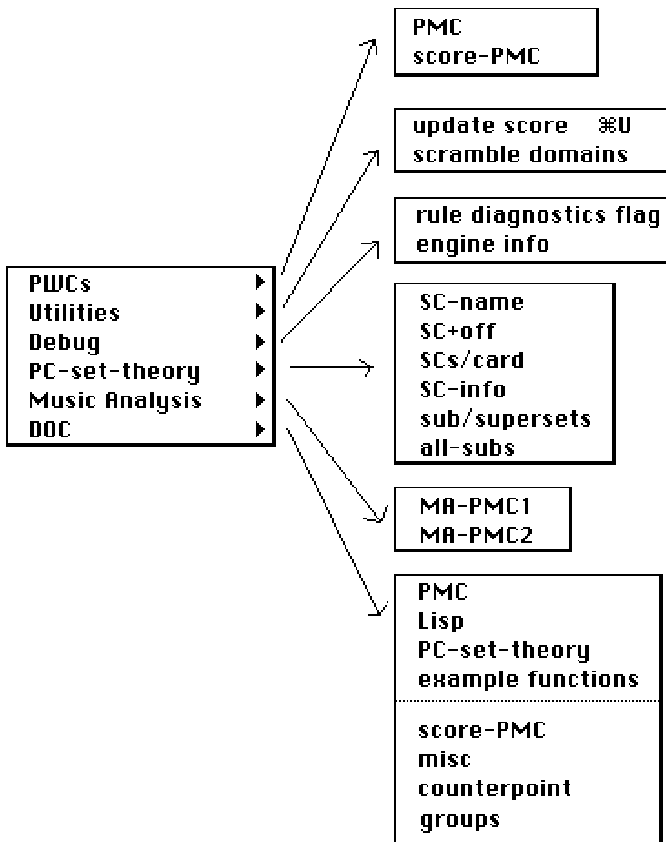


Fig.1: The PWConstraints main menu with its sub menus.

4.1 PWCs Menu

Fig.2: PWCs menu.

The "PWCs" menu (figure 2) creates the two main PW-boxes used by PWConstraints: `PMC` and `score-PMC`. `PMC` is discussed in detail in the Appendix (sections 5.1 and 5.2) and `score-PMC` in section 5.3.

4.1.1 PMC Box

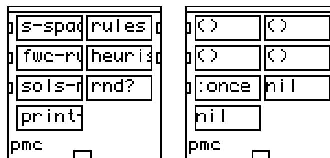


Fig.3: `PMC` box.

```
patch-work::pmc
  s-space rules
  fwc-rules heuristic-rules
  sols-mode rnd?
  print-fl
```

`PMC` (figure 3) creates a search-engine and starts the search. After the search is completed, `PMC` returns a list of solutions. The solutions should satisfy the constraints given by the user. For more details, refer to the Appendix (especially sections 5.1 and 5.2).

`PMC` has the following arguments:

- `s-space`	a list of domains for each search-variable
- `rules`	a list of rules (ordinary PWConstraints rules)
- `fwc-rules`	a list of forward-checking rules
- `heuristic-rules`	a list of heuristic rules
- `sols-mode`	indicates the number of solutions required: `:once` (the default case, one solution) or `:all` (all solutions). `sols-mode` can also be a positive integer giving the number of desired solutions.
- `rnd?`	a flag indicating whether or not the search- space is randomly reordered (by default `rnd?` is `nil`).

- print-fl a flag, if true then the index of the current search-variable is printed in the Listener window indicating how far the search has proceeded.

4.1.2 score-PMC Box

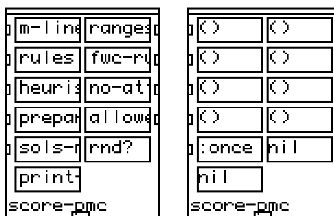


Fig.4: score-PMC box.

```
patch-work::score-PMC
  m-lines ranges
  rules fwc-rules
  heuristic-rules no-attack-rules
  prepare-fns+args allowed-pcs
  sols-mode rnd?
  print-fl
```

score-PMC (figure 4) reads its inputs, creates a search-engine and starts the search. After the search is completed, score-PMC updates the input score. The solution should satisfy the constraints given by the user. For more details, refer to the Appendix (section 5.3).

score-PMC has the following arguments:

- m-lines a list of measure-lines, defining the input score.
- ranges either a simple range list, a list of lists of ranges or a list of measure-lines. In the latter case the argument is given as a search-space score.
- rules a list of rules
- heuristic-rules a list of heuristic rules.
- non-attack-rules a list of non-attack rules.
- prepare-fns+args a list of user-definable preparation functions that are called before the search starts. They allow the user to customise the search problem.
- allowed-pcs a list of allowed pitch-classes. It is used to filter unwanted pitch-classes from the domains of the search-variables. For instance, by giving the list (0 2 4 5 7 9 11) we permit only 'white' notes of the piano

  keyboard.
- sols-mode indicates the number of solutions required:

  once (the default case, one solution) or

	`all` (all solutions). `sols-mode` can also be a positive integer giving the number of desired solutions.
- `rnd?`	a flag indicating whether or not the search-space is randomly reordered (by default `rnd?` is `nil`).
- `print-fl`	a flag, if `true` then the index of the current search-variable is printed in the Listener window indicating how far the search has proceeded.

4.2 Utilities Menu

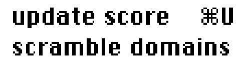


Fig.5: Utilities menu.

The "Utilities" menu (figure 5) contains two menu-items: "update score" and "scramble domains". These menus are used in connection with the `score-PMC` box.

The "update score" menu is typically used while a `score-PMC` search is running. When the user selects this menu (or types command-U) the current input score used by the `score-PMC` box is updated. The partial solution found so far is drawn up to the current search-variable. After this middle C (or 6000 in midics) is set as the default pitch (this is true only for chords that have not been constrained by the user). This tool is useful especially when a search is time consuming, as it allows the user to inspect a partial solution while the search is still running.

The "scramble domains" menu is more experimental and it may be useful in situations where the search is not proceeding well. Like the "update score" menu, the "scramble domains" menu is used while a `score-PMC` search is running. When selected it "scrambles" (or reorders randomly) all domains. This may sometimes help in dead-end situations, as the search can be sensitive to the order in which the values of the domains are tried out.

4.3 Debug Menu

Fig.6: Debug menu.

The "Debug" menu (figure 6) contains two menu-items: "rule diagnostics flag" and "engine info". These menu-items are useful for debugging purposes.

The "rule diagnostics flag" menu shows whether the "diagnostics" mode of the search-engine is on or off (the mode is on when a check-mark is drawn in front of the menu-title, it is off if there is no check-mark). When the diagnostics mode is on the engine collects during search information on how often rules fail. This information is printed in the Listener window. For instance a message like:

("suspension 1. rule" 100)

means that the rule with the documentation string `"suspension 1. rule"` has failed `100` times. This tool can be helpful in finding out if a given rule or a group of rules fail very often. This can in turn be a sign that these rules are too strict.

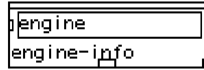


Fig.7: `engine-info` box.

The "engine info" menu creates a PW-box called `engine-info` (figure 7). It has one input which is a menu-box. Here the user can select various functions that allow to access the internal state of the current search-engine. Currently following functions are found in the menu-box:

- `engine` returns the current search-engine
- `allsols` returns all solutions found by the current search-engine
- `domains` returns the contents of the domain slots of the search-variables
- `other-values` returns the contents of the other-values slots of the search-variables
- `partialsol` returns the current partial solution.

4.4 PC-set-theory Menu

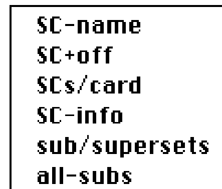


Fig.8: PC-set-theory menu.

PWConstraints provides a small PC-set-theoretical package (figure 8). No attempt will be made here to explain PC-set-theoretical concepts. Interested readers can find references and examples of the usage of PC-set-theory in combination with PWConstraints in my thesis (especially in section 5.1). PWConstraints contains the following PC-set-theory boxes (figure 9):

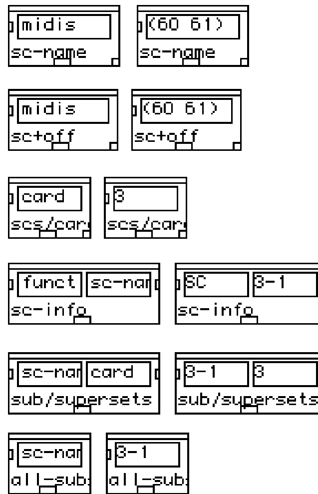


Fig.9: PC-set-theory boxes

4.4.1 SC-name midis

returns the SC-name of `midis` (a list of midi-values), `midis` can also be a list of lists of midis in which case `SC-name` returns the SC-names for each midi-value sublist.

4.4.2 SC+off midis

returns a list containing the SC-name and the offset (i.e. the transposition relative to the prime form of the SC) of `midis` (a list of midi-values). `midis` can also be a list of lists of midi-values in which case `SC+off` returns the SCs with offsets for each midi-value sublist.

4.4.3 SCs/card card

returns all SCs of a given cardinality (`card`).

4.4.4 SC-info function SC-name

allows to access information of a given SC (second input, `SC-name`). The type of information is defined by the first input (`function`). This input is a menu-box and contains the following menu-items:

- `CARD`	returns the cardinality of SC
- `PRIME`	returns the prime form of SC
- `ICV`	returns the interval-class vector of SC
- `MEMBER-SETS`	returns a list of the member-sets of SC (i.e. all 12 transpositions of the prime form)

- COMPLEMENT-PCs returns a list of PCs not included in th prime form of SC.

The second input is also a menu-box, where the user selects the SC-name. When the input is scrolled, it displays all SC-names of a given cardinality. The cardinality can be incremented by alt-clicking or decremented by alt-command-clicking the input. The input accepts also a list of SC-names. In this case the SC-info box returns the requested information for all given SC-names.

4.4.5 sub/supersets sc-name card

returns all subset classes of SC (when card is less than the cardinality of SC) or superset classes (when card is greater than the cardinality of SC) of cardinality card. The first input is a menu-box, where the user selects the SC-name. When the input is scrolled, it displays all SC-names of a given cardinality. The cardinality can be incremented by alt-clicking or decremented by alt-command-clicking the input.

4.4.6 all-subs sc-name

returns all subset classes of the given SC (SC-name). Accepts also a list of SCs in which case all subset classes of the given SCs are appended (all duplicates are removed from the result). The first input is a menu-box, where the user selects the SC-name. When the input is scrolled, it displays all SC-names of a given cardinality. The cardinality can be incremented by alt-clicking or decremented by alt-command-clicking the input.

4.5 Music Analysis Menu

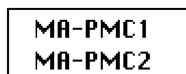


Fig.10: Music analysis menu.

MA-PMC1 and MA-PMC2 are the two primary music analysis tools provided by PWConstraints. MA-PMC1 has the following arguments (figure 11):

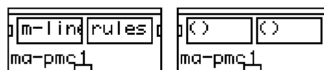


Fig.11: MA-PMC1 box.

MA-PMC1 m-lines rules

- m-lines a list of measure-lines (the output of a polyphonic RTM-box) defining the input score
- rules a list of PWConstraints rules.

MA-PMC1 is meant to be used to analyse a given score with standard PWConstraints rules. The rules are run for each note in the score (just like with score-PMC). If a given rule fails at some note, a message will be printed to the Listener window, giving the number of the rule, the name of the rule and the exact position of the failure. For instance, if we assume that a rule allowing only certain melodic intervals fails at measure 2 of part 1, we get the following message:

Rule no 1 'allowed ints' failed at: part 1, measure 2, beat 2, note 71 (B3).

MA-PMC2, in turn, has the following arguments (figure 12):

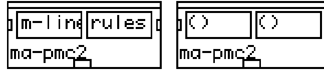


Fig.12: MA-PMC2 box.

MA-PMC2 m-lines rules

- m-lines — a list of measure-lines (a polyphonic RTM-box) defining the input score
- rules — a list of PWConstraints "analysis-rules".

MA-PMC2 is used to write analysis information directly into a PW RTM-editor. The rules used by MA-PMC2, called *analysis-rules*, are standard PWConstraints rules with a pattern-matching part and a Lisp-code part, except the value returned by an analysis-rule is not a truth value, but a collection of information consisting of: (1) the information to be printed below a note in the score; (2) the "note-position" where the information is to be printed (given as a variable name - i.e. ?1, ?2, etc.- found in the pattern-matching part); (3) the row number where the information is to be printed allowing the user to control the layout of the information. For instance, the following simple analysis-rule prints the melodic-index (mindex) of each note in the input score:

```
(* ?1
  (?if
    (values
     (mindex ?1)    ; information to be printed
     ?1             ;  note-position
     1)))            ; print at row number 1
```

The following example prints the SC identity (using the function SC-name) of each melodic three note succession in the input score:

```
(* ?1 ?2 ?3
  (?if
    (values
     (sc-name (list (m ?1) (m ?2) (m ?3)))    ; information to be printed
     ?1                                       ; print below ?1
     2)))                                     ; print at row number 2
```

4.6 DOC Menu

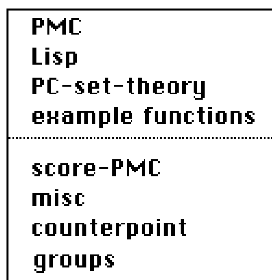


Fig.12: DOC menu.

PWConstraints is somewhat different from other PW user-libraries, as the user typically writes the rules as Lisp expressions in a text-editor. This means that there are relatively few PW-boxes provided by the system (the most important boxes are `PMC` and `score-PMC`). The main bulk of PWConstraints is in pure Lisp code. PWConstraints adds a menu called "DOC" in order to provide access to the documentation strings of the most essential functions and macros needed when writing PWConstraints rules. The menu is divided into eight sub menus: "`PMC`", "Lisp", "PC-set-theory", "example functions", "`score-PMC`", "misc", "counterpoint" and "groups". The functions included in the sub menus below the dotted line are all using the `score-PMC` box. If one of the sub menus is selected, the Lisp Documentation window is opened containing the documentation strings of the functions and macros grouped under the sub menu.

4.6.1 PMC Sub Menu

The "`PMC`" sub menu contains the documentation strings of some basic `PMC` functions:

```
pwcs::cur-index ()
[function]
```
returns the current search-index (counted from 1).

```
pwcs::cur-slen ()
[function]
```
returns the number of search-variables of the current search-engine.

`pwcs::read-key &method next-method-context self key`
`[generic-function]`
read data from place with `key`.

`pwcs::write-key &method next-method-context self key data`
`[generic-function]`
write `data` to place with `key`.

`pwcs::engine ()`
`[function]`
returns the current search-engine.

4.6.2 Lisp Sub Menu

The "Lisp" sub menu contains the documentation strings of some utility functions:

`pwcs::count-adjacent-items list  &optional test`
`[function]`
counts the number of adjacent equal items found at the beginning of `list`. The equality can be defined by giving an optional `test` function.

`pwcs::eq-ints? ints  &rest midis`
`[function]`
returns `true` if `midis` (a list of midi-values) forms the interval succession given by `ints`, where `ints` can be an interval or a list of intervals or a list of lists of intervals, `midis` can be either single midi-values or a list of midi-values.

`pwcs::find-all item sequence  &rest keyword-args  &key test test-not  &allow-other-keys`
`[function]`
find all those elements of `sequence` that match `item`, according to the keywords (doesn't alter `sequence`).

`pwcs::group list group-lens`
`[function]`
groups `list` into subsequences, where `group-lens` indicates the length of each sublist. `group-lens` can be a number or a list of numbers. If `list` is not exhausted by `group-lens`, the last value of `group-lens` will be used as a constant until `list` has been exhausted.

`pwcs::imbricate lowcard highcard step list`
`[function]`
returns a list of subsequences of `list` using `lowcard` and `highcard` to define the range of the subsequence lengths; `step` indicates how many items we proceed in `list` before starting with the next subsequence.

`pwcs::inds->vals inds ls`
`[function]`
translates a list of indices (`inds`) to actual values found in `ls`.

`pwcs::map-group-pos? l size pos`
`[function]`
`l` is first grouped into subsequences of equal length (given by `size`). `map-group-pos?` returns `true` if the last item of the last subsequence is at the position given by `pos` (the position is counted from 1) within the last subsequence. This function is useful if a rule wants to check if the last item of a partial solution is at a given position, where the partial solution is been split into subsequences of equal length. For instance, if we split the list `(1 2 3 1 2 3 1)` into subsequences of length 3 and check whether the last item of the last subsequence is at position 1, we call:
`(map-group-pos? '(1 2 3 1 2 3 1) 3 1)`
which returns true as the last item (1) is at the first position of the last subsequence.

`pwcs::setp list  &key test key`
`[function]`
returns `true` if `list` is a set (i.e. `list` does not contain duplicates).

`pwcs::unique-cell2? x y list  &aux list1`
`[function]`
checks whether the succession `x y` is not found in `list`, `list` is given in reversed order, also `x` and `y` have to be in reversed order. This function is optimised for partial solutions in reversed order (`rl`).

`pwcs::unique-cell3? x y z list  &optional cnt`
`[function]`
checks whether the succession `x y z` is not found in `list`, `list` is given in reversed order, also `x, y` and `z` have to be in reversed order. This function is optimised for partial solutions in reversed order (`rl`).

`pwcs::unique-int? int rl  &key key`
`[function]`
checks if `int` is unique (not found in `rl`). `rl` is a reversed list of midi-values. This function is optimised for partial solutions in reversed order (`rl`).

`pwcs::unique-ints? ints rl  &key key`
`[function]`
checks if `ints` are unique (not found in `rl`). `rl` is a reversed list of midi-values. This function is optimised for partial solutions in reversed order (`rl`).

`pwcs::window-list list win`
`[function]`
returns a subsequence of `list` starting from the first item. The length of the subsequence is given by `win`. If `win` is `:all` or if `win` is greater than the length of `list` then the complete `list` is returned.

4.6.3 PC-set-theory Sub Menu

The "PC-set-theory" sub menu contains the following documentation strings:

`pwcs::all-subs SCs`
`[function]`
returns all subset classes of `SCs` (a single SC or a list of SCs).

`pwcs::card SC`
`[function]`
returns the cardinality of `SC`.

`pwcs::eq-SC? set-classes  &rest midis`
`[function]`
checks whether the SC identity of `midis` (a list of midi-values) is found in `set-classes` (a list of SC-names). `set-classes` can be a single SC or list of SCs, `midis` individual midi-values or a list of midi-values.

`pwcs::ICV SC`
`[function]`
returns the interval-class vector (ICV) of `SC`.

`pwcs::prime SC`
`[function]`
returns the prime form of `SC`.

`pwcs::SC+off midis`
`[function]`
returns a list containing the SC-name and the offset (i.e. the transposition relative to the prime form of the SC) of `midis` (a list of midi-values), `midis` can also be a list of lists of midi-values in which case `SC+off` returns the SCs with offsets for each midi-value sublist.

`pwcs::SC-name midis`
`[function]`
returns the SC-name of `midis` (a list of midi-values), `midis` can also be a list of lists of midi-values in which case `SC-name` returns the SC-names for each midi-value sublist.

`pwcs::sub/supersets SC card`
`[function]`
returns all subset classes of `SC` (when `card` is less than the cardinality of `SC`) or superset classes (when `card` is greater than the cardinality of `SC`) of cardinality `card`.

`pwcs::subsets SC card`
`[function]`
returns all subset classes of cardinality `card` of `SC`.

`pwcs::supersets SC card`
[function]
returns all superset classes of cardinality `card` of `SC`.

4.6.4 Example Functions Sub Menu

The "example functions" sub menu contains the documentation strings of the functions that are used in the examples I give in my thesis (sections 5.1, 5.2, 5.4):

`pwcs::make-note-ranges start ints card`
[function]
returns a reduced search-space for chords of size `card` and with the adjacent-interval structure given by `ints`. `start` gives the pitch of the first note as a midi-value.

See the 'Constraining Chords' example.

`pwcs::check-chain? l card sc-names`
[function]
checks whether `l` (imbricated with a step size `card/2`) is a chain consisting of SCs (given by `SC-names`). No SC with the same transposition duplicate should be found in the chain.

This function is used by the 'Chains' example.

`pwcs::mk-chain-index-rules indexes scs`
[function]
returns a list of index-rules for the 'Chains' example.

`pwcs::check-stats? item count stats &optional tolerance`
[function]
checks that `item's` count (given by `count`) does not exceed `item's` count found in `stats`, where `stats` is a list of count-item pairs: `((count1 item1) (count2 item2) ... (countN itemN))`. The difference `(- count tolerance)` should be less or equal than the respective count found in `stats`.

See the 'Automatic Rules and Statistical Distribution' example.

`pwcs::SC-distribution card l`
[function]
gives all SC-names found in `l` when `l` is grouped into subsequences starting from each successive item. The length of each subsequence is given by `card`.

See the 'Automatic Rules and Statistical Distribution' example.

`pwcs::interval-distribution card l`
[function]
gives all interval successions found in `l` when `l` is grouped into subsequences starting from each successive item. The length of each subsequence is given by `card`.

See the 'Automatic Rules and Statistical Distribution' example.

`pwcs::+-distribution card l`
[function]

gives the '+-' movements of `l` when `l` is grouped into subsequences starting from each successive item. The length of each subsequence is given by `card`.

See the 'Automatic Rules and Statistical Distribution' example.

`pwcs::count-stats lst`
[function]

calculates the frequency distribution of `lst` and returns the result as `count-item` pairs. The result is sorted in descending order according to the count information

(i.e. the items that are found most often are found first in the result list).

`pwcs::atleast-cnt-check l total-len atleast-cnt-item-lst &optional test`
[function]

checks whether `l` fulfils the 'at-least property'.

See the 'Atleast property' example.

`pwcs::data-group-of-part partnum-data-lists group-len partnum`
[function]

`partnum-data-lists` is a list of `partnum-data` pairs, `group-len` indicates the length of subgroups and `partnum` is the part number of the current part. `data-group-of-part` collects all `partnum-data` pairs that belong to the current part to a list (the current part is given by `partnum`). Then this list is grouped into sublists (the length of the sublists is given by `group-len`). `data-group-of-part` returns the data items of the first sublist.

See the 'Splitter' example.

`pwcs::sp-partnum partnum-data`
[function]

returns the partnum part of `partnum-data`.

See the 'Splitter' example.

`pwcs::reduce-mel mel`
[function]

returns a list of lists of time-midi value pairs defining the reduced melodic line of `mel`, `mel` can be a list of midi-values or a list of lists where each sublist consists of time-midi value pairs.

See the 'Syyssonetti' example.

`pwcs::part-arc-lens mel`
[function]

returns the partial arc lengths of `mel`. See the 'Syyssonetti' example.

`pwcs::skyline mel`
[function]

returns the skyline of `mel`.

See the 'Syyssonetti' example.

4.6.5 Score-PMC Sub Menu

The "`score-PMC`" sub menu contains the documentation strings of some essential functions and macros that are used in connection with the `score-PMC` box:

`pwcs::part-collection ()`
`[function]`
returns the current `part-collection`.

`pwcs::m (s-variable)`
`[macro]`
returns the value (`m` for midi) of `s-variable`.

`pwcs::nindex (s-variable)`
`[macro]`
returns the note index (given by score-sort) of `s-variable` (starting from 0).

`pwcs::mindex (s-variable)`
`[macro]`
returns the mel-index (melodic index) of `s-variable` (starting from 1).

`pwcs::beat (s-variable)`
`[macro]`
returns the linked-beat of `s-variable`.

`pwcs::measure (s-variable)`
`[macro]`
returns the linked-measure of `s-variable`.

`pwcs::hc (s-variable)`
`[macro]`
returns the harmonic-context of `s-variable` as a list `s-variables`.

`pwcs::hslice (s-variable)`
`[macro]`
returns the harmonic-slice of `s-variable` as a list `s-variables`.

`pwcs::beatnum (s-variable)`
`[macro]`
returns the beat number (inside a measure) of the beat of `s-variable` (starting from 1).

`pwcs::measurenum (s-variable)`
`[macro]`
returns the measure number of the measure of `s-variable` (starting from 1).

pwcs::partnum (s-variable)
[macro]
returns the part number of the part of `s-variable` (starting from 1).

pwcs::hc-midis (s-variable)
[macro]
returns all midi-values of the harmonic-context of `s-variable`.

pwcs::l->ms s-variables
[function]
returns the midi-values of `s-variables`.

pwcs::startt obj
[function]
returns the start time (in ticks) of `obj` (`s-variable` or `hslice`).

pwcs::endt obj
[function]
returns the end time (in ticks) of `obj` (`s-variable` or `hslice`).

pwcs::durt obj
[function]
returns the duration (in ticks) of `obj` (`s-variable` or `hslice`).

pwcs::attack-item? hslice s-variable
[function]
returns `true` if `s-variable`, being a member of hslice, is an attack `s-variable`.

pwcs::attack-items? hslice s-variables
[function]
returns `true` if all `s-variables` in `hslice` are attack-items.

pwcs::collect-only-attacks hslice
[function]
returns only attack-items of `hslice` (harmonic-slice).

pwcs::rtm-pattern (beat)
[macro]
returns the rtm-pattern of `beat`.

pwcs::match-rtm? (rtm-pat+vars)
[macro]
returns `true` if all variables mentioned in `rtm-pat+vars` share the same rtm-pattern and the rtm-positions of the variables match the positions given by `rtm-pat+vars`. For instance a `rtm-pat+vars` like `(1 ((?1 1) (?2 1)))` would match all cases where `?1` and `?2` form the rtm-pattern `(1 (1 1))`.

`pwcs::voice-count ?1`
`[function]`
returns the number of (sounding) parts at `?1`.

`pwcs::bass-num ()`
`[function]`
returns the partnum of the lowest (bass) part.

`pwcs::bass-item? ?1`
`[function]`
returns `true` if `?1` is a bass-item, i.e. has the largest partnum.

`pwcs::give-bass-item ?1`
`[function]`
returns the bass-item of `?1`, i.e. the `s-variable` that has the largest partnum.

`pwcs::sop-item? ?1`
`[function]`
returns `true` if `?1` belongs to the highest (soprano) part.

`pwcs::rest-item? ?1`
`[function]`
returns `true` if `?1` is followed by a rest.

`pwcs::prev-rest? ?1`
`[function]`
returns `true` if `?1` is preceded by a rest.

`pwcs::one-prev-rests? &rest s-variables`
`[function]`
returns `true` if one of `s-variables` is preceded by a rest.

`pwcs::last-measure? ?1`
`[function]`
returns `true` if `?1` belongs to the last measure.

`pwcs::nextto-last-measure? ?1`
`[function]`
returns `true` if `?1` belongs to the next to last measure.

`pwcs::last-item? ?1`
`[function]`
returns `true` if `?1` is the last note within a part.

`pwcs::constraint? ?1`
`[function]`
returns `true` if `?1` is constrained, i.e. its domain contains only one value.

pwcs::constraints? &rest s-variables
[function]
returns true if all s-variables are constrained, i.e. their domains contain only one value.

pwcs::one-constraint? &rest s-variables
[function]
returns true if at least one s-variable of s-variables is constrained.

pwcs::complete-chord? s-variable
[function]
returns true if s-variable 'completes' the chord, i.e. is the last note (in accordance with score-sort) in the hslice of s-variable.

pwcs::downbeat? s-variable
[function]
returns true is s-variable is on downbeat.

pwcs::on-main-beat? s-variable
[function]
returns true is s-variable is on downbeat and in the first beat of a measure.

pwcs::prev-item-on-downbeat s-variable
[function]
returns previous downbeat s-variable of the previous beat.

pwcs::prev-item-on-downbeat-2 s-variable
[function]
returns previous downbeat s-variable on (previous (previous beat)).

4.6.6 Misc Sub Menu

The "misc" sub menu contains the documentation strings for miscellaneous functions that are used with the score-PMC box:

pwcs::no-voice-crossings? s-var
[function]
no voice crossings allowed.

pwcs::no-bass-crossings? s-var
[function]
bass (= lowest part at s-var) should have the lowest midi-value.

pwcs::no-sop-crossings? s-var
[function]
sop (= highest part at s-var) should have the highest midi-value.

`pwcs::no-unisons? s-var  &optional only-downbeat?`
`[function]`

harmonic unisons not allowed, if `only-downbeat?` is `true` then only downbeat cases are checked.

`pwcs::no-parts-voice-crossings? s-var partnums`
`[function]`

checks that parts belonging to `partnums` are not allowed to cross.

`pwcs::inside-voice-distance? ?1 tolerance`
`[function]`

voices above or below `?1` should have pitches that are within `tolerance` distance to the pitch of `?1`.

`pwcs::prev-long-note-midis rl time-limit  &optional cnt time-window`
`[function]`

collects all previous midi-values for notes whose durations are equal or longer than `time-limit` (in ticks). If `cnt` is given only `cnt` previous midi-values are returned. If `time-window` (in ticks) is given `prev-long-note-midis` returns only 'long-note' midi-values inside `time-window`.

`pwcs::parallel-movements? &rest sop-bass-ints`
`[function]`

returns `true` if sop-bass intervals (`sop-bass-ints`) move in parallel.

`pwcs::no-chord-duplicates? ?1 rl  &optional ch-key`
`[function]`

checks that `rl` does not contain chord-duplicates with `?1`.

`pwcs::vertical-interval-range? s-var minv maxv`
`[function]`

`s-var` and the `s-variable` belonging to the part below `s-var's` part should have pitches within the range given by `minv` and `maxv`.

`pwcs::vertical-intervals? s-var intervals`
`[function]`

`s-var` and the `s-variable` belonging to the part below `s-var's` part should have pitches that form vertical intervals that belong to `intervals`.

`pwcs::search-n-mel-moves n ?1`
`[function]`

returns `n` (`?1` included) previous melodic pitches disregarding repetitions.

`pwcs::octaves? midis`
`[function]`

returns `true` if `midis` contains octaves, unisons are not considered.

```
pwcs::no-voice-octaves? s-var
[function]
```
the harmonic-context (`s-var` included) of `s-var` should not contain octaves (unisons allowed).

4.6.7 Counterpoint Sub Menu

The "counterpoint" sub menu contains the documentation strings of miscellaneous counterpoint functions that are used with the `score-PMC` box (section 5.3):

```
pwcs::ballistic? midi1 midi2 midi3
[function]
```
`midi1`, `midi2` and `midi3` should form a "ballistic" melodic movement. A ballistic movement allows two jumps in the same direction, but the larger jump has to be below the smaller one.

```
pwcs::check-parallels? ?11 ?12 ?21 ?22 PC-int-pairs
[function]
```
checks whether the melodic movement in `?11-?12` in one voice and `?21-?22` in another voice forms a parallel movement. `PC-int-pairs` (given as pitch-class interval pairs) defines the harmonic parallel movements we are interested in. For instance if `PC-int-pairs` is `((0 0) (7 7))`, it means that we check for parallel octaves or fifths.

```
pwcs::open-parallels? ?1 ?2  &optional fifths?
[function]
```
checks if there are open parallels among the harmonic-context of `?2`. Parallel octaves, parallel 5ths and tritones proceeding to perfect 5ths are disallowed. If the optional argument `fifths?` flag is `nil` then only parallel octaves are disallowed.

```
pwcs::downbeat-parallels? ?1 ?2  &key dist fifths?
[function]
```
similar to the function `open-parallels?`, except all notes should be downbeat notes.

```
pwcs::hidden-parallels? ?1 ?2  &optional fifths?
[function]
```
checks if there are hidden parallels among the harmonic-context of `?2`. Hidden octaves and 5ths are disallowed. If the optional argument `fifths?` flag is `nil` then only hidden octaves are disallowed.

```
pwcs::dissonant-bass-int? int
[function]
```
returns `true` if modulo 12 of `int` is a minor or major 2nd, a perfect 4th, a tritone, or a minor or major 7th.

```
pwcs::dissonant-upper-int? int
[function]
```
returns `true` if modulo 12 of `int` is a minor or major 2nd, or a minor or major 7th.

```
pwcs::harmonic-dissonance? ?1  &optional hc
[function]
```
returns a dissonance `s-variable` if `?1` forms a harmonic dissonance or otherwise `nil`. Assumes that the lowest part has always the lowest pitch.

```
pwcs::match-upbeat-rtm? ?1 ?2
[function]
```
returns `true` if `?2` is an upbeat note, i.e. `?2` is on the unstressed part of the beat.

```
pwcs::cambiata-ints? ?1 ?2 ?3 ?4
[function]
```
checks whether the four notes in the argument list form a cambiata succession of intervals.

```
pwcs::cambiata-rtm? ?1 ?2 ?3 ?4
[function]
```
checks whether the four notes in the argument list form a cambiata rtm-pattern.

```
pwcs::scale-movem? n1 n2 n3
[function]
```
returns `true` if `n1 n2 n3` (in midi-values) are in ascending or descending order and the adjacent note intervals between them do not exceed a major 2nd.

```
pwcs::side-movem? n1 n2 n3  &optional only-down
[function]
```
returns `true` if `n1` is equal to `n3` and `n2` is maximally a major 2nd higher or lower than `n1` (`n1 n2 n3` are midi-values). The optional argument, `only-down`, is a flag indicating if only downward movements are allowed.

```
pwcs::upbeat-dissonance? ?1 ?2 ?3 ?4  &optional only-down
[function]
```
checks whether `?2` is an upbeat dissonance. The optional argument, `only-down`, is a flag indicating if only downward neighbor notes are allowed.

```
pwcs::suspension-rtm-match? ?1 ?2 ?D
[function]
```
matches the possible 'suspension' rtm-patterns.

```
pwcs::harmonic-dissonance2? ?1 ?2
[function]
```
returns true if `?1` and `?2` form a harmonic dissonance.

```
pwcs::suspension-part1? ?D  &optional stepwise
[function]
```
checks whether `?D` is consonant or a potential suspension dissonance. If the optional argument `stepwise` is `true` then the melodic movement to `?D` should be stepwise.

pwcs::suspension-part2? ?1 ?2
[function]
checks that ?2 resolves the suspension when ?D is a suspension dissonance.

pwcs::imitation? ?1 ref-part start-index end-index
imit-part imit-start-index imit-int
[function]
imitation rule, where ?1 is the current note, ref-part is the partnum of the part to be imitated, start-index is the start position and end-index the end position (both counted from 1) of the melodic line to be imitated; imit-part is partnum of the part that imitates ref-part, start-index is the start position where the imitation starts and imit-int gives the imitation interval (or a list of intervals). For instance, if we want the first 5 notes of part 1 to be imitated one octave lower by part 2 also starting from note 1, we write the following rule:

(* ?1 (imitation? ?1 1 1 5 2 1 -12) "imitation rule")

4.6.8 Groups Sub Menu

The "groups" sub menu contains the documentation strings of "grouping" functions that are used with the score-PMC box (groups are discussed in more detail in the PW Extensions section):

pwcs::note-in-group-at-pos? s-variable pos group-name
[function]
returns true if s-variable belongs to a group with group-name and is found at the position pos within that group (counting from 1).

pwcs::note-found-in-nth-group? s-variable group-pos group-name
[function]
returns true if s-variable belongs to a group with group-name and the group is at the position group-pos (counting from 1) within the groups of s-variable's part.

pwcs::prev-group-svars s-variable group-name
[function]
access previous group-s-variables with group-name found in s-variable's part.

pwcs::prev-group-midis s-variable group-name
[function]
access previous group-midis with group-name found in s-variable's part.

pwcs::group-at-pos-svars s-variable partnum group-pos group-name
[function]
access group-s-variables (indexed by partnum and group-pos) with group-name.

pwcs::group-at-pos-midis s-variable partnum group-pos group-name
[function]
access group-midis (indexed by partnum and group-pos) with group-name.

5 Appendix

Sections 5.1, 5.2 and 5.3 from the fifth chapter of my thesis:

Mikael Laurson, PATCHWORK, A Visual Programming Language and some Musical Applications, Sibelius Acadely, Department of Composition and Music Theory, *Studia Musica* No. 6, 1996.

5.1 Introduction

In this chapter we give an example of a large PW application, *PWConstraints*. It is a rule-based programming language used to solve complex musical problems. These are typically situations where the user wants to describe the end result using several analytical standpoints. Writing counterpoint is a good example. The end result is described with the help of rules controlling melodic lines, harmony, voice-leading, etc. Like all user-libraries, PWConstraints is loaded on top of PW.

The chapter is divided into four main sections. Section 5.1 offers background information and examines the main features of PWConstraints from a user's point of view.

First we see where PWConstraints sits within the field of different programming strategies. We analyse it as a constraint satisfaction problem language and refer to a number of other related environments (section 5.1.1).

When introducing PWConstraints features themselves, our starting point will be the concept of a *search-space*. We see how it is defined and how it affects the end result (section 5.1.2). We then discuss how to write PWConstraints rules. We implement a *pattern-matching scheme* that allows us to extract information needed by the rules (section 5.1.3).

In sections 5.1.4 and 5.1.5 we study several concrete examples, either general or specific in musical nature.

Section 5.2 is mostly a technical discussion. We first examine in detail the implementation of the backtrack search-engine (sections 5.2.1). The compilation of pattern-matching rules to ordinary Lisp functions is discussed in section 5.2.2. In section 5.2.3 we introduce an important extension, allowing us to represent a *partial solution* as CLOS objects. Section 5.2.4 introduces a new type of PWConstraints rule, allowing the user to search for "better" results. We then investigate the possibility of adding the PWConstraints rule formalism to another existing CSP language (section 5.2.5). The interface between PWConstraints and PW is discussed in section 5.2.6. Finally, in section 5.2.7, we address efficiency issues by implementing a forward checking algorithm.

There then follows a discussion on two large-scale case studies (sections 5.3 and 5.4, respectively).

In the first of these, we start by studying the problem of translating counterpoint rules to the PW-Constraints formalism. The rules are drawn from counterpoint text books. Section 5.3.1 gives a preparatory discussion by making comparisons between some corresponding aspects of PW-Constraints and the *CHORAL* system given in Ebcioglu (1992). We then discuss the representation

of musical information, extending a simple queue structure to a full-scale score representation scheme (section 5.3.2). The actual translation of the counterpoint rules, observing a number of melodic, voice-leading, harmonic, etc. considerations, is done in section 5.3.3. The extended score representation scheme requires both an extended search-space specification and a new version of the PW interface. These are given in sections 5.3.4 and 5.3.5, respectively.

The function of the latter case study is to harmonise a precomposed melodic line according to a set of rules. The basic aspects of this application, called *Harmonic Sequence Generator*, or HSG, are discussed in sections 5.4.1 and 5.42. A user-definable harmonic library, providing the basic chord materials for HSG, is then examined in section 5.4.3. The design of the actual search problem will be discussed in two parts. In the former part (sections 5.4.4 and 5.4.5), we first examine the generation of a three-part contrapuntal "skeleton" for the final result. We then go into a bit more detail, defining a number of rules controlling melodic intervals, repetitions, aspects of voice-leading, harmonic progressions, etc. The second part investigates how the contrapuntal skeleton is to be turned into a final musical result (section 5.4.6). After implementing the interface between HSG and PW (section 5.4.7) we are ready to produce an actual musical example, harmonising a melodic line by the composer Paavo Heininen (section 5.4.8).

This chapter is concluded by a short discussion that attempts to evaluate PWConstraints (section 5.5).

5.1.1 PWConstraints in Perspective

5.1.1.1 Background

When using a procedural programming language, such as C or Pascal, or a functional language, like Lisp or PW, the user has to solve a problem in a stepwise manner.[1] In many cases this approach is an adequate one, but for many types of problem it may lead to programs that are difficult to design or understand.

Descriptive (declarative) languages, like Prolog, offer an alternative way to look at this problem: instead of trying to solve a problem step-by-step, the user describes a possible result with the help of a set of rules. It is then up to the language to find solutions that are coherent with the descriptions.[2] This approach is probably more natural for individuals with a musical background. A typical music-theoretical writing offers a discussion on some properties of some pieces of music, not a step-by-step description of how those pieces were made.

PWConstraints can be thought of as a descriptive language. When using it we do not formulate stepwise algorithms, but define a search-space and produce systematically potential results from it. Typically we are not interested in all possible results, but filter (or, rather, *constrain*) these with the help of rules describing an acceptable solution.

Besides the advantages, the descriptive languages may have also some disadvantages. First, in most cases one has to specify an extremely large search-space. As a result, one has to be able to write highly efficient rules to find solutions within reasonable time. Furthermore, rules should be

1. A general discussion of imperative, procedural, functional, declarative and object oriented languages is found in Norvig (1992:434-435).
2. Clocksin and Mellish (1984:VII).

functional also with *partial solutions*. As it turns out, achieving this is sometimes difficult. Finally, if the rules are too strict or conflict with each other, there is no guarantee of finding a solution at all.

A specific problem in the musical domain is that the task of describing a musical result is far from trivial. As such a result is typically seen from many different points of view, we need highly flexible data structures to describe musical structures. Each rule should be able to dynamically extract required information out of a data structure, allowing the rule to analyse a result from its own point of view. One of the main problems in formulating such rules is to find a clear formalism with which to point to the required data objects.

5.1.1.2 PWConstraints as a Constraint Satisfaction Problem Language

PWConstraints is an attempt to solve some of the descriptive language problems identified above. It is a *constraint satisfaction problem* (CSP) language, containing a number of powerful computational tools such as the *backtrack search-engine*, the *pattern-matching interface* and the *forward checking* algorithm.[1]

A CSP consists of a set of *search-variables*, each of which must be instantiated in a particular *domain* and a set of *rules* (or *predicates*) that the *values* of the search-variables must simultaneously satisfy.[2] We will call the set of search-variables, each having its domain, a *search-space*. A CSP is most often solved by using a backtrack search.

Constraint satisfaction problems already have a long history in the field of AI.[3] One of the main trends in CSP research has been the development of tools improving the efficiency of a pure backtrack search. In principle, tools like these should be as general as possible, i.e. they should not be domain specific.

In contrast to this general trend in CSP research, our discussion will be highly domain specific. Whenever we can, we will optimise a search by using the musical knowledge we have about the desired result. Although this approach is not as general as that of the classical CSP, we believe that both domain specific and non-domain specific optimisations are needed when trying to solve complex compositional tasks in an efficient way.

5.1.1.3 Related Work

Other constraint-based languages devoted to musical problems include Camilo Rueda's *Situation*, developed at IRCAM, and Kemal Ebcioglu's CHORAL project.[4]

---

1. Although backtracking is used extensively in PWConstraints, it is not our intention here to offer a general discussion on other types of search methods (neural networks, genetic algorithms, etc.). At an early state of the PWConstraints project, genetic algorithms were in fact tested together with the backtracking approach. The results, however, were not encouraging. For a discussion on genetic algorithms, see Goldberg (1989).

2. Mackworth (1977:99). We will use the concept "search-variable" instead of "variable" used by Mackworth, as the latter term will be reserved for the pattern-matching part of our discussion.

3. The interested reader can follow this history in journals like Artificial Intelligence. These publish CSP-related studies regularly.

Like PWConstraints, Rueda's Situation is also written in Common Lisp and has a backtrack search-engine, a set of predefined rules and a pattern-matching language. The last-mentioned is intended to help the user in giving arguments to existing rules in a flexible way. Situation is designed specifically for generating chord sequences. It has a PW interface and allows the user to build new rules by using PW-boxes. Recently, Situation has also been used to generate metric rhythms, guitar fingerings, etc.

CHORAL will be examined in more detail in section 5.3.1, when we discuss the problem of translating counterpoint rules to the PWConstraints formalism.

Besides the above-mentioned examples, there is also a general-purpose constraint-based language implemented in Common Lisp, called *SCREAMER*. It can replace the search-engine part of PWConstraints. SCREAMER will be described in section 5.2.5, after some necessary concepts have been introduced.[1]

5.1.2 Defining a Search-Space

Solving a search problem in PWConstraints is typically done in three steps: (1) a search-space is defined; (2) a set of rules is written; (3) the search-engine is run by giving the search-space and the rules as arguments.

We will proceed with the help of some simple musical examples. In the first one of these, the task is to find all possible three-note successions in a certain search-space.

Let us start by defining the search-space. Each of the three notes (or search-variables) can be either C4, D4 or E4. In other words, C4, D4 and E4 constitute the domain of each of the three notes. The domain can also be given in MIDI-values: `60`, `62` and `64`. For practical reasons we will from now on use the latter standard.

Technically we can say that we have assigned the MIDI-value list `(60 62 64)` to each of the three notes. All solutions to our problem are found by taking all possible three-note paths. This is achieved by selecting *one* MIDI-value at a time from each MIDI-list. More formally, we can say that we take the *Cartesian product* of all domains. The *number of possible paths* is calculated by multiplying the lengths of the given domains together. In this case there are three lists, each of them containing three "candidates". Altogether we have 3*3*3 = 27 paths.

```
N1  N2  N3

60  60  60
62  62  62
64  64  64
```

Fig.5.1: The search-space of a three-note succession.

Figure 5.1 gives the search-space of our example. It consists of three notes (search-variables) `N1`, `N2` and `N3`. The domains are given as a column under each note.

4. *Situation* is described in Rueda and Bonnet (1993b), CHORAL in Ebcioglu (1992).
1. For more information about SCREAMER, see Siskind and McAllester (1993).

All possible paths (result lists) can be found for example as follows: first we pick the first candidate of each note, producing the result list `(60 60 60)`. The next list is found by taking the first candidates of `N1` and `N2` and the second of `N3`, producing the list `(60 60 62)`. Next we take the first candidates of `N1` and `N2` and the third of `N3`: `(60 60 64)`. The first candidate of `N1`, the second of `N2` and the first of `N3` produce `(60 62 60)`, etc. The entire set of paths is given in figure 5.2.

```
(60 60 60) (60 60 62) (60 60 64)
(60 62 60) (60 62 62) (60 62 64)
(60 64 60) (60 64 62) (60 64 64)
(62 60 60) (62 60 62) (62 60 64)
(62 62 60) (62 62 62) (62 62 64)
(62 64 60) (62 64 62) (62 64 64)
(64 60 60) (64 60 62) (64 60 64)
(64 62 60) (64 62 62) (64 62 64)
(64 64 60) (64 64 62) (64 64 64)
```

Fig.5.2: The Cartesian product of a three-note problem.

The search in our first example had two specific notions to it: the search-engine was run without any rules and we asked for all solutions.

Let us take a new example by extending the previous one. We now have a search-space of five notes, each note having the domain `(60 62 64 65 67 69 71 72 74 75 77)`:

```
N1  N2  N3  N4  N5
60  60  60  60  60
62  62  62  62  62
64  64  64  64  64
65  65  65  65  65
67  67  67  67  67
69  69  69  69  69
71  71  71  71  71
72  72  72  72  72
74  74  74  74  74
75  75  75  75  75
77  77  77  77  77
```

Fig.5.3: A five-note search-space.

The length of each domain being 11, the number of solutions is 11*11*11*11*11 = 161051. The first path is `(60 60 60 60 60)`, the second `(60 60 60 60 62)`, the third `(60 60 60 60 64)`, etc.

If we then extend our five-note example so that each note would have the domain `(60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77)`, the number of all paths would be 18*18*18*18*18 = 1889568. Having a 20-note example with these same 18-value domains would produce 1274823621639607817443 7376 paths.

As can be inferred from these enormous numbers, the user has to be very careful in defining the search-space.

On the other hand, a Cartesian product is very rich in possibilities: it contains all ordered pitch successions within the constraints of the search-space. Furthermore, a musical problem is often easy to translate into a search-space, and representing the search-space with the domain lists is also very economical. We need only 20 lists, each containing 18 items, to describe a search-space producing the above mentioned number of paths.

5.1.2.1 Random Ordering

Next we examine some variations on how a search-space can be defined. Until now we have looked at all solutions. Often it is enough to get only the first solution. In this case the order of the candidates affects the result. For instance we can randomly reorder each domain of figure 5.3. One possible ordering is given in figure 5.4:

N1	N2	N3	N4	N5
74	67	74	62	74
67	64	69	65	77
65	72	67	69	60
75	62	77	77	62
64	71	60	67	65
60	74	75	60	75
71	69	72	71	67
62	65	62	75	69
69	75	64	72	64
72	60	65	74	72
77	77	71	64	71

Fig.5.4: Five note search-space after reordering.

Now the first path is `(74 67 74 62 74)` and not `(60 60 60 60 60)` as in figure 5.3. This is an interesting feature because it allows us to easily get results which are probably quite different, even though we use the same basic search-space and the same set of rules.

5.1.2.2 Preference Ordering

Ordering a search-space gives us a way to prefer some values.

N1	N2	N3	N4	N5
60	64	67	72	77
62	65	69	71	75
64	62	65	74	74
65	67	71	69	72
67	60	64	75	71
69	69	72	67	69
71	71	62	77	67
72	72	74	65	65
74	74	75	64	64
75	75	60	62	62
77	77	77	60	60

Fig.5.5: A sorted search-space.

In figure 5.5 the domain of `N1` is ordered so that we always try first out values that are near to `60`. The domain of `N2` is sorted to be near `64`, `N3` to `67`, `N4` to `72` and `N5` to `77`. The overall effect of this ordering is that there is a strong tendency to produce ascending pitch successions.

5.1.2.3 Constraining Individual Notes

Another interesting variation is to constrain some notes in advance.

N1	N2	N3	N4	N5
74	67	72	62	74
67	64		65	77
65	72		69	60
75	62		77	62
64	71		67	65
60	74		60	75
71	69		71	67
62	65		75	69
69	75		72	64
72	60		74	72
77	77		64	71

Fig.5.6: A five note search-space with the third note constrained to 72.

In figure 5.6 we have constrained the third note to 72 (C5). This means that the third note will be C5 in all possible solutions. This feature allows us to "compose" part of the solution in advance and ask the search-engine to fill in the unknown parts. Now the number of possible paths is reduced to 11*11*1*11*11 = 14641.

5.1.2.4 Moulding a Search-Space

As a last example we can force the five note succession to ascend by "moulding" the domains as here:

N1	N2	N3	N4	N5
60				
62	62			
64	64	64		
	65	65	65	
	67	67	67	67
		69	69	69
		71	71	71
			72	72
			74	74
				75
				77

Fig.5.7: Five note ascending succession.

The number of paths in figure 5.7 is 3*4*5*6*7 = 2520 instead of 161051. Of course one can have many variations on this idea: descending, ascending-descending, making holes in the search-space, etc.

5.1.3 Writing Rules

Using the search-engine without any rules is of course not interesting because there are typically too many solutions and most of these are probably totally irrelevant. One of the central problems in a constraint based language is how the user can write rules in an efficient, clear and compact way without having to know in detail how the actual search-engine has been implemented. This is of course a crucial question because it is in the rules that the user defines the musical identity

of the result.

PWConstraints provides a standard protocol for writing rules. Figure 5.8 gives the structure of a PWConstraints rule:

```
(<pattern-matching part> <Lisp-code> <doc string>)
```

Fig.5.8: The structure of a PWConstraints rule.

The rules are written always in three parts. We start with a *pattern-matching part*. Next we write a *Lisp-code part*, that is typically using information extracted by the pattern-matching part. The Lisp-code is a test function that either returns true or nil. As a final step we write a documentation string (*doc string*).

5.1.3.1 Partial Solutions

In order to understand how PWConstraints rules work during a search we will now take a close look on how the backtrack search-engine produces candidates for the rules. To assist the further discussion we repeat in figure 5.9 the search-space of the three note problem:

```
N1  N2  N3
60  60  60
62  62  62
64  64  64
```

Fig.5.9: Three note succession search-space.

When we explained above how the search-engine produces solutions we simplified somewhat by saying that it makes the first solution by picking the first items from each search-variable: (60 60 60). What actually happens is that the search-engine does not make complete solutions in one go, but builds up the solution step-by-step. After this it calls the rules that either accept or reject the new candidate. This means that the rules are typically working with *partial solutions*.

The other difference is that instead of picking only the first accepted candidate of a given search-variable and going immediately to the next one, the search-engine checks all possible values of the search-variable in one phase and collects the accepted ones in a list. The first item of this list is stored as the current accepted value and the rest of the list is stored in a buffer. This buffer will be used later if the search runs in a dead-end situation and we have to come back (or backtrack) to the search-variable. In this case the rules are not run again because each value in the buffer is equally valid at the current state of the search. The first item is simply popped from the buffer and the old accepted value is replaced with this item.

We call the current accepted candidate as *value* and the buffer that contains all other accepted candidates as *other-values*.

We will next simulate step-by-step the three note problem by assuming a rule that does not allow duplicates in the result:

```
                   N1           N2          N3
1.step            Ø
 value          60            ()             ()
 other-values   (62 64)       ()             ()

 2.step                       ↓
 value          60            62             ()
 other-values   (62 64)       (64)           ()

 3.step                                      ↓
```

```
value          60             62             64
other-values   (62 64)        (64)           ()        succeed! (60 62 64)

4.step                        ↓
value          60             64             ()
other-values   (62 64)        ()             ()

5.step                                       ↓
value          60             64             62
other-values   (62 64)        ()             ()        succeed! (60 64 62)

6.step         ↓
value          62             ()             ()
other-values   (64)           ()             ()

7.step                        ↓
value          62             60             ()
other-values   (64)           (64)           ()

8.step                                       ↓
value          62             60             64
other-values   (64)           (64)           ()       succeed!  (62 60 64)

9.step                        ↓
value          62             64             ()
other-values   (64)           ()             ()

10.step                                      ↓
value          62             64             60
other-values   (64)           ()             ()       succeed!  (62 64 60)

11.step        ↓
value          64             ()             ()
other-values   ()             ()             ()

12.step                       ↓
value          64             60             ()
other-values   ()             (62)           ()

13.step                                      ↓
value          64             60             62
other-values   ()             (62)           ()       succeed!  (64 60 62)

14.step                       ↓
value          64             62             ()
other-values   ()             ()             ()

15.step                                      ↓
value          64             62             60
other-values   ()             ()             ()       succeed!  (64 62 60)
```

Fig.5.10: The simulation of the three note problem.

The search starts by selecting `N1` as the current note (see the small arrow pointing at `N1` in figure 5.10). Next we loop through the domain of `N1`, `(60 62 64)`. We check each time that the partial result has no duplicates. At `N1` the partial solution contains only one item meaning that all items are accepted. The first accepted item, `60`, is written as a value and the rest of the accepted items `(62 64)` as other-values (step `1`).

Now we proceed to N2 (step 2) and loop through the domain of N2. The partial solution is built by collecting all values of previous notes as a list. Then we add to the *tail* of this list the first item from the domain of the current note. The first partial solution is (60 60) that fails because it contains duplicates. The next partial solution, (60 62), succeeds. Also the next one, (60 64), succeeds. Now there are two accepted values for N2, (62 64), and we write 62 as value of N2 and (64) as other-values.

We go to step 3 and loop through the domain of N3. We first collect the values of previous notes (60 62) and add to the tail of this list 60. The partial solution, (60 62 60), fails. Also the next one, (60 62 62), fails. The next one, (60 62 64), succeeds. This is also the first complete solution because we are pointing to the last note. If we look only for one solution we would now be finished with the search.

If we want to find more solutions we have to backtrack to the previous note N2 (see step 4). We disregard the old value and write the first item of other-values as the new value: value of N2 is now 64. It is important to note that the other-values list of N2 is now (). Also we do not have to run any rules in this case, because 64 was already accepted previously as a valid value.

We go now to step 5. We point now at N3 and collect all previous values and loop through the domain of N3. The first partial solution, (60 64 60), fails. The next one, (60 64 62), succeeds and it is also a complete solution because we are pointing to the last note. We continue and take the next partial solution, (60 64 64), that fails.

Next we backtrack again to N2 (step 6). But now we can not select a new value because other-values of N2 is (). This means that we have to backtrack to N1 and write a new value 62 for N1.

In step 7 we find two acceptable partial solutions: (62 60) and (62 64). We write as value 60 and as other-values (64).

In step 8 we find one acceptable partial solution: (62 60 64) which is also our third complete solution.

In step 9 we backtrack to N2 and write 64 as the new value of N2. We go to N3 (step 10) and succeed with (62 64 60). This is our fourth complete solution.

To get more solutions we have to backtrack to N2. N2 has no new values because other-values of N2 is (). We backtrack to N1 (step 11) and select 64 as the new value for N1.

In step 12 we proceed to N2 and find two acceptable partial solutions (64 60) and (64 62). We continue to N3 and find the fifth complete solution, (64 60 62), (step 13).

To find more solutions we backtrack to N2 and select 62 as the new value for N2 (step 14). We proceed to N3 and find the sixth complete solution, (64 62 60), (step 15).

We could still backtrack to N2 to find more solutions but we have no items in other-values of N2. The same situation occurs also when we backtrack further to N1. This means also that we have to stop the search because we can not backtrack beyond N1.

This technique of interleaving partial solutions - produced systematically by the search-engine - with the testing of those solutions by the rules, makes it possible to get results in a reasonable time even when the search-space is very large.[1] The rules should reduce the size of the search-space as early as possible. In our example the rule that disallowed duplicates was already able to make choices with partial solutions containing only two items.

Backtracking provides us with a way of undoing partial solutions that lead to dead-end situations. This is an important feature because we have to make choices before we know the complete solution.

1. Of course this does not *guarantee* that we will get a result. As we will see later in this chapter the success of a search depends on many things like the search-space, the rules, etc.

5.1.3.2 Pattern-Matching

Pattern-matching has already a long tradition in artificial intelligence. Maybe the most famous example is a program called *ELIZA*.[1] The main idea is to give an illusion that ELIZA understands a conversation with the user. This can be accomplished by defining a library of pairs of patterns and responses. For instance, let us assume the following pattern-response pair:

```
pattern: (* my mother *)
response: (Tell me more about your mother ?)
```

The character "*" in the pattern is a *wild card* and matches to anything. The string "my mother" is a constant part of a sentence and the second "*" is also a wild card. Any sentence given by the user containing the words "my mother" would match to the pattern and would trigger the response "Tell me more about your mother?".

In the following pattern-response pair the pattern contains a *variable* (?X) that matches to any single word in a sentence that begins with "I need a":

```
pattern: (I need a ?X)
response: (What would it mean to you if you got a ?X ?)
```

For instance in the sentence "I need a car" the variable ?X is bound to "car". This sentence would trigger the response "What would it mean to you if you got a car?".

The pattern-matching language of PWConstraints is a variation of the main ideas behind the pattern-matching in ELIZA. The input to the pattern-matching part of a PWConstraints rule comes from the search-engine that is systematically producing new partial solutions.

5.1.3.3 Pattern-Matching Syntax

We start by giving the complete syntax of our pattern-matching language for the pattern-matching part and for the Lisp-code part of a PWConstraints rule:

1. ELIZA was developed in the 60s by Joseph Weizenbaum at MIT. In ELIZA the computer imitates a psychotherapist. The user, in turn, is the patient and types sentences in plain English. ELIZA tries to respond to this input in a reasonable way. For details see Norvig (1992:151).

```
Pattern-matching part:
?1          = variable
?           = anonymous-variable
*           = wild card
i1          = index-variable

Lisp-code part:
(?if <test>)  = begins a Lisp expression
l             = partial solution
rl            = reversed partial solution
len           = length of the partial solution
```

Fig.5.11: The pattern-matching syntax in PWConstraints.

5.1.3.3.1 Variable

A *variable* name begins always with the character "?" and continues with a string that can contain numbers or letters or both. Thus ?1, ?2, ?X, and ?Foo1 are all valid variable names, whereas X?1 and ? are not, because the first one does not begin with the character "?" and the second one contains only the character "?". Variable names are unique and should be mentioned in the pattern-matching part of the rule only once. So the expression (?1 ?2 ?3 <Lisp-code>) is a valid rule because it contains each variable name only once, whereas (?1 ?2 ?1 ?3 <Lisp- code>) is not correct, because the variable ?1 is mentioned twice.

5.1.3.3.2 Anonymous-Variable

An *anonymou*s-variable is given only by the character "?". It can appear in the pattern-matching part several times. The difference between an anonymous-variable and a variable is the following: when a rule contains variables, each variable is bound to a unique value and the name of the variable can be used in the Lisp-code part, whereas an anonymous-variable is never bound to a value i.e. it only acts as a "place-holder" in the pattern.

Let us examine two concrete examples. We assume that the search-engine has produced a partial solution consisting of the list (1 2 3 4 5). To define a rule that accesses the two last values of the list, we write the following expression: (? ? ? ?1 ?2 <Lisp-code>). It will bind ?1 to 4 and ?2 to 5 (figure 5.12). The three anonymous-variables are used only to specify the place of the two last values, otherwise we are not interested about their bindings.

```
input:   (1  2  3  4  5)
pattern: (?  ?  ? ?1 ?2)

match:    ?1 = 4, ?2 = 5
```

Fig.5.12: Pattern-matching of `(1 2 3 4 5)` and `(? ? ? ?1 ?2)`.

In the second example we access the first and the fourth value of the list (1 2 3 4 5) with the following rule: (?1 ? ? ?2 ? <Lisp-code>). This rule binds ?1 to 1 and ?2 to 4 (figure 5.13):

```
input:   ( 1  2  3  4  5)
pattern: (?1  ?  ? ?2  ?)

match:    ?1 = 1, ?2 = 4
```

Fig.5.13: Pattern-matching of `(1 2 3 4 5)` and `(?1 ? ? ?2 ?)`.

5.1.3.3.3 Wild Card

A *wild card* is always represented by the character "*". It can match to any continuous part of a list, this part can also be an empty list or (). It is important to note that a pattern can contain *only one* wild card at a time. This is in contrast with many other pattern-matching languages that allow several wild cards.

A wild card is typically used in combination with variables. For example if we have a rule, `(* ?1 <Lisp-code>)`, and match it with a list `(1 2 3 4 5)`, the wild card part of the list is `(1 2 3 4)` and `?1` is bound to `5` (figure 5.14). This means that the variable `?1` will always be bound to the last value of a given list independent of the length of the list.

```
input:   (1  2  3  4  5)
pattern: (*           ?1)

match:    * = (1  2  3  4), ?1 = 5
```

Fig.5.14: Pattern-matching of `(1 2 3 4 5)` and `(* ?1)`.

If the rule is written as (* ?1 ?2 <Lisp-code>), then ?2 is bound to the last item of a list and ?1 to the second to last item. So for instance matching (* ?1 ?2 <Lisp-code>) with (1 2 3 4) will produce the following result: * = (1 2), ?1 = 3 and ?2 = 4 (figure 5.15):

```
input:   (1  2  3  4)
pattern: (*     ?1 ?2)

match:    * = (1  2), ?1 = 3, ?2 = 4
```

Fig.5.15: Pattern-matching of (1 2 3 4) and (* ?1 ?2).

If we try to match (* ?1 ?2 <Lisp-code>) with the list (1), we have a problem because there are not enough items in the list to bind both ?1 and ?2. In such cases PWConstraints never runs the Lisp-code part of a rule, because all variables in the pattern-matching part are not bound. This is a convenient feature because the pattern-matching part of a rule not only helps us to access useful information from a partial solution, but also decides *when* a test function will run.

5.1.3.3.4 Index-Variable

An *index-variable* name begins with the character "i". The second part of the name is a number indicating the *absolute* position of the index-variable in a given list (we start counting from one). Let us assume that we are interested in accessing the fifth, the seventh and tenth item of a list. In this case we write a rule (i5 i7 i10 <Lisp-code>). We can of course write a similar rule using anonymous-variables, (? ? ? ? ?1 ? ?2 ? ? ?3 <Lisp-code>), but this is not as compact and clear as the example using index-variables.

The pattern-matching part should not mix index-variables with variables, `anonymous-variables` or wild card.

5.1.3.3.5 Lisp-Code Part

Next we discuss the pattern-matching syntax of the Lisp-code part of a rule (see figure 5.11 above). Inside the Lisp-code we can use the variable names that were bound in the pattern-matching part. Also the Lisp-code part can use three special reserved variables called l, rl and len.[1] 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. l and rl can always be accessed inside the Lisp-code part independent of the pattern matching part of the rule. len gives the length of the partial solution. The expression (if? <test>) is used to begin the Lisp-code part of the rule.

5.1.3.4 PWConstraints Pattern-Matching Compared with Other Pattern-Matchers

Our syntax may seem somewhat limited compared with other pattern-matching algorithms. For instance many implementations allow several wild cards in a pattern-matching expression. There are two main reasons for our limited or "minimal" syntax.

Firstly, we have to be very concerned about efficiency as the rules are typically called very often. This means that the pattern-matching part of a rule should be as fast as possible. Pattern-matchers that allow several wild cards are typically written as recursive Lisp functions and are too slow for our purposes. The pattern-matching part of a PWConstraints rule, however, can be compiled very efficiently.[1]

Secondly, we should have a clear and compact way of writing rules. Our main aim is to capture the most "obvious" cases in the pattern-matching part of a rule.[2] Typical pattern-matchers deal with a static stream of data (consisting for example of sentences, strings, symbols, numbers). In PWConstraints the situation is more complex - we have to match partial solutions that are constantly shrinking or expanding. Allowing multiple wild cards in such a situation, mixing index-variables with other variables and wild cards, etc., is likely to lead to a confusing system: the pattern-matching part may become extremely difficult to understand and to control.

Where we are not able to solve a special case with our limited syntax in the pattern-matching part, we can always refer in the Lisp-code part to the special variables l and rl. These allow us to write any appropriate Lisp code to extract the required information.

5.1.3.5 Rule Examples

Let us first define a PWConstraints rule that disallows two adjacent equal values in a result. We write the rule in two steps: (1) the pattern-matching part; (2) the Lisp-code part. First we formulate the rule so that it better meets our requirements: the last two values in a partial solution should not be equal.

The pattern-matching part of the rule can be defined as follows: (* ?1 ?2 <Lisp-code>). This means that we access always the last two adjacent items of a list irrespective of the length of the list. For instance if the partial solution is (60 62), ?1 is bound to 60 and ?2 to 62; if the partial solution is (60), the Lisp-code part will not run, because ?1 is not bound; if the partial solution is (60 62 64), ?1 is bound to 62 and ?2 to 64, etc.

It is sufficient for the rule to check only the last two items of the partial solution, because this list

---

1. The short names l, `rl` and `len` were chosen to make the rules as compact as possible.

1. See section 5.2.2.

2. What is "obvious" in this context is of course a moot point since the main problem - describing a musical result - can be a very complex one. Our strategy is based on experience of writing dozens of rules for a wide range of musical problems.

is built step-by-step and the rule is run each time a new candidate is written at the tail of the partial solution.

Next we define the Lisp-code part. It must always start with the expression (?if <test>). After running the pattern-matching part, the rule is able to access the values of both ?1 and ?2. The Lisp-code part is a test function that returns either true or nil. If it returns true, the rule succeeds and the partial solution is accepted. If it returns nil then the rule fails and the last value of the partial solution will be rejected.

We write the Lisp-code part of the rule, which is as follows: (?if (/= ?1 ?2)). It simply means that ?1 and ?2 should not be equal.

The final step is to combine the pattern-matching part with the Lisp-code part to obtain the complete rule:

```
(* ?1 ?2 (?if (/= ?1 ?2)) "No adjacent notes equal")
```

Let us examine another rule that disallows duplicates in the result:

```
(* ?1 (?if (not (member ?1 (rest rl)))) "No duplicates")
```

The pattern-matching part is (* ?1 <Lisp-code>), meaning that ?1 is bound to the last item of the partial solution. Thus, for instance, if this list is (1 2 3 4), then ?1 is bound to 4.

The Lisp-code part is given by the expression: (?if (not (member ?1 (rest rl)))). We check if ?1 is *not* a member of (rest rl). If this is true, we have no duplicates of ?1 in the partial solution and the rule succeeds. rl gives the partial solution in reversed order. For instance, if the partial solution is (1 2 3 4), then ?1 is 4, rl is (4 3 2 1) and (rest rl) is (3 2 1). Next we check if 4 is *not* a member in the list (3 2 1), which returns true.

It is sufficient to ask if the last candidate is not a member in (rest rl), because the partial solution is built step-by-step and the rule is run for each new partial solution.

For our final example let us consider a rule which requires that neither of the first two items of a result list should be duplicated in the rest of the list. The rule is written as follows:

```
(?1 ?2 * ?3 (?if (and (/= ?1 ?3) (/= ?2 ?3)))
  "?1 and ?2 differ from the rest").
```

`?1` and `?2` are the first and second items in a partial solution, as they are found *before* the wild card. `?3` is always the last item whenever the length of the partial solution exceeds two. In the Lisp-code part we check that `?3` is different from `?1` and `?2`.

5.1.4 Classical Constraint Examples

Next we try out our tools with three classical constraint examples. First we define a Lisp function to run the search-engine. Let us call it PMC (short for "Pattern Matching Constraints"):

```
(defun PMC (search-space rules
            &key (sols-mode :once) (rnd? nil) (print-fl ()) ...)
```

PMC has two required arguments. search-space consists of a list of domains for each search-variable. rules is a list of rules. The first keyword argument is used to specify if we want all solutions (:sols-mode :all) or only one solution (:sols-mode :once). :once is the default case. The keyword :sols-mode can also be a positive integer giving the number of desired solutions. The second keyword argument, rnd?, is a flag indicating whether or not the search-space is randomly reordered (by default rnd? is nil). Also the third keyword argument, print-fl, is a flag. If it is true then the index of the current search-variable is printed indicating how far the search has proceeded.

PMC first makes an instance of a search-engine and then starts the search.[1] After the search is completed, PMC returns a list of solutions.

5.1.4.1 A Cartesian Product Example

If we run the search-engine without any rules and ask for all solutions, PMC returns the Cartesian product of a given search-space:

```
? (PMC  '((a b c) (a b c) (a b c))
        ()
        :sols-mode :all)  =>

((a a a) (a a b) (a a c) (a b a) (a b b) (a b c) (a c a)
 (a c b) (a c c) (b a a) (b a b) (b a c) (b b a) (b b b)
 (b b c) (b c a) (b c b) (b c c) (c a a) (c a b) (c a c)
 (c b a) (c b b) (c b c) (c c a) (c c b) (c c c))
```

5.1.4.2 Subset Indices

Subsets have many interesting musical applications. Let us look at an example in which we want to find all 3-member subsets of the list (0 1 2 3 4).[2]

One way to solve this problem is to use indices: the first possible 3-member subset is found by taking the first, the second and the third item from the superset. This subset can be represented by a list of indices (0 1 2) (counting from zero). The next choice is found by incrementing the last index, which produces (0 1 3) and (0 1 4). Now the last index cannot be incremented anymore because 4 already points to the last element of the list. We therefore increment the second index to obtain (0 2 3) and (0 2 4). We again increment the second index and get (0 3 4). Next we increment the first index, which gives us (1 2 3) and (1 2 4). We increment the second index to get (1 3 4). The last possible solution, (2 3 4), is found by incrementing the first index again.

To formulate this algorithm as a constraint problem let us first define the search-space: there are three indices (search-variables) called I1, I2 and I3. Each index is in the range from 0 to 4. Thus,

---

1. The actual implementation of the search-engine is discussed in section 5.2.1.

2. In the general case, the *number of subsets* can be counted by the formula *n!/(m!*(n -m)!)*, where *n* is the size of the superset and *m* is the size of the subset. In our example the number of subsets is 5!/3!*(5-3)! = 10.

the search-space is defined as ((0 1 2 3 4) (0 1 2 3 4) (0 1 2 3 4)), where each sublist represents the domain of a given search-variable (figure 5.16):

```
I1  I2  I3

0   0   0
1   1   1
2   2   2
3   3   3
4   4   4
```

Fig.5.16: The search-space of the 3-member subset indices.

The Cartesian product of this search-space produces lists such as (0 0 0), (0 1 1), (0 1 0), etc. which we are not looking for. This is because there should be no duplicates in the solution, i.e. we should never index an item in the superset more than once. Also we should avoid redundant solutions like (0 1 2) and (0 2 1) which point to the same subset.[1] All these unwanted cases can be avoided by observing that all solutions to our problem are in strict ascending order. We therefore write the following rule:

```
(* (?if (apply #'< l)) "Result in ascending order")
```

`l` found in the Lisp-code part is bound to the partial solution. The expression `(apply #'< l)` returns `true` only if the items in `l` are in ascending order.

Finally, let us run the search-engine:

```
? (PMC   '((0 1 2 3 4) (0 1 2 3 4) (0 1 2 3 4))
         '((* (?if (apply #'< l)) "Result in ascending order"))
         :sols-mode :all)  =>

((0 1 2) (0 1 3) (0 1 4) (0 2 3) (0 2 4)
 (0 3 4) (1 2 3) (1 2 4) (1 3 4) (2 3 4))
```

5.1.4.3 All Permutations

Permutations give all possible orderings of a list. For instance, let us produce all permutations of the list `(0 1 4 6)`.[2]

We find the permutations of `(0 1 4 6)` by first defining the search-space. There are four search-variables (`N1`, `N2`, `N3` and `N4`) and each search-variable has the domain `(0 1 4 6)` (figure 5.17):

1. Note that we are not interested in the order of the items in a subset.

2. In the general case, the *number of permutations* is given by *n!* where *n* is the length of the list. In our example the length is 4 so we will get 1*2*3*4 = 24 results.

```
N1  N2  N3  N4

0   0   0   0
1   1   1   1
4   4   4   4
6   6   6   6
```

Fig.5.17: The search-space of the list (0 1 4 6).

The search-engine returns the Cartesian product, but to get the desired result we have to add a rule that disallows duplicates in the result. This algorithm works only for sets, i.e. the list should not contain duplicates. The "No duplicates" rule was explained earlier in the text, so we just restate it here:

```
(* ?1 (?if (not (member ?1 (rest rl)))) "No duplicates")

? (PMC   '((0 1 4 6) (0 1 4 6) (0 1 4 6) (0 1 4 6))
  '((* ?1 (?if (not (member ?1 (rest rl)))) "No duplicates"))
         :sols-mode :all)  =>

((0 1 4 6) (0 1 6 4) (0 4 1 6) (0 4 6 1) (0 6 1 4)
 (0 6 4 1) (1 0 4 6) (1 0 6 4) (1 4 0 6) (1 4 6 0)
 (1 6 0 4) (1 6 4 0) (4 0 1 6) (4 0 6 1) (4 1 0 6)
 (4 1 6 0) (4 6 0 1) (4 6 1 0) (6 0 1 4) (6 0 4 1)
 (6 1 0 4) (6 1 4 0) (6 4 0 1) (6 4 1 0))
```

All of our examples (Cartesian product, subset indices and all permutations) can be programmed in a normal procedural or functional way, although it is probably not easy to find algorithms that are equally elegant and simple as we found here using the search-engine. What is more important though is that both subsets and permutations are examples of problems where the amount of solutions grows extremely fast as the problem gets larger: for instance, to take all the 6-member subsets of a set of 50 items will produce 15890700 results; to make all permutations of a 12-element set will produce 479001600 results, etc. In procedural or functional programming there is no natural way to filter the search-space when the results are generated by the algorithm. Contrastingly, in constraint languages the user can reduce a large search-space simply by adding new rules. These rules are typically able to make choices already from partial solutions given by the search-engine.

5.1.5 Musical Examples

5.1.5.1 Constraining Chords[1]

Let us assume the task of calculating all possible 12-note chords where each chord is constrained by two rules: (1) the possible simultaneous intervals between adjacent notes (or *adjacent note intervals)* are given by a list of allowed intervals; (2) no pitch-class duplicates are allowed.[2]

1. Simultaneous note collections are henceforth called "chords".

We start by defining the search-space for 12 notes. Each note has a domain consisting of the whole piano register or in MIDI-values in the range from `21` to `108` (figure 5.18 below shows the beginning of this idea). But this approach would produce a truly large search-space.

```
N1   N2   N3   N4   N5   N6   N7   N8   N9   N10   N11   N12

21   21   21   21   21   21   21   21   21   21    21    21
22   22   22   22   22   22   22   22   22   22    22    22
..   ..   ..   ..   ..   ..   ..   ..   ..   ..    ..    ..
..   ..   ..   ..   ..   ..   ..   ..   ..   ..    ..    ..
```

Fig.5.18: The search-space of a 12-note chord.

A more sensible solution is to realise that the chord will have "gaps" or "holes" in the search-space depending on the given adjacent note intervals. For instance, let us assume that the MIDI-value of the first note is `24` and the allowed adjacent note intervals are `5` and `6`.[1] This means that the next MIDI-value of the chord has to be between `24+5 = 29` and `24+6 = 30`, i.e. the only values that we have to consider for the second note are `29` and `30`. We continue the same reasoning for the third note. We know that the *lowest* MIDI-value of the second note is `29` and we add to this number the *smallest* interval, `5`, that gives us the low limit of the third note `29+5 = 34`, etc. In this manner we can reduce drastically the search-space.

Figure 5.19 shows the reduced search-space where the allowed adjacent note intervals are `(5 6)`.

```
N1   N2   N3   N4   N5   N6   N7   N8   N9   N10   N11   N12

24   29   34   39   44   49   54   59   64   69    74    79
     30   35   40   45   50   55   60   65   70    75    80
          36   41   46   51   56   61   66   71    76    81
               42   47   52   57   62   67   72    77    82
                    48   53   58   63   68   73    78    83
                         54   59   64   69   74    79    84
                              60   65   70   75    80    85
                                   66   71   76    81    86
                                        72   77    82    87
                                             78    83    88
                                                   84    89
                                                         90
```

Fig.5.19: The reduced search-space of a 12-note chord for the adjacent note intervals `(5 6)`.

We define a Lisp function, ***make-note-*ranges**, to obtain the reduction:

```
(defun make-note-ranges (start ints card) ...)
```

2. This example and the examples in sections 5.1.5.4 and 5.1.5.6 were suggested to the author by Paavo Heininen.

1. Intervals are given as semitones.

`start` is a MIDI-value for the first note, `ints` is the list of allowed adjacent note intervals and `card` gives the number of notes of the chord. `make-note-ranges` returns a list of lists, each sublist giving the possible MIDI-values for each note.

Next we define two rules. The first one disallows pitch-class duplicates in the chord:

```
(* ?1 (?if (not (member (mod ?1 12) (rest rl)
                        :key #'(lambda (n) (mod n 12)))))
        "No pitch class duplicates")
```

Each time we have a new value we check that its pitch-class, `(mod ?1 12)`, is not a member of the list of pitch-classes of the previous values.[1]

The second rule performs the following check: each time we get a new pair of values, the interval between them, `(- ?2 ?1)`, should be a member of the allowed adjacent note intervals `(5 6)`:

```
(* ?1 ?2 (?if (member (- ?2 ?1) '(5 6)))) "Interval rule")
```

Finally, we run `PMC` (the search-space is calculated by the `make-note-ranges` function) and ask for all possible 12-note chords with adjacent note intervals `(5 6)`:

```
? (PMC  (make-note-ranges 24 '(5 6) 12)
       '((* ?1 (?if (not (member (mod ?1 12) (rest rl)
                            :key #'(lambda (n) (mod n 12)))))
             "No pitch class duplicates")
         (* ?1 ?2 (?if (member (- ?2 ?1) '(5 6))) "Interval rule"))
       :sols-mode :all)   =>

((24 30 35 41 46 52 57 63 68 74 79 85)
 (24 29 34 40 45 51 56 62 67 73 78 83)
 (24 29 34 39 44 50 55 61 66 71 76 81)
 (24 29 34 39 44 49 54 59 64 69 74 79))
```

5.1.5.2 A PC-Set-Theoretical Example[2]

The notion of *chains* is found in Morris (1983:432) and (1987:90).[3] Suppose that we have a list of pitch-classes. We group it into sublists where the size of each sublist is given by *n*. Each sublist

---

1. Note the partial solution consists of MIDI-values.

2. We will not explain any standard pitch-class set-theoretical concepts that are used in this study. Interested readers can refer for instance to Forte (1973), Straus (1990) or Castrén (1989). We use in this chapter the Tn-classification (Morris 1987:78). The set-class names are from Forte (1973), and the extensions "a" and "b", which allow us to distinguish between inversionally related set-classes, are from Castrén (1994). At times we give the prime form of a set-class after the set-class name. The prime form is written in brackets, the pitch-classes being separated by commas. For instance:

4-1 [0,1,2,3].

should overlap, so that the last $n/2$ pitch-classes of a given sublist are also found at the beginning of the next sublist.[1] For example, let us assume that we have a list of 12 pitch-classes and that n is 6:

```
(n1 n2 n3 n4 n5 n6 n7 n8 n9 n10 n11 n12)
```

The list is grouped to the following sublists:

```
((n1 n2 n3 n4 n5 n6) (n4 n5 n6 n7 n8 n9) (n7 n8 n9 n10 n11 n12))
```

We will constrain our problem so that the set-class identity of each sublist should belong to a given list of set-classes. Also, if a set-class is found several times, all instances should be different member sets of that set-class.

To implement this rule we need several new Lisp functions. First we define a function that groups a given list to sublists, called ***`group-to-chain`***:

```
(defun group-to-chain (l card) ...)
```

`l is the list to be grouped and card is the size of the sublists.`[2]

```
? (group-to-chain '(1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18
                    19 20 21 22) 6)  =>

((1 2 3 4 5 6) (4 5 6 7 8 9) (7 8 9 10 11 12) (10 11 12 13 14 15)
 (13 14 15 16 17 18) (16 17 18 19 20 21) (19 20 21 22))
```
[3]

Next, we need a Lisp function that determines the set-class identity of a list of MIDI-values:

```
(defun SC-name (midis) ...)

? (SC-name '(0 1 2 3 4 5)) => 6-1
```

Finally, we define a function called *`check-chain?:`*

---

3. We will not give details of the actual chains algorithm given in Morris (1987:92) mainly for two reasons. Firstly, we are only concerned in the chains example as a compositional idea, i.e. we want only to describe the end result, not how it is constructed. Secondly, the approach presented by Morris is a typical step-by-step algorithm. As we focus in this chapter on constraint satisfaction problems, the algorithm given by Morris is beyond the scope of our presentation.

1. Morris uses the term "two-partition" to describe the sublists of a chain. Also he uses the term "string" to describe the complete chain. We prefer using the terms "sublist" and "list" as all our functions described below work with lists.

2. We are assuming that the overlapping length is `card/2`. One could make of course variations of this idea by having different overlapping lengths.

3. As can be seen from the result the last sublist can also be shorter than `card`.

```
(defun check-chain? (list card SC-names) ...)
```

`list` is the partial solution given by the search-engine, `card` is the size of the sublists and `SC-names` is a list of set-classes.

`check-chain?` first groups `list` by calling `group-to-chain`. Then it calculates the last groups' set-class by calling the function `SC-name` and checks that this set-class is a member of `SC-names`. Also it checks that we do not find a given set-class several times with identical transposition. The latter check occurs only when the length of the last sublist is equal to `card`. If all checks are true then `check-chain?` returns `true` otherwise `nil`.

To be efficient we should calculate all subset-classes of the allowed set-classes in advance. Having a list of subset-classes allows us determine already from a partial solution of two or three pitches whether it potentially is able to form one of the given set-classes.[1] To accomplish this we define a Lisp function called ***all-subs***:

```
(defun all-subs (set-classes) ...)
```

`all-subs` returns a list of all subset-classes of the argument `set-classes` and removes any duplicates, for instance:

```
(all-subs '(4-z15a 4-z15b))  =>

(3-3a 3-5a 3-7b 3-8b 4-z15a 1-1 0-1 2-1 2-2 2-3 2-4 2-5 2-6 3-3b 3-5b 3-
7a 3-8a 4-z15b)
```

Let us next implement a chain rule where the set-classes `(6-5a 6-5b 6-18a 6-18b)` are to be found in the chain. We first store all subset-classes of the allowed set-classes under the global variable `*subs*`:

```
(defparameter *subs* (all-subs '(6-5a 6-5b 6-18a 6-18b)))
The rule itself is then defined as follows:
(* (?if (check-chain? l 6 *subs*)) "Check chain")
```

There are no variables in the pattern-matching part, because we use only the special variable `l` inside the Lisp-code part.

To make the example more interesting let us add some other rules. First, we design a rule that disallows certain set-classes within a group of three adjacent pitches. The following rule disallows the 3-member set-classes `3-9, 3-11a` and `3-11b`:

1. Another reason why we must use subset-classes is that the last sublist returned from `group-to-chain` may be shorter than `card`.

```
(* ?1 ?2 ?3 (?if (not (eq-SC? '(3-9 3-11a 3-11b) ?1 ?2 ?3)))
    "Disallowed 3-member set-classes")
```

The pattern-matching part extracts always the three last values of a partial solution. These are passed to a function called *eq-SC?* that returns true if the set-class of `?1`, `?2` and `?3` is found in the list (`3-9 3-11a 3-11b`). To disallow such cases we call `eq-SC?` inside `not`. `eq-SC?` is defined as follows:

```
(defun eq-SC? (set-classes &rest midis) ...)
```

In the next example we force certain set-classes to appear rather than to disallow them. We use index-variables to specify the exact positions of the given set-classes. Rules using index-variables in the pattern-match part are called *index-rules*. For instance, if we want one of the following set-classes, `(4-z15a 4-z15b 4-16a 4-16b)`, to appear at the sixth, seventh, tenth and eleventh position we write the following index-rule:

```
(i6 i7 i10 i11 (?if (eq-SC? '(4-z15a 4-z15b 4-16a 4-16b) i6 i7 i10 i11))
    "Permitted sets at indices (6 7 10 11)")
```

Let us complete the chain example by making a 24 element chain with the following rules: (1) the chain is built out of the following 6-member set-classes: `(6-5a 6-18b 6-z38 6-z6 6-z43b 6-z43a 6-z41a 6-z41b 6-z12b 6-z12a 6-5b 6-z36a 6-z17b 6-z17a 6-z3a 6-18a 6-z11b)`;[1] (2) we disallow the following 3-member set-classes `(3-9 3-11a 3-11b)`; (3) we force the set-class `4-1 [0,1,2,3]` to appear at indices `(1 2 4 8)`, `(3 5 7 9)`, `(6 10 11 13)`, `(12 14 17 18)`, `(15 16 19 20)` and `(21 22 23 24)` - this produces in all six index-rules.

We first calculate the subset-classes of the given set-classes:

```
(defparameter *subs*
  (all-subs '(6-5a 6-18b 6-z38 6-z6 6-z43b 6-z43a 6-z41a 6-z41b 6-z12b
              6-z12a 6-5b 6-z36a 6-z17b 6-z17a 6-z3a 6-18a 6-z11b)))
```

The search-space is defined simply by giving each element a list of all possible pitch-classes: `(0 1 2 3 4 5 6 7 8 9 10 11)`. Finally we call `PMC`:

1. The list was found by using a similarity measure called *RECREL* (Castrén 1994). RECREL attempts to measure the degree of similarity for two set-classes by giving a numeric value called *RECREL value*. RECREL values are in the range from 0 to 100. 0 indicates highest degree of similarity whereas 100 indicates highest degree of dissimilarity. All 6-member set-classes used in this example are within the RECREL value limits 0 and 25, inclusive, when compared to the set-class `6-5a`.

```
? (PMC
    (make-list 24 :initial-element '(0 1 2 3 4 5 6 7 8 9 10 11))
   '((* ?1 (?if (check-chain? l 6 *subs*)) "chain rule")
     (* ?1 ?2 ?3 (?if (not (eq-SC? '(3-9 3-11a 3-11b) ?1 ?2 ?3)))
         "Disallowed 3-member set-classes")
     (i1 i2 i4 i8 (?if (eq-SC? '(4-1) i1 i2 i4 i8)) "index rule1")
     (i3 i5 i7 i9 (?if (eq-SC? '(4-1) i3 i5 i7 i9)) "index rule2")
     (i6 i10 i11 i13 (?if (eq-SC? '(4-1) i6 i10 i11 i13)) "index rule3")
    (i12 i14 i17 i18 (?if (eq-SC? '(4-1) i12 i14 i17 i18)) "index rule4")
    (i15 i16 i19 i20 (?if (eq-SC? '(4-1) i15 i16 i19 i20)) "index rule5")
    (i21 i22 i23 i24 (?if (eq-SC? '(4-1) i21 i22 i23 i24)) "index rule6"))
      :sols-mode :once :rnd? t)  =>
(8 9 7 10 4 3 6 7 5 1 0 9 2 8 3 4 10 11 6 5 2 0 11 1)
```

We ask for one solution with a randomly ordered search-space. This means that we can get a different result just by running again `PMC` with the same arguments.

Next we translate the result to MIDI-values and transpose the obtained pitch list to C4 (figure 5.20).[1] Above the staff we find a list of indices. In order to make the indexed 4-member set-classes more noticeable we transpose the notes at indices (`3 5 7 9`) and `(12 14 17 18)` one octave higher and the notes at indices `(6 10 11 13)` one octave lower. All 4-member set-classes at the given indices are analysed (the notes of each 4-member set-class are connected by braces) and we see that they all represent `4-1`.

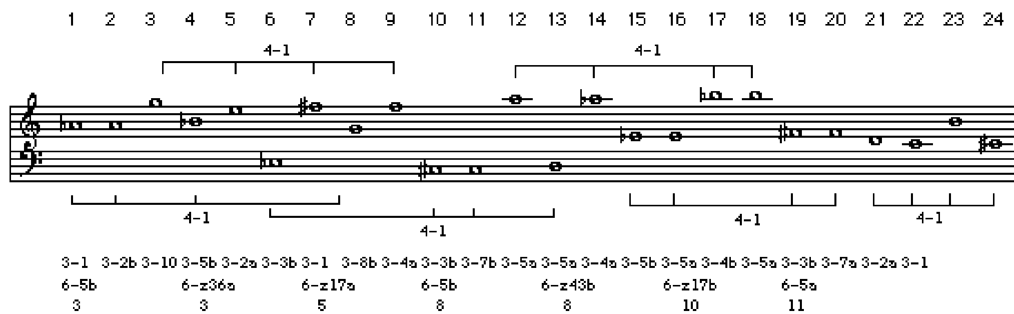


Fig.5.20: 24 element chain with overlapping 4-member set-classes `4-1`.

Below the staff, in the first line, we analyse the 3-member set-classes starting from each note. None of the disallowed set-classes `(3-9 3-11a 3-11b)` is found in this line.

In the second line the sublists of the chain is analysed. We see that all 6-member set-classes belong to the list of allowed chain set-classes. The last line gives the transposition of each 6-member set-class and we do not find a member set twice.[2]

1. The translation is simply done as follows: pitch-class 0 is MIDI-value 0, pitch-class 1 is MIDI-value 1, etc.
2. Each transposition number refers to the transposition of the prime form.

In figures 5.21 and 5.22 we give two other chain examples. We have changed the index-rules so that in the first example we allow only set-class `4-7 [0,1,4,5]` and in the second only set-class `4-27a [0,2,5,8]`.

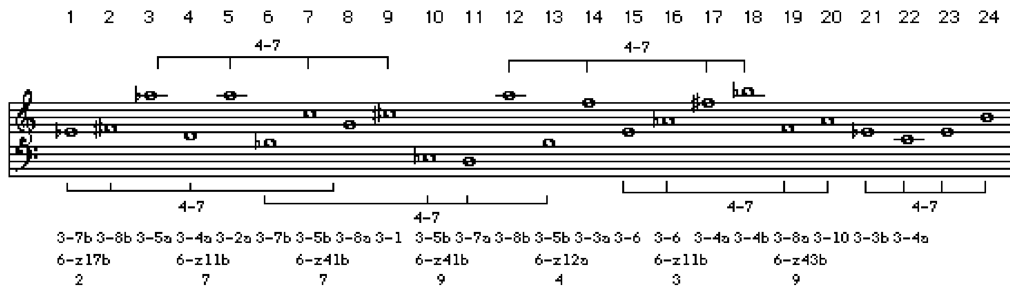


Fig.5.21: 24 element chain with overlapping 4-member set-classes `4-7`.

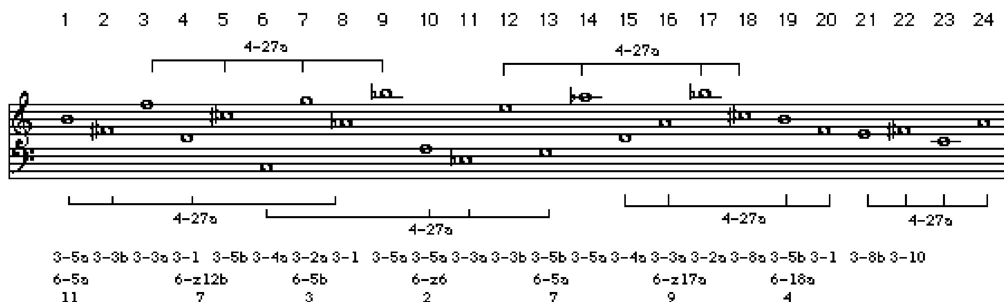


Fig.5.22: 24 element chain with overlapping 4-member set-classes `4-27a`.

5.1.5.3 Subsets

Let us suppose that we want to find all possible subsets from a list of pitches. We constrain the problem so that the set-class identities of the subsets should be members of a list of given set-classes.

We start by defining a list of pitches:

```
(defparameter *pitches*
 '(68 69 79 70 76 51 78 67 77 49 48 81 50 80 63 64 82 83 66 65 62 60 71 61))
```

We are interested in finding all instances of set-classes `4-z15a [0,1,4,6]` and `4-z15b [0,2,5,6]` in `*pitches*`. Like in the previous example we calculate all subset-classes of `4-z15a` and `4-z15b` in advance.

```
(defparameter *subs* (all-subs '(4-z15a 4-z15b)))
```

To be able to translate a list of indices to actual values we define a Lisp function called *inds->vals:*

```
(defun inds->vals (inds ls)
  (mapcar #'(lambda (ind) (nth ind ls)) inds))
```

`inds` is a list of indices and `ls` is a list of values. For instance, if we index the sixth, first and second element of `(3 4 5 6 7 8 9)`, we get the following result (counting from `0`):

```
(inds->vals '(6 1 2) '(3 4 5 6 7 8 9))  =>  (9 4 5)
```

We start by modifying the subset indices example from section 5.1.4.2:[1]

```
? (PMC (make-list 4 :initial-element
         '(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23))
       '((* (?if (apply #'< l)) "Result in ascending order"))
       :sols-mode :all)
```

`PMC` returns all possible indices to the 4-member subsets of `*pitches*`.

Next we add a rule that forces each subset to be either `4-z15a` or `4-z15b`:

```
(*                                                                  ; 1
     (?if (let ((pitches (inds->vals rl *pitches*)))               ; 2
                  (and (not (member (mod (car pitches) 12)          ; 3
                                    (rest pitches)
                              :key #'(lambda (n) (mod n 12))))
                     (member (SC-name pitches) *subs*))))        ; 4
  "no pc-duplicates and given SC")
```

The pattern-matching part of this rule is `(* <Lisp-code>)`, which means that this rule is run for each new value in the partial solution (1). Next we convert indices to `pitches` (2). We check that this list does not contain pitch-class duplicates (3). Then we calculate the set-class of `pitches` and check that it is found in `*subs*` (4). If we were to run `PMC` with the new rule, we would get indices to 384 subsets, each subset being a member either of `4-z15a` or `4-z15b`.

To make the example more interesting let us add two new rules to constrain the indices. For instance, we define a rule that states that the largest gap between two adjacent indices is `3`:[2]

---

1. As the length of `*pitches*` is 24 each domain is defined as a list of indices ranging from 0 to 23.

2. Note that `?1` and `?2` refer to *indices*, not to actual pitches.

```
(* ?1 ?2 (?if (<= (- ?2 ?1) 3)) "largest gap 3")
```

We also decide that the largest allowed gap between two adjacent indices can be found maximally only once. This is accomplished by first calculating all adjacent index gaps. Then we check that the result contains the number `3`, i.e. the largest gap, once:

```
(* ?1 ?2 (?if (<= (count 3 (mapcar #'- (cdr l) l)) 1))
  "maximally one distance of 3")
```

These two new rules will force the indices to be quite near each other. Let us put everything together and run `PMC`:

```
? (PMC  (make-list 4 :initial-element
          '(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23))
           '((* ?1 (?if (apply #'< l)) "asc-order")
             (* ?1 ?2 (?if (<= (- ?2 ?1) 3)) "largest gap 3")
             (* ?1 ?2 (?if (<= (count 3 (mapcar #'- (cdr l) l)) 1))
                  "maximally one distance of 3")
             (* ?1 (?if (let ((pitches (inds->vals rl *pitches*)))
                  (and (not (member (mod (car pitches) 12) (rest pitches)
                                    :key #'(lambda (n) (mod n 12))))
                          (member (SC-name pitches) *subs*))))
                   "no pc-duplicates and given sc"))
         :sols-mode :all)   =>

((1 2 4 5) (1 3 4 6) (1 4 5 7) (4 6 9 10) (5 6 7 9) (6 8 10 12) (7 8 9 12)
 (7 9 10 11) (8 11 13 14) (9 11 14 16) (10 11 12 13) (11 12 14 17)
 (12 15 16 17) (13 15 16 19) (14 17 19 21) (15 16 17 20) (18 19 20 21))
```

The result is a list of lists of indices. We translate it to actual pitches using `inds->vals`, which produces the following result (figure 5.23):

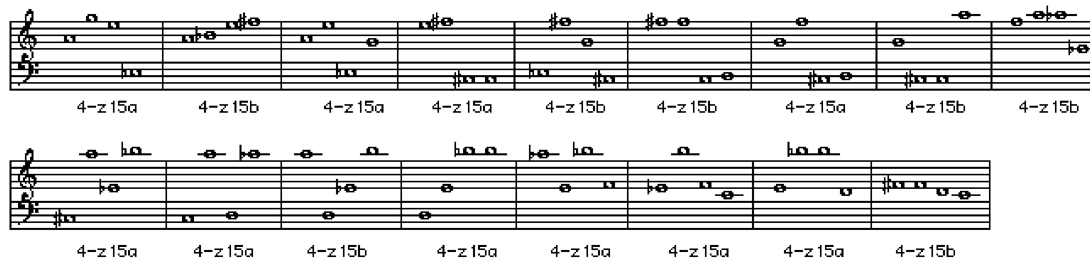


Fig.5.23: The 17 4-member subsets of `*pitches*`.

5.1.5.4 Statistical Distribution and Automatic Rules

Until now we have been able to define rules that simply either accept or reject a candidate. Let us, for instance, consider the following interval rule:

```
(* ?1 ?2 (?if (member (- ?2 ?1) (-1 3 -5 -4 -13 -8 -9 8 -11)))
          "Interval rule")
```

This rule first extracts the last two values of a partial solution. Then we calculate their difference and check if the interval is found in the list of allowed intervals. The problem is that this rule does not state anything about the distribution of the intervals. We have still no way of saying that we would like to have a lot of interval -1, or that interval 6 should be very rare. One solution to this problem is to make statistics of the partial solution and compare this result with the desired distribution.

The following search example is carried out in two steps. First we get the program to derive rules from an existing melodic line. These rules are inferred from the input material through statistical analysis. They are then used to make similar pitch successions by running the search-engine.

Our starting point is the soprano line from Anton Webern's Op. 16. No. 3. We extract only the pitch information in MIDI-values:

```
(68 60 71 66 61 67 72 71 66 65 76 71 63 66 77 75 74 61 70 64 78 73 60
 80 79 82 78 81 72 68 83 82 69 80 79 78 81 70 78 74 65 68 64 75 67)
```

Next we count how often each melodic interval is found in the MIDI-value list. This analysis produces the statistics given in figure 5.24 below. The result is given as a list of lists. Each sublist consists of a *count-interval pair:* we find `7` instances of interval `-1`, `5` instances of interval `3`, `5` instances of interval `-5`, etc.:

```
((7 -1) (5 3) (5 -5) (5 11) (4 -4) (3 -13) (3 -8) (2 -9)
 (1 8) (1 -11) (1 15) (1 20) (1 14) (1 -6) (1 9) (1 -2) (1 5) (1 6))
```

Fig.5.24: Interval statistics of Webern's Op. 16. No. 3.

Let us suppose that we want to create an example that has the same number of notes as the original melodic line has. In this case the rule accepts a partial solution that has *less than or equal to* `7` times interval `-1`, less than or equal to `5` times interval `3`, etc. In other words the count value determines the high limit of instances of a given interval. If, for instance, the interval `-1` is found `8` times or more, then the rule will not accept the partial solution. This rule is probably too strict and time consuming and therefore we add a *tolerance* to the rule. The tolerance indicates how much the high limit can be exceeded without failing. This means that we will get only an approximation of the desired distribution, but giving a tolerance value will speed the calculation considerably.

Next we define a Lisp function called *`check-stats?`*. It is general in the sense that it can be applied to any kind of data found in the original material. This data can consist of intervals, set-classes, etc.

```
(defun check-stats? (item count stats &optional (tolerance 0)) ...)
```

`item` is the data we are making statistics of. `count` indicates how many times `item` is found in the partial solution. `stats` is the statistical distribution of the original input material given as a list of count-interval pairs.

We are now ready to define a rule for interval statistics:

```
(* ?1 ?2
     (?if
      (let ((int (- ?2 ?1)))                                   ; 1
        (check-stats? int
                      (count int (mapcar #'- (cdr l) l))       ; 2
                      '((7 -1) (5 3) (5 -5) (5 11) (4 -4)      ; 3
                        (3 -13) (3 -8) (2 -9) (1 8) (1 -11)
                        (1 15) (1 20) (1 14) (1 -6) (1 9)
                        (1 -2) (1 5) (1 6))
                      1))) "Interval statistics")              ; 4
```

First we extract the two last candidates and calculate the interval between them (1). This interval, int, is the first argument of the function check-stats?. The second argument indicates how many times int is found in the partial solution (2). The third argument is the interval statistics found in figure 5.24 (3). The fourth argument is the tolerance which is 1 (4).

Besides the interval distribution we are also interested in the following statistics: the set-class distribution of 3-member and 4-member set-classes and statistics of four-note melodic movements in register space. The latter distribution is calculated by considering the +- movements of a given note group. For instance the list (+ + -) applied to a four note group means "up-up-down".

It is clear that making these kind of statistics by hand is quite tedious. We can automate the process by first calculating the required statistical information and then producing the Lisp text for the rules algorithmically.

The statistical analysis applied to Webern's Op. 16. No. 3 voice line produces automatically the following Lisp text:

```
(PMC
  (make-list 45 :initial-element
   '(60 61 62 63 64 65 66 67 68 69 70 71
     72 73 74 75 76 77 78 79 80 81 82 83))
  '((* ?1 ?2
       (?if
        (let ((int (- ?2 ?1)))
          (check-stats? int
                        (count int (mapcar #'- (cdr l) l))
                        '((7 -1) (5 3) (5 -5) (5 11) (4 -4)
                          (3 -13) (3 -8) (2 -9) (1 8) (1 -11)
                          (1 15) (1 20) (1 14) (1 -6) (1 9)
                          (1 -2) (1 5) (1 6))
                        1))) "Interval statistics")
    (* ?1 ?2 ?3
       (?if
        (let* ((scs (SC-distribution 3 l))
               (sc (car (last scs))))
          (check-stats? sc
                        (count sc scs)
                        '((7 3-3b) (6 3-1) (5 3-5b) (4 3-3a)
                          (3 3-10) (3 3-2a) (3 3-2b) (3 3-4b)
                          (2 3-4a)(2 3-5a) (1 3-12) (1 3-7b)
                          (1 3-8a) (1 3-11b) (1 3-9))
                        1))) "3-member set-class statistics")
    (* ?1 ?2 ?3 ?4
       (?if
         (let* ((scs (SC-distribution 4 l))
                (sc (car (last scs))))
          (check-stats? sc
```

```
                         (count sc scs)
                         '((5 4-1) (4 4-3) (3 4-12b) (3 4-6)
                           (2 4-19b) (2 4-12a) (2 4-27b) (2 4-2a)
                           (2 4-8) (2 4-5b) (2 4-9) (1 4-7)
                           (1 4-4a) (1 3-3b) (1 4-2b) (1 4-11a)
                           (1 4-16b) (1 4-z15a) (1 4-4b) (1 3-2b)
                           (1 4-z29b) (1 4-14b) (1 4-16a) (1 4-z15b))
                         2))) "4-member set-class statistics")
(* ?1 ?2 ?3 ?4
   (?if
    (let* ((+-lists (+-distribution 3 l))
           (+-list-ref (car (last +-lists))))
      (check-stats? +-list-ref
                         +-list-ref +-lists :test #'equal)
                         (count +-list-ref +-lists :test #'equal)
                         '((13 (- + -)) (9 (- - +)) (9 (+ - -))
                           (5 (+ - +)) (2 (- - -)) (2 (+ + -))
                           (2 (- + +)))
                         2))) "+- statistics"))
:rnd? t)[1]
```

We use in the rules `"3-member set-class statistics"` and `"4-member set-class statistics"` the function *`SC-distribution:`*

```
(defun SC-distribution (card l) ...)
```

`SC-distribution` analyses all set-classes found in `l`, where `l` is grouped in subsequences starting from each successive item. The length of each subsequence is given by `card`. The function `+--distribution` found in the rule `"+- statistics"` is similar, except we analyse in this case the +- movements found in `l`.

Figure 5.25 below shows on the upper staff the pitches of the original voice line. The lower staff, in turn, gives a result obtained by using automatic rules.

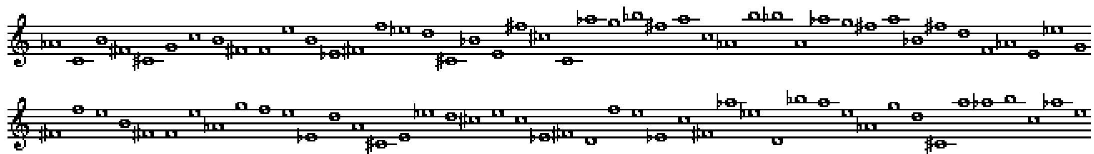


Fig.5.25: The original melodic line and a result produced by automatic rules.

To complete the example we give the interval distribution, the set-class distribution of 3-member and 4-member set-classes and +- statistics of the result (figure 5.26):[2]

1. The tolerance for `"4-member set-class statistics"` and `"+- statistics"` is `2`.
2. The statistics of the original was given in the previous `PMC` expression.

```
Interval distribution:
((8 -1) (6 -5) (6 11) (4 3) (4 -13) (3 -4) (3 -8) (2 20) (1 8)
 (1 -11) (1 14) (1 -6) (1 9) (1 15) (1 -9) (1 -2))

3-member set-class distribution:
((7 3-3b) (7 3-1) (4 3-4b) (4 3-5b) (3 3-2a) (3 3-2b) (3 3-3a) (3 3-5a)
 (2 3-4a) (2 3-10) (1 3-12) (1 3-7b) (1 3-8a) (1 3-11b) (1 3-9))

4-member set-class distribution:
((6 4-3) (5 4-1) (5 4-6) (3 4-2a) (2 4-8) (2 4-16b) (2 4-27b) (2 4-4b)
 (2 4-12b) (2 4-5b) (1 4-19b) (1 3-3b) (1 4-11a) (1 4-9) (1 4-z15b)
 (1 4-16a) (1 4-z15a) (1 4-2b) (1 4-12a) (1 4-z29b) (1 4-14b))

+- movement distribution:
((13 (- + -)) (9 (- - +)) (9 (+ - -)) (5 (+ - +)) (4 (- - -))
 (1 (+ + -)) (1 (- + +)))
```

Fig.5.26: Statistics of the result.

Our example is of course somewhat artificial because we do not consider at all metric aspects, phrase boundaries, etc. of the original melody. We will come back to these problems - i.e. to problems that deal with *musical context* - in the third and fourth part of this chapter. Also we should note that in this example we used *absolute* counts. This means that our example works only if the search problem has exactly the same amount of notes than is found in the original. In the general case the statistical distributions should be defined as *proportional* (for instance as percentage) values. This would allow the number of notes of the search problem to differ from the original.

5.1.5.5 The "At Least" Property

The statistical rules discussed above could be called *at most* rules, because we give in the rules an upper limit of how many instances of some musical property can be found in the final solution. A related problem is how to define in the rules the *at least* property.[1]

One solution to this problem is to insist that the partial solution has the desired properties as early as possible.[2] For instance let us consider the following example. The length of a complete solution (`total-len`) is 5. We want to have at least 3 (`atleast-count`) items with a certain property in the final solution. In this case we can insist that there is at least *one* such item in a partial solution when the length of the partial solution is 3, *two* such items when the length is 4, *three* such items when the length is 5, etc.

Let us call this count as `current-atleast-cnt` and the length of the current partial solution as `current-len`. The value of `current-atleast-cnt` is calculated by the following expression: `(- current-len (- total-len atleast-count))`.

Based on this idea we define a general function, ***`atleast-cnt-check`***, that checks for the at least property:

1. The at least property example was suggested to the author by Camilo Rueda.

2. The trivial solution is of course to check for the at least property only when we have a complete solution, but this approach is very inefficient.

```
(defun atleast-cnt-check
  (l total-len atleast-cnt-item-lst &optional (test #'eq)) ...)
```

`l` is the current partial solution and `total-len` is the length of the final solution. `atleast-cnt-item-lst` is a list (or a list of lists) of count-item pair(s).

For instance, let us suppose the following search example. The search-space consists of 6 search-variables, each having the domain `(0 1 2 3)`. A result should contain at least 3 zeros and at least 2 ones. Also, no adjacent duplicates are allowed in a result:

```
(PMC (make-list 6 :initial-element '(0 1 2 3))
  '((* ?1 (?if (atleast-cnt-check l (cur-slen) '((3 0) (2 1))))
      "at least check")
    (* ?1 ?2 (?if (/= ?1 ?2)) "no adjacent dups"))
 :sols-mode :all)  =>

((3 0 1 0 1 0) (2 0 1 0 1 0) (1 0 3 0 1 0) (1 0 2 0 1 0) (1 0 1 0 3 0)
 (1 0 1 0 2 0) (1 0 1 0 1 0) (0 3 1 0 1 0) (0 3 0 1 0 1) (0 2 1 0 1 0)
 (0 2 0 1 0 1) (0 1 3 0 1 0) (0 1 2 0 1 0) (0 1 0 3 1 0) (0 1 0 3 0 1)
 (0 1 0 2 1 0) (0 1 0 2 0 1) (0 1 0 1 3 0) (0 1 0 1 2 0) (0 1 0 1 0 3)
 (0 1 0 1 0 2) (0 1 0 1 0 1))
```

We use in the `"at least check"` rule the function ***cur-slen*** that returns the number of search-variables in the current search-engine.

5.1.5.6 Splitter

The final example, called *splitter*, is a search problem where we distribute some musical information among two (or more) parts. For example, our starting point can be the following list of pitch-classes:

```
(defparameter *pcs* '(0 1 2 3 6 7 0 1 8 2 3 9 0 7 8 1 2 5 0 6 11 1
                      4 7 2 3 8 1 7 10 0 5 6 7 8 11))
```

We assume that this list should be split into two parts according to some rules given by the user.

Our first problem is how to construct a search-space for the splitter. One simple solution is to take each item from `*pcs*` and say that this item has to belong *either* to the first *or* to the second part. This can be accomplished by defining each value of a domain as a list of two items. This list will be called *partnum-data pair*. The first item in this pair gives the part number, while the second one contains some musical data.

For instance in our particular example the first domain of the search-space is defined as `((1 0) (2 0))`. This means that the first item of `*pcs*`, `0`, belongs *either* to part `1` *or* to part `2`. The second domain, in turn, is `((1 1) (2 1))`. This means that the second item of `*pcs*`, `1`, belongs either to part `1` or to part `2`. The third domain is `((1 2) (2 2))`, etc. We generate the search-space by the following expression:[1]

1. We show due to space limitations only the beginning and the end of the search-space.

```
? (mapcar #'(lambda (n) (list (list 1 n) (list 2 n))) *pcs*)  =>
(((1 0) (2 0)) ((1 1) (2 1)) ((1 2) (2 2)) ((1 3) (2 3)) ((1 6) (2 6))
.....................................................................
 ((1 8) (2 8)) ((1 11) (2 11)))
```

Next, we define four help functions. The first two are simple ones allowing us to extract the part number and the data of a partnum-data pair:

```
(defun partnum (partnum-data) (first partnum-data))
(defun data (partnum-data) (second partnum-data))
```

As the next help function we define the function *setp:*

```
(defun setp (list &key (test #'eq) (key #'identity)) ...)
```

`setp` returns `true` if `list` is a set, i.e. does not contain duplicates.

The fourth help function, *data-group-of-part,* is defined as follows:

```
(defun data-group-of-part (partnum-data-lists group-len partnum) ...)
```

`partnum-data-lists` is a list of partnum-data pairs, `group-len` indicates the length of subgroups and `partnum` is the part number of the current part.

First, `data-group-of-part` collects all partnum-data pairs that belong to the current part to a list (the current part is given by `partnum`). Then this list is grouped into sublists. The length of the sublists is given by `group-len`. Finally, `data-group-of-part` returns the data items of the first sublist.

After this we discuss an example of a splitter rule that forces certain set-classes to appear within adjacent items inside a part. We define the possible set-classes and their subset-classes as a global variable:

```
(defparameter *subs* (all-subs '(4-16b 4-16a 4-14a 4-8 4-6 4-5b)))
```

The splitter rule is defined as follows:

```
(* ?1 (?if
         (let ((pcs (data-group-of-part rl 4 (partnum ?1)))      ; 1
               (if pcs
                  (and (setp pcs) (member (SC-name pcs) *subs*)) ; 2
                   t))) "splitter rule")
```

First, we collect the most recent sublist of data items, `pcs`, using the function `data-group-of-part` (1).[1] The group length of this particular rule is 4, i.e. we are interested only in 4-member set-classes. Finally, we check that `pcs` does not contain duplicates and that its set-class belongs to `*subs*` (2).

After this we define another rule that guarantees us that the result will be balanced in the sense that both parts are forced to alternate after a certain number of pitch-classes. For this rule we need the following help function:

```
(defun count-adjacent-items (list &optional (test #'identity)) ...)
```

count-adjacent-items counts the number of adjacent equal items found at the beginning of `list`. The equality can be defined by the user by giving an optional `test` function.

The rule itself is defined as follows:

```
(* ?1 (?if (let ((max-cnt 3))                         ; 1
             (if (>= (length rl) (1+ max-cnt))
               (<=  (count-adjacent-items rl         ; 2
                      #'(lambda (a b)
                          (= (partnum a) (partnum b))))
                    max-cnt)
               t))) "max 3 adjacent notes in the same part"))
```

`max-cnt` (here `max-cnt` is `3`) indicates the maximum number of adjacent items in a part (1). The function `count-adjacent-items` counts the number of adjacent equal items found in `rl` (2). The equality is defined as a test function that checks if two adjacent items belong to the same part.

Finally we run the splitter:

```
(PMC
  (mapcar #'(lambda (n) (list (list 1 n) (list 2 n))) *pcs*)
  '((* ?1 (?if
          (let ((pcs (data-group-of-part rl 4 (partnum ?1))))
               (if pcs
                  (and (setp pcs) (member (SC-name pcs) *subs*))
                   t))) "splitter rule")
     (* ?1 (?if (let ((max-cnt 3))
                      (if (>= (length rl) (1+ max-cnt))
                        (<=  (count-adjacent-items rl
                               #'(lambda (a b)
                                   (= (partnum a) (partnum b))))
                             max-cnt)
                        t))) "max 3 adjacent notes in the same part"))
   :rnd? t) =>

((2 0) (1 1) (2 2) (2 3) (1 6) (2 7) (1 0) (2 1) (1 8) (1 2) (1 3) (2 9)
 (1 0) (1 7) (2 8) (1 1) (2 2) (1 5) (2 0) (1 6) (1 11) (2 1) (1 4) (2 7)
 (1 2) (1 3) (2 8) (2 1) (2 7) (1 10) (1 0) (1 5) (2 6) (1 7) (1 8) (1 11))
```

The result is given as partnum-data pairs. As in section 5.1.5.2 we translate the pitch-classes of the result to pitches. Figure 5.27 shows the result where the pitches belonging to part `1` are transposed to C3 and the pitches belonging part `2` to C5:

---

1. We collect the most *recent* sublist as we use `rl`.

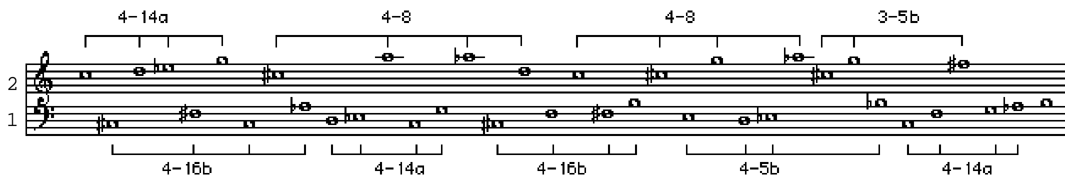


Fig.5.27: A two part split.

Our splitter-rule is general in the sense that it can split into an arbitrary number of parts. For instance we can split `*pcs*` in three parts by defining the search-space with the following expression:

```
(mapcar #'(lambda (n) (list (list 1 n) (list 2 n) (list 3 n))) *pcs*)
```

Also we add an extra rule that guarantees that all three parts are always present inside five successive pitch-classes:

```
(* ?1
  (?if
    (let ((size 5))
       (if (>= (length rl) size)
         (and (>= (count 1 rl :key #'partnum :end size) 1)
              (>= (count 2 rl :key #'partnum :end size) 1)
              (>= (count 3 rl :key #'partnum :end size) 1))
          t))) "at least one instance of each part inside 5 pcs")
```

Figure 5.28 shows a result of the three part split where, after translating the pitch-classes to pitches, part `1` is transposed to C3, part `2` to C4 and part `3` to C5:

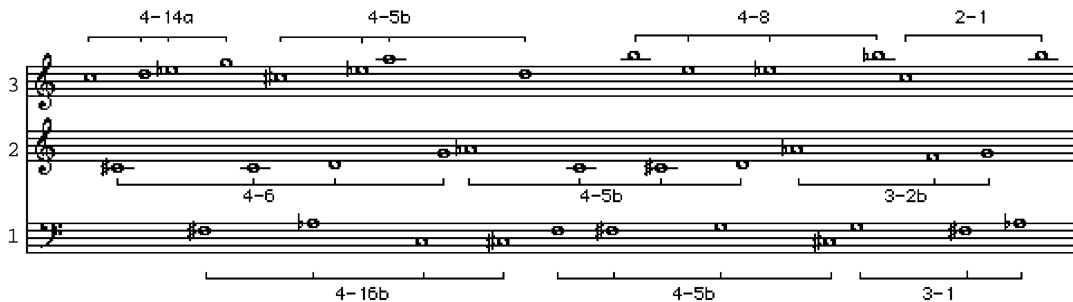


Fig.5.28: A three part split.

5.2 Implementation details and extensions of PWConstraints

The second section of this chapter covers mostly technical aspects of PWConstraints. First we examine in detail the implementation of the search-engine. After this we discuss the compilation scheme of PWConstraints rules. Then we extend PWConstraints by representing partial solutions as CLOS objects instead of simple data structures such as numbers or lists. Another extension follows, introducing the *heuristic rules*, which allow the user to search for "better" results. Next, we interface PWConstraints with two environments. Firstly, we examine the possibility of using the PWConstraints rule scheme in SCREAMER. Secondly, the interface between PW and PWConstraints is studied by translating PWConstraints problems to ordinary Lisp functions. The final issue discussed in this section relates to efficiency: we implement a forward checking scheme to PWConstraints.

5.2.1 Implementation of the Search-Engine

We start our discussion by implementing the search-engine part of PWConstraints. To accomplish this we define two classes, *search-variable* and *search-engine*.

5.2.1.1 Search-Variable

The search-variable class definition contains three slots:

```
(defclass search-variable ()
  ((domain :initform () :initarg :domain :accessor domain)
   (value :initform () :accessor value)
   (other-values :initform () :accessor other-values)))
```

domain is a list defining the possible values of search-variable. For instance, the example in figure 5.1 has three search-variable objects, each having the domain (60 62 64). value is the current choice from domain and other-values is a list of all other values accepted by the search.

We define three methods for search-variable. The first method, *update-values*, has one argument, accepted-values:

```
(defmethod update-values ((self search-variable) accepted-values)
  (setf (value self) (first accepted-values))
  (setf (other-values self) (rest accepted-values)))
```

accepted-values is a list of values accepted by the search. The update-values method stores the first item of accepted-values in the value slot. The rest of this list is written in the other-values slot.

The second method, *set-init-state*, sets the value and the other-values slots to nil:

```
(defmethod set-init-state ((self search-variable))
  (setf (value self) ())
  (setf (other-values self) ()))
```

The third method, *set-new-forward-state*, updates the current state of search-variable.

```
(defmethod set-new-forward-state ((self search-variable))
  (if (other-values self)
    (progn
      (setf (value self) (first (other-values self)))
      (setf (other-values self) (rest (other-values self)))
      t)
    (progn
      (set-init-state self)
       nil)))
```

If other-values contains values, the first item of other-values is stored in the value slot. The rest of the list is then written back to the other-values slot and subsequently set-new-forward-state returns t. On the other hand, if other-values is nil, set-new-forward-state calls set-init-state and returns nil.

5.2.1.2 Search-Engine

The search-engine class is defined as follows:

```
(defclass search-engine ()
  ((search-variables :initform () :initarg :search-variables
        :accessor search-variables)
   (variable-pos :initform 0 :accessor variable-pos)
   (all-sols :initform () :accessor all-sols)
   (rules :initform nil :initarg :rules :accessor rules)
   (sols-mode :initform :once :initarg :sols-mode :accessor sols-mode)))
```

The search-variable class definition contains five slots. The most important one is search-variables. It is a vector of search-variable objects. variable-pos is an integer indicating the position of the *current search-variable*. all-sols is a list of all complete solutions found so far. rules is a list of compiled PWConstraints rules. sols-mode indicates whether we want to have only the first solution (the default case, given by the keyword :once) or all solutions (given by :all). sols-mode can also be a positive integer. In this case it gives the number of desired solutions.

Next we list a number of simple help methods of search-engine. The first one of these is the method *get-current-variable:*

```
(defmethod get-current-variable ((self search-engine))
  (svref (search-variables self) (variable-pos self)))
```

get-current-variable returns the current search-variable of search-engine. It is found by indexing the search-variables vector with variable-pos.

The current search-variable divides the `search-variables` vector into two parts. The first part consists of search-variables before the current search-variable. Each of them already contain a value in the `value` slot. The second part, in turn, consists of the search-variables following the current search-variable. Their `value` slots are still undefined or `()`. Figure 5.29 below gives an example of a search-engine having eight search-variables. The arrow indicates the current search-variable (`V6`). This means that the search-variables prior to `V6`, i.e. `V1`, `V2`, `V3`, `V4` and `V5`, have already a value. We are in the process of finding a value for `V6`. The `value` slots of `V7` and `V8` are still undefined.[1]

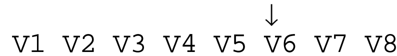


Fig.5.29: A search-engine with eight search-variables.

The next help method, ***`succeed-case?`***, returns true if the current search-variable is equal to the last search-variable of `search-engine`:

```
(defmethod succeed-case? ((self search-engine))
  (= (variable-pos self) (length (search-variables self))))
```

This method is used to check if the current partial solution is a complete one. Let us assume, for instance, a search problem with three search-variables `V1`, `V2` and `V3`. If the partial solution is `(60 62 64)`, then it is also a complete solution because we are pointing to the last search-variable.

The *`fail-case?`* method checks if the `variable-pos` is equal to `-1`. If this is true then the search must stop.

```
(defmethod fail-case? ((self search-engine))
  (= (variable-pos self) -1))
```

The *`step-forwards`* method increments the `variable-pos` slot, i.e. we move forward in the search.

```
(defmethod step-forwards ((self search-engine))
  (incf (variable-pos self)))
```

The *`step-backwards`* method, in turn, decrements the `variable-pos` slot, i.e. we backtrack in the search.

1. This scheme makes it impossible to use some well-known heuristics for variable selection, like *minimum domain-size*. In the latter the search-variables are ordered before search according to domain size, i.e. search-variables with small domains are found at the beginning of the search. The trade-off here is between ease of use (i.e. the pattern-matching rules used by PWConstraints) against efficiency (search-variable re-ordering).

```
(defmethod step-backwards ((self search-engine))
  (decf (variable-pos self)))
```

The final help method, *set-start-position*,, writes in the `variable-pos` slot 0.

```
(defmethod set-start-position ((self search-engine))
  (setf (variable-pos self) 0))
```

Next we explain in more detail the two main methods of the `search-engine` class: *forward* and ***backtrack*.** The `forward` method is called each time we proceed in the search. `forward` calls, in addition to some simple help methods, two methods of importance, *succeed* and *apply-rules*.

```
(defmethod succeed ((self search-engine)) ...)
```

`succeed` appends the current complete solution to the list of solutions stored in the `all-sols` slot. `succeed` also decides whether the search continues or not. It continues if there are not enough solutions in the `all-sols` slot. If this is the case `succeed` calls the `backtrack` method. Otherwise, `succeed` returns the contents of `all-sols`.

```
(defmethod apply-rules ((self search-engine) current-search-variable) ...)
```

The apply-rules method first builds a partial solution. This done by collecting a list of all value slots of the search-variables prior to the current search-variable. The current search-variable is given by the argument current-search-variable. Then apply-rules loops through the domain list of current-search-variable. At each iteration apply-rules appends the current domain value to the tail of the partial solution. The resulting list is called the current partial solution.

Next, apply-rules loops through the list of rules found in the rules slot of search-engine. At each iteration the current rule is called with three arguments: the current partial solution, its reversed version and the length of the current partial solution.

If all rules succeed then we add the current domain value to a list of accepted values. If, however, a rule fails, then we immediately drop the current domain value and go to the next one.

After finishing its loops apply-rules sends update-values to the current search-variable. The accepted-values argument of update-values consists of the list of accepted values.

Finally, apply-rules returns a flag indicating whether or not it found acceptable values for the current search-variable.

The forward method itself is defined as follows:

```
(defmethod forward ((self search-engine))
  (if (succeed-case? self)                                      ; 1
    (succeed self)                                              ; 2
    (let ((search-variable (get-current-variable self)))
      (if (value search-variable)                               ; 3
        (progn (step-forwards self)                          ; 4
      (forward self))
        (if (apply-rules self search-variable)                  ; 5
          (progn (step-forwards self)                         ; 6
      (forward self))
          (progn (set-init-state search-variable)             ; 7
      (step-backwards self)
      (backtrack self)))))))
```

forward calls first succeed-case? (1). If succeed-case? returns true it means that the search has found a complete solution. In this case we call the succeed method (2).

If, however, succeed-case? returns nil, we ask if the current search-variable still has a value (3). If this is the case we proceed with the search. This is accomplished by calling the methods step-forwards and forward (4). This alternative takes place only when forward has been called from the backtrack method.

If the current search-variable has no value we call apply-rules. apply-rules, in turn, returns a flag indicating whether or not it found acceptable values from the current search-variable's domain (5). If the flag is true, we proceed with the search and go on to the next search-variable (6). If, however, the flag is nil, we must backtrack and go to the previous search-variable (7). This is accomplished by calling the methods set-init-state, step-backwards and backtrack.

After this we discuss the second main method of search-engine. The backtrack method is always called when we must go back to a previous search-variable. It is called by forward if apply-rules returns nil, by succeed if there are not enough solutions and by backtrack itself, if the current search-variable does not contain a value. In the latter case we have to backtrack still further.

The backtrack method is defined as follows:

```
(defmethod backtrack ((self search-engine))
  (if (fail-case? self)                                         ; 1
    :fail                                                       ; 2
    (let ((search-variable (get-current-variable self)))
      (if (set-new-forward-state search-variable)               ; 3
        (forward self)                                          ; 4
        (progn (set-init-state search-variable)              ; 5
      (step-backwards self)
      (backtrack self))))))
```

backtrack first checks if we have failed the whole search by calling the fail-case? method (1). If this call returns true then backtrack simply returns the keyword :fail (2), meaning that we stop the search.

If, however, fail-case? returns nil, the search can continue. In this case we send set-new-forward-state to the current search-variable (3).

If set-new-forward-state returns true, it indicates that the current search-variable has still a new value in the other-values slot. We can proceed the search by calling forward (4).

If, on the other hand, `set-new-forward-state` returns `nil`, we must go back to the previous search-variable by calling the methods `set-init-state`, `step-backwards` and `backtrack` (5).

As the last method of `search-engine` we define the *`start`* method. It is called to start a search. It first clears the `all-sols` slot. Then it calls `set-start-position` and starts the search by calling `forward`.

```
(defmethod start ((self search-engine))
  (setf (all-sols self) nil)
  (set-start-position self)
  (forward self))
```

As the final step we define the constructor *`make-search-engine`*. It is typically called by the function `PMC`.[1]

```
(defun make-search-engine
   (search-space rules &optional (sols-mode :once)) ...)
```

`make-search-engine` creates first an instance of `search-engine`. Then it makes instances of `search-variables` and stores them as a vector in the `search-variables` slot. The `search-space` argument, being a list of domains, determines the number of `search-variable` objects. The second argument, `rules`, is a list of PWConstraints rules. These are compiled to Lisp functions and stored in the `rules` slot. Finally, the optional argument, `sols-mode`, is stored in the `sols-mode` slot. `make-search-engine` returns the new `search-engine` instance.

5.2.2 Pattern-Matching Compiler

Next we discuss the compilation of PWConstraints rules. This is done by converting a rule to an ordinary Lisp function. This is a crucial step, as the rules are typically called by the search-engine very often. They should therefore be as efficient as possible.

The resulting Lisp function will always have three arguments: the partial solution, `l`, the partial solution in reversed order, `rl`, and the length of the partial solution, `len`.

The compilation is done by the function ***`compile-pattern-matching-rule:`***

```
(defun compile-pattern-matching-rule (rule) ...)
```

The `rule` argument is a PWConstraints rule containing a pattern-matching part, a Lisp-code part and a documentation string (figure 5.8). `compile-pattern-matching-rule` returns a Lisp expression which is then compiled to a Lisp function.

The translation is done as follows. First, we check if we can call the Lisp-code part of a rule. This can occur only if we can bind the variables and index-variables of the pattern-matching part. If this is the case we perform the actual binding. Finally, we call the Lisp-code part of the rule.

1. `PMC` was defined and its arguments explained in section 5.1.4.

Next we examine how the position of the wild card affects the compilation of a rule. Let us start with a concrete example where the wild card is at the beginning of the pattern-matching part:

```
(* ?1 ?2 (?if (/= ?1 ?2)) "No two equal adjacent notes")
```

To compile this rule we first examine the pattern-matching part. The expression `(* ?1 ?2 <Lisp-code>)` implies that we always access the last two values of a partial solution.

In order to translate this rule we first check if the length of a partial solution, `len`, is less than two. In this case we do not call the Lisp-code part, because the variables `?1` and `?2` cannot be bound. The Lisp function simply returns `t`.

Otherwise, i.e. when the length of a partial solution is equal or greater than two, we bind `?1` and `?2`. The easiest way to do this is to take the reversed partial solution `rl` as our starting point. Next we bind `?2` to the first item and `?1` to the second item of `rl`. We use `rl` because `?1` and `?2` are always found at the tail of a partial solution. For instance, suppose that `rl` is `(5 4 3 2 1)`. In this case we bind `?2` to `5` and `?1` to `4`.

When both `?1` and `?2` are bound, we call the Lisp part of our example. The expression `(/= ?1 ?2)` simply checks if `?1` and `?2` are not equal.

We are now ready to compile the example rule:

```
? (compile-pattern-matching-rule
       '(* ?1 ?2 (?if (/= ?1 ?2)) "No two equal adjacent notes"))  =>

(defun #:g3666 (l rl len)
  "No two equal adjacent notes"
  (if (< len 2)                                      ; 1
    t                                                ; 2
    (let ((?1 (nth 1 rl)) (?2 (nth 0 rl)))           ; 3
       (/= ?1 ?2))))                                 ; 4
```

The name of the function "`#:g3666`" is produced by calling the Lisp function `gensym` that returns a unique symbol. This guarantees that we will not have name conflicts while compiling our rules.

In the code produced by the function `compile-pattern-matching-rule` we first check the length of `l` (1). It should be at least two, otherwise we return `t` (2). If, however, it is two or more, we first bind both `?1` and `?2` (3). Finally we call the Lisp-code part (4).

In the next example, let us change the pattern-matching part of the rule by placing the wild card at the end:

```
? (compile-pattern-matching-rule
       '(?1 ?2 * (?if (/= ?1 ?2))
           "First two notes should not be equal"))  =>

(defun #:g3667 (l rl len)
  "First two notes should not be equal"
  (if (< len 2)
    t
    (let ((?1 (nth 0 l)) (?2 (nth 1 l)))
         (/= ?1 ?2))))
```

In this example we ask for the first two values of a partial solution. This is why we use `l` and not `rl` to access values for `?1` and `?2`.

Finally, let us put the wild card in the middle:

```
? (compile-pattern-matching-rule
       '(?1 * ?2 (?if (/= ?1 ?2)) "First note is unique"))  =>

(defun #:g3668 (l rl len)
  "First note is unique"
  (if (< len 2)
t
     (let ((?1 (nth 0 l)) (?2 (nth 0 rl)))
       (/= ?1 ?2))))
```

In the general case `compile-pattern-matching-rule` extracts variables *before* a wild card from `l` and *after* a wild card from `rl`.

The next example contains anonymous-`variables`:

```
? (compile-pattern-matching-rule
       '(* ?1 ? ? ?2  (?if (/= ?1 ?2))
           "No duplicates at a distance of four notes"))  =>

(defun #:g3671 (l rl len)
  "No duplicates at a distance of four notes"
  (if (< len 4)
    t
    (let ((?1 (nth 3 rl)) (?2 (nth 0 rl)))
        (/= ?1 ?2))))
```

We access always the last four values from `rl` (all variables are behind the wild card). Then we bind `?1` to the fourth item and `?2` to the first item of `rl`. In this case we are not interested of the second and third items.

As the last example let us take a rule using index- variables:

```
? (compile-pattern-matching-rule
       '(i2 i4  (?if (/= i2 i4))
           "Second and fourth note should not be equal"))  =>

(defun #:g3678 (l rl len)
  "Second and fourth note should not be equal"
  (if (not (= len 4))
    t
    (let ((i2 (nth 1 l)) (i4 (nth 3 l)))
      (/= i2 i4))))
```

Here we call the Lisp-code part only if the length of `l` is four. If the length is less than four, then `i2` and `i4` cannot be bound. If, however, the length is greater than four, there is no point of calling the Lisp-code part. This is because the values at indexes `i2` and `i4` have not changed. [1]

5.2.3 Objects in a Partial Solution

Until now the partial solution consisted only of simple data structures like numbers or lists. In complex cases this scheme leads to problems because we are not able to easily access other relevant information during the search. In the musical domain we are typically interested of the rhythmic, voice-leading, harmonic, etc. aspects of each search-variable.

One way to solve this problem is to define the partial solution as a list of search-variable *objects* instead of numbers or lists. Each search-variable object has a value accepted by the rules. It can also potentially contain arbitrary information about its surroundings.

In order to accomplish this modification we first redefine the class definition of search-variable so that we can both write to and read from search-variable objects. Instead of changing the class definition directly let us define a general class called `key-item`. This class can be later inherited by any other class.

```
(defclass key-item ()
  ((prop-list :initform (gensym) :accessor prop-list)))
```

The `key-item` class has one slot, `prop-list`, that contains a symbol generated by the function `gensym`.[1] Next we define two methods, one for writing and one for reading:

```
(defmethod write-key ((self key-item) key data)
  (setf (get (prop-list self) key) data))

(defmethod read-key ((self key-item) key)
  (get (prop-list self) key))
```

Let us make an instance of `key-item`:

```
(defparameter k-item (make-instance 'key-item))
```

We store information into `k-item` by using the `write-key` method:

```
(write-key k-item :foo '(1 2 3))
```

Then we access the information with `read-key`:

```
? (read-key k-item :foo) => (1 2 3)
```

---

1. Our compilation scheme is somewhat inefficient as the actual test function of the index-variable rule is called only at the last index-variable of the pattern-matching part. A better compilation scheme would link these search-variables directly to the index-rules. The search-variables would know which index-variable rules to call. In this case the index-variable rules would be called only when necessary.

1. Inside the class definition of `key-item` we use a property list. Another possibility is to use hash-tables, but property lists seem more efficient for storing only a small amount of data.

The next step is to change the first line of the class definition of search-variable:

```
(defclass search-variable (key-item) ...)
```

Because `key-item` is useful, let us also change the class definition of `search-engine`:

```
(defclass search-engine (key-item) ...)
```

In order to support the `key-item` feature let us define a subclass of `search-engine` and redefine the `apply-rules` method for it:

```
(defclass object-search-engine (search-engine) ...)

(defmethod apply-rules
  ((self object-search-engine) current-search-variable) ...)
```

The `apply-rules` method does not collect the partial solution as a list of values as before. Instead, it collects all search-variable objects prior to `current-search-variable`. Then it appends `current-search-variable` to the tail of this list. While looping through the domain of `current-search-variable`, we write the current domain value into the `value` slot of `current-search-variable`. This enables the rules to access the correct value of `current-search-variable`.

Whenever the partial solution consists of search-variable objects, we must translate the rules discussed in section 5.1 in the following way. Inside the Lisp-code part we replace each variable and index-variable name with the expression `(v <search-variable>)`. *v* (for value) is a macro that returns the contents of the `value` slot:

```
(defmacro v (search-variable) `(value ,search-variable))
```

Let us, for instance, translate the following rule:

```
(* ?1 ?2 (?if (/= ?1 ?2))
    "No two equal adjacent notes")
```

We simply exchange `?1` with `(v ?1)` and `?2` with `(v ?2)`. In this rule both `?1` and `?2` are search-variable objects:

```
(* ?1 ?2 (?if (/= (v ?1) (v ?2)))
    "No two equal adjacent notes")
```

5.2.4 Heuristic Rules

In section 5.2.1.2 above we explained how the search-engine, after calling the rules, collects all accepted values in a list. Because the first items of the accepted values list are always tried out first, the *order* of the list affects the end result. The control of this order would allow us to preference certain results. The final result would not only be correct, but it would also be a "better" one.

We add this feature by implementing a new type of PWConstraints rule, called *heuristic rule*. A heuristic rule is exactly like the standard one, except the Lisp-code part returns always a number,

called *heuristic value*, instead of a truth value. The numbers obtained from the heuristic rules are in turn used to sort the accepted values list. The sort is accomplished so that the accepted values having the largest heuristic value are found at the beginning of the list.

In order to support the heuristic rules, we must redefine the `PMC` function. We add a new keyword argument, `heuristic-rules`, to the argument list. It gives a list of heuristic rules to the search-engine:

```
(defun PMC (search-space rules
             &key (heuristic-rules ()) (sols-mode :once) (rnd? nil)) ...)
```

Let us examine two examples to demonstrate the effect of heuristic rules to an end result. The first example consists of three simple standard PWConstraints rules and of one heuristic rule:[1]

```
(PMC (make-list 24 :initial-element
        '(60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78))
'((* ?1 ?2 (?if (member (abs (- ?2 ?1)) '(1 2 10 11)))
     "allowed intervals")
  (* ?1 ?2 ?3 ?4 ?5 (?if (setp (list ?1 ?2 ?3 ?4 ?5)
                              :key #'(lambda (n) (mod n 12))))
     "no pc duplicates inside five notes")
  (* ?1 ?2 ?3 ?4 (?if (/= (- ?2 ?1) (- ?3 ?2) (- ?4 ?3)))
     "three unique adjacent intervals"))
  :heuristic-rules '((* ?1 ?2 (?if (abs (- ?2 ?1)))
                        "prefer large intervals"))
  :rnd? t)  =>

(68 78 67 65 76 75 73 62 72 70 71 61 63 74 64 66 77 67 69 70 60 71 73 62)
```

The first standard rule, `"allowed intervals"`, controls the adjacent note intervals. We allow only "small" intervals, 1 and 2, or "large" ones, 10 and 11. The second rule, `"no pc duplicates inside five notes"`, checks that we have no pitch-class duplicates within a sequence of five notes. The third rule, `"three unique adjacent intervals"`, checks that we have no interval duplicates within a group of four adjacent notes. The heuristic rule `"prefer large intervals"`, given by the keyword `:heuristic-rules`, simply returns the interval of two adjacent notes. In other words we prefer large intervals. The result of this search problem contains 13 large intervals.

In the second example we will prefer *small* intervals. This is achieved simply by reversing the sign of the interval of the heuristic rule:

```
(* ?1 ?2 (?if (- (abs (- ?2 ?1)))) "prefer small intervals")
```

Now the result contains only 6 large intervals:

1. The result is shown immediately after the `PMC` expression.

```
(74 73 75 76 66 68 67 65 75 76 78 68 67 65 75 74 76 66 65 63 73 74 72 71)
```

5.2.5 SCREAMER Interface

In this section we discuss briefly an interface from PWConstraints to SCREAMER. SCREAMER is an extension of Common Lisp that adds support for constraint programming. SCREAMER could in some cases replace the search-engine part of PWConstraints. This opens up a range of future possibilities because SCREAMER offers an elegant and rich Prolog-like environment inside Common Lisp. We will not give here any detailed description of SCREAMER, but present only the necessary Lisp code with comments.[1]

The interface is defined by two Lisp functions *run-screamer* and *run-screamer-rules*.

```
(defun run-screamer (s-space rules &key (rnd? nil))
  (let ((functions                                                  ; 1
          (mapcar #'(lambda (rule)
            (compile-pattern-matching-rule rule)) rules)))
    (when rnd? (setq s-space (mapcar #'pw::permut-random s-space)))   ; 2
     (local
       (let (collection)
         (dotimes (i (length s-space)) #-mcl (declare (ignore i))
                 (push (a-member-of (pop s-space)) collection)         ; 3
                 (when (not (run-screamer-rules functions collection)) ; 4
                     (fail)))
         (reverse collection)))))
```

run-screamer has three arguments. s-space is the search-space, rules is a list of PWConstraints rules and rnd? indicates whether or not the search-space will be randomly reordered.

run-screamer first compiles all rules to Lisp functions (1). Then it orders the search-space if necessary (2). Inside the dotimes loop we call the *non deterministic* SCREAMER function a-member-of (3). After this, inside when, we call the help function run-screamer-rules (4).

```
(defun run-screamer-rules (functions collection)
 (let ((coll1 (reverse collection))          ; 1
       (fl t) fn)
  (while (and fl (setq fn (pop functions)))   ; 2
    (setq fl (funcall fn coll1 collection)))
  fl))                                        ; 3
```

run-screamer-rules, in turn, has two arguments: functions and collection. functions is a list of compiled rules. collection is a list of partial solutions in reversed order. We first reverse collection to get the partial solution in the "right" order (1). Then we loop through all the rules (2). Finally, run-screamer-rules returns a flag indicating whether we succeeded or not (3).

We give here the SCREAMER version of the example discussed in section 5.1.5.1:

```
(screamer:print-values
  (run-screamer
```

1. We will discuss only the non-deterministic part of SCREAMER. For more details see Siskind and McAllester (1993).

```
      (make-note-ranges 0 '(5 6) 12)
      '((* ?1 (?if (not (member (mod ?1 12) (rest rl)
                                :key #'(lambda (n) (mod n 12))))))
          (* ?1 ?2 (?if (member (- ?2 ?1) '(5 6)))))
        :rnd? t))
```

`screamer:print-values` is a SCREAMER function that returns all solutions to our problem.

5.2.6 PW Interface

One obvious way to interface PWConstraints to PW is to translate PWConstraints expressions to ordinary Lisp functions. For instance, let us take as the starting point the "constraining chords" example discussed in section 5.1.5.1:

```
(PMC (make-note-ranges 24 '(5 6) 12)
       '((* ?1 (?if (not (member (mod ?1 12) (rest rl)
                                 :key #'(lambda (n) (mod n 12)))))
            "No pitch class duplicates")
          (* ?1 ?2 (?if (member (- ?2 ?1) '(5 6))) "Interval rule"))
       :sols-mode :all)
```

This expression is then translated to the Lisp function ***`int-chords,`*** which has three arguments: `start` (the MIDI-value for the first note), `ints` (the list of allowed adjacent note intervals) and `card` (the number of notes of the chords):

```
(defun int-chords (start ints card)
  (eval
    `(PMC (make-note-ranges ,start ',ints ,card)
          '((* ?1 (?if (not (member (mod ?1 12) (rest rl)
                                    :key #'(lambda (n) (mod n 12)))))
               "No pitch class duplicates")
             (* ?1 ?2 (?if (member (- ?2 ?1) ',ints)) "Interval rule"))
          :sols-mode :all)))
```

The translation is done simply by replacing the number 24 with `start`, all occurrences of `(5 6)` with `ints` and the number 12 with `card`.

Now we are ready to call `int-chords` like any other Lisp function:

```
? (int-chords 24 '(5 6) 12)  =>
((24 30 35 41 46 52 57 63 68 74 79 85)
 (24 29 34 40 45 51 56 62 67 73 78 83)
```

```
 (24 29 34 39 44 50 55 61 66 71 76 81)
 (24 29 34 39 44 49 54 59 64 69 74 79))

? (int-chords 24 '(5 6) 8)  =>
((24 30 35 41 46 52 57 63) (24 30 35 41 46 52 57 62)
 (24 30 35 41 46 51 56 62) (24 30 35 41 46 51 56 61)
 (24 30 35 41 46 51 57 62) (24 30 35 40 45 51 56 62)
 (24 30 35 40 45 51 56 61) (24 30 35 40 45 50 55 61)
 (24 30 35 40 45 50 56 61) (24 30 35 40 46 51 56 62)
 (24 30 35 40 46 51 56 61) (24 30 35 40 46 51 57 62)
 (24 29 35 40 45 51 56 62) (24 29 35 40 45 51 56 61)
 (24 29 35 40 45 50 55 61) (24 29 35 40 45 50 56 61)
 (24 29 35 40 46 51 56 62) (24 29 35 40 46 51 56 61)
 (24 29 35 40 46 51 57 62) (24 29 34 40 45 51 56 62)
 (24 29 34 40 45 51 56 61) (24 29 34 40 45 50 55 61)
 (24 29 34 40 45 50 56 61) (24 29 34 39 45 50 55 61)
 (24 29 34 39 45 50 56 61) (24 29 34 39 44 50 55 61)
 (24 29 34 39 44 49 54 59))
```

The problem with this scheme is that we have hardwired the rules inside the Lisp function. If the user wants to change them she/he has to redefine the function. One solution to this problem is to use a standard PW box, `text-win`, that allows the communication with a Lisp text editor window. Figure 5.30 below gives a patch containing the interval chords example. The `text-win` box communicates with a text editor (see arrow `rules`). If we call `patch-value` to the `text-win` box, it returns simply all the rules inside the "`rules.lisp`" window as a list. This scheme allows the user to edit rules dynamically. Rules can also be saved, loaded, etc.

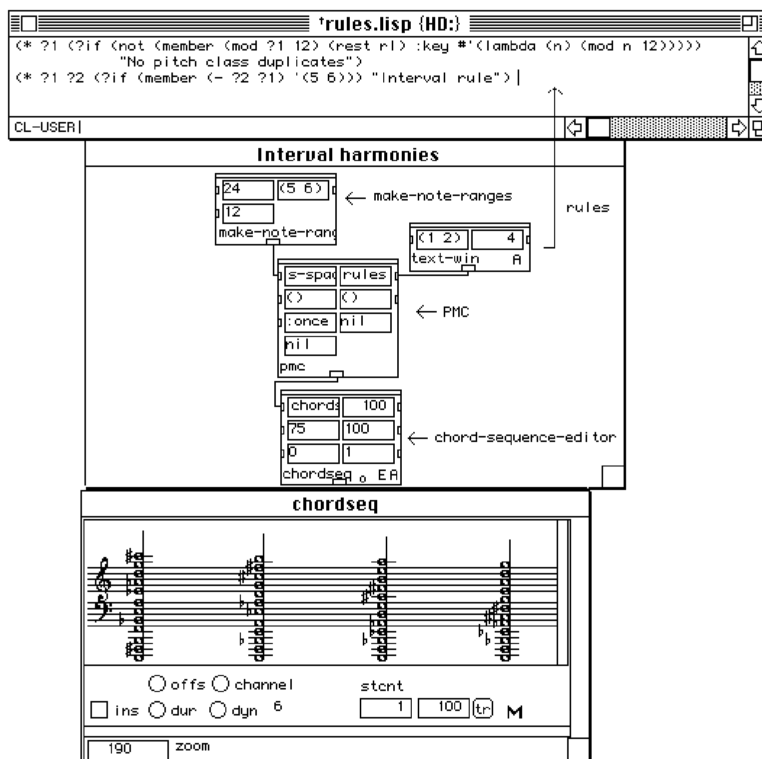


Fig.5.30: The "constraining chords" example in PW.

The `PMC` box performs the actual search (arrow `PMC`).[1] The first input is connected to the `make-note-ranges` box, that calculates the search-space (arrow `make-note-ranges`). The second input, `rules`, is given by the `text-win` box. The output of the `PMC` box is connected to the `chord-sequence-editor` box (arrow `chord-sequence-editor`). The lowest window "`chordseq`" shows the final result.

5.2.7 Efficiency Issues - Forward Checking

5.2.7.1 Graph Representation of a CSP

Until now we have solved all PWConstraints problems using pure (called also chronological or naive) backtrack search. In this section we will discuss why pure backtrack search is sometimes inefficient. We show also how to correct these problems by making our search-engine more intelligent.

One of the central issues in CSP research has been improving the efficiency of pure backtrack search. Most papers assume that a CSP is defined as a *directed graph* in which the search-variables are represented by *nodes*.[2] The *unary* constraints, i.e. one node constraints, are represented by loops on the nodes. *Binary* constraints, or two node constraints, are represented by *directed arcs*.[3]

We start our discussion by showing a small CSP example in graph representation:

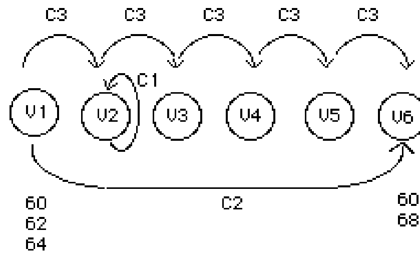


Fig.5.31: A CSP as a graph.

Figure 5.31 gives a CSP with six search-variables with both unary and binary constraints. The search-variables (or nodes) are shown as circles with labels `V1`, `V2`, etc. The constraints, in turn, are given as labelled arcs (`C1`, `C2`, `C3`). Below the respective nodes we show also the domains of `V1, (60 62 64)`, and `V6, (60 68)`.

Let us define a unary constraint, `C1`, for the second search-variable, `V2`, using our syntax:

1. Note that the boxes `PMC` and `make-note-ranges` are exact graphical equivalents of the Lisp functions defined in section 5.1.
2. A general discussion of graph theory can be found in Luger and Stubblefield (1993:78).
3. A more detailed discussion can be found in Mackworth (1977:101).

```
(i2 (?if (/= i2 60)) "The second note should not be 60")
```

The binary constraint, `C2`, between search-variables `V1` and `V6` is defined as follows:

```
(i1 i6 (?if (= (- i6 i1) 6))
  "The interval between the first note and the sixth note should be
   a tritone")
```

As a final example we define a rule that adds a constraint between each adjacent pair of search-variables:[1]

```
(* ?1 ?2 (?if (/= ?1 ?2)) "No adjacent duplicates")
```

As can be seen in figure 5.31, the arcs add knowledge of how a given choice in one search-variable affects other search-variables. For instance, `V1` has a constraint with both `V2` and `V6`. If we assign a value for `V1`, we know that this choice will affect the future choices of `V2` and `V6`. The arcs help us to detect incoherent choices much earlier than is the case with pure backtrack search.

The CSP optimisation techniques discussed in this section are called collectively *consistency inference* techniques. They involve establishing or restoring some form of *arc consistency*.[2] For instance, let us assume that the domain of `V1` is `(60 62 64)` and that of `V6` `(60 68)` (see figure 5.31 above). We say that the arc `C2` is *inconsistent* as the *source domain*, `V1`, contains values that are not *supporting* the constraint `C2`.[3] Values 60 and 64 of `V1` should be eliminated to make `C2` *consistent*.

As another example let us examine a situation where `C2` cannot be made consistent: the domain of `V1` is `(60 61)` and that of `V6` `(60 68)`. In this case, assuming that we use only pure backtrack search, we would start at `V1`, proceed to `V2`, etc. until we reach `V6`. We find no solution for the constraint `C2`. This means that we backtrack to `V5`, select a new value, try `V6` again, etc. All the backtracking is done without realising the actual cause for our failure: `V1` has no values that can support `C2`. Using consistency inference techniques would tell us either very early in the search, or even before the search, that there are no solutions for the problem.

In order to implement consistency inference tools for PWConstraints we will next discuss the following questions: Which consistency inference technique(s) should we choose?[4] How to handle constraints whose *arity* is greater than two?[5] How to translate a PWConstraints problem to a graph?

---

1. Although this rule produces five arcs, we label them collectively C3 because they were produced by a single rule.
2. Sabin and Freuder (1994).
3. `C2` states that the interval between `V1` and `V6` should be 6.
4. Actually this is a very difficult question and it has been discussed in several papers. Probably it is hard to judge pruning techniques because CSP problems behave very differently depending on the search-space, constraints, etc. The results depend very much on the actual CSPs that are used in the tests.
5. The arity of a constraint gives the number of search-variables addressed by the constraint.

5.2.7.2 Choice of Consistency Inference Techniques

Consistency inference techniques can be divided in two main groups depending whether they are applied *before* search (*preprocessing* techniques) or *during* search.[1]

For instance the classical preprocessing technique, *AC-3*, described in Mackworth (1977), aims at pruning all values from a CSP graph that are not arc consistent before search.[2] AC-3 and its followers have been criticised because the added computational cost caused by the additional preprocessing effort may outweigh subsequent savings.[3] Also, it is not very useful in our case because we cannot restrict ourselves to unary and binary constraints. Preprocessing constraints for arities greater than two seems often to be very inefficient.[4]

A more convenient choice for us is a technique used during search called forward checking. Forward checking is considered to be one of the most successful pruning techniques during search.[5] It is a good choice because it is efficient and it can be implemented quite easily in PWConstraints.[6] With it we prune inconsistent values *after* making search choices.

To demonstrate how forward checking works we assume a simplified version of the CSP given in figure 5.31 above.

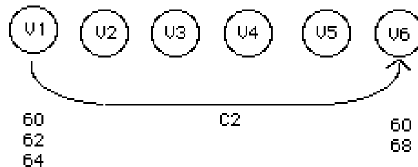


Fig.5.32: A simplified CSP graph.

Figure 5.32 gives a graph consisting of six search-variables and of a single constraint, `C2`, between `V1` and v6.[7] The domain of `V1` is `(60 62 64)` and that of `V6 (60 68)`. We make search choices for `V1` as in pure backtrack search and accept each value of `V1`'s domain. Before we proceed to `V2`, we run the forward checking algorithm that loops through the list of accepted values of `V1`, `(60 62 64)`.

1. Sabin and Freuder (1994).
2. AC-3 has been developed and improved later by several writers.
3. Sabin and Freuder (1994).
4. We will soon come back to the arity problem.
5. Sabin and Freuder (1994).
6. This is true specially for index-rules and for some wild card rules. For some other cases the addition of forward checking is more problematic. We will discuss these problems in section 5.2.7.4 in more detail.
7. As before `C2` states that the interval between `V1` and `V6` should be 6.

We start with 60 and check if we find *at least* one value in the domain of V6 that supports 60. It turns out that none of the domain values of V6 are able to do this. We can safely disregard 60 from the list of accepted values because it will never be able to satisfy the constraint C2. Next we pick 62 and check if it is supported by the domain of V6. In this case we find that there is one value, 68, that supports 62 (62 subtracted from 68 gives 6). This means that we keep 62 in the list of accepted values. Next we check if we find support for 64. It can also be disregarded because there is no support in the domain of V6. After this we order the list of accepted values with the help of the heuristic rules.[1] Once the accepted list is ordered we choose the first one as the current value, which is in our case 62. Next we eliminate all values in the domain V6 that are not consistent with 62. After this operation the domain of V6 is (68).

Now the forward checking algorithm is finished and the list of accepted values of V1 has been reduced from (60 62 64) to (62). Also, after choosing 62 as the current value of V1, the domain of V6 has reduced to (68). This is a considerable improvement compared to pure backtrack search.

If we assume a case where the domain of V6 is (60 61), we would reduce the list of accepted values of V1 from (60 62 64) to (). This is because none of the accepted values can be supported by the domain of V6. This means that the search would stop here. In pure backtrack search we end up with a lot of unnecessary backtracking before we find out that there are no solutions for our problem.

5.2.7.3 N-arity Constraints and Forward Checking

A large number of CSP papers assume that the search-problems use only unary and binary constraints. This is a serious limitation because we often need rules that deal with an arbitrary number of search-variables. We have to find a scheme that allows us to combine efficiently N-arity constraints with the forward checking algorithm. The central question in this context is *when* the forward checking algorithm should be called.

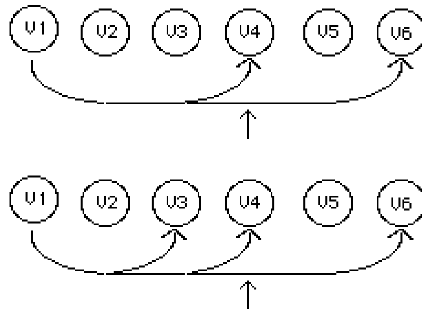


Fig.5.33: Two constraints with arities 3 and 4.

1. This ordering takes place only if the user has given heuristic rules. See for more details section 5.2.4.

In PWConstraints the forward checking algorithm is called at a search-variable situated just before the last one in a N-arity constraint. Figure 5.33 shows two CSPs each having six search-variables. The upper CSP has a 3-arity constraint and the lower one has a 4-arity constraint. In both cases we call the forward checking algorithm at `V4`, that is just before `V6` (see the arrows pointing at `V4`). For instance, in the 3-arity case a value is already chosen for both `V1` and `V4`. The forward checking algorithm checks if there is support in the domain of `V6` for this particular combination of values.

We could also consider an alternative scheme where the forward checking algorithm is called earlier. For instance, let us assume that this takes place at `V1` in the upper CSP of figure 5.33. In this case we would have to check a choice made in `V1` with all possible combinations of the domains of `V4` *and* `V6`. This would result in a new search problem inside the forward checking algorithm. This is probably not a good strategy because it would in all likelihood slow down the forward checking algorithm considerably.

5.2.7.4 Rule Translator

We saw already in section 5.2.7.1 that some PWConstraints rules can be translated to arcs by examining the pattern-matching part. Let us look in a more systematic way how to implement a rule translator and what are its limitations. We start with rules that can be translated automatically.

An index variable rule can always be translated easily to a CSP graph. For instance, in figure 5.34 we translate the index rule `(i1 i4 i6 <Lisp-code>)` as follows:

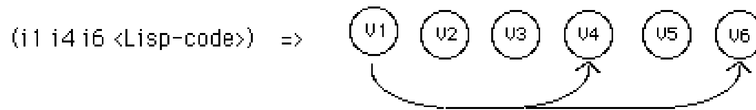


Fig.5.34: Translation of the index variable rule `(i1 i4 i6 <Lisp-code>)`.

A more complex situation occurs if we have a wild card in the pattern-matching part (figure 5.35):

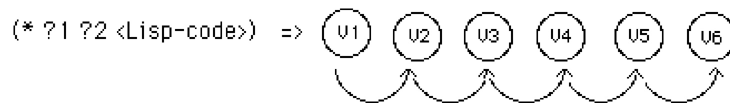


Fig.5.35: Translation of the wild card rule `(* ?1 ?2 <Lisp-code>)`.

The pattern-matching part of this rule indicates that we should constrain every adjacent search-variable pair.

Figure 5.36 below gives another wild card rule where we constrain the first search-variable with all other search-variables.

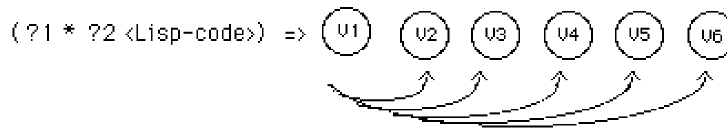


Fig.5.36: Translation of the wild card rule `(?1 * ?2 <Lisp-code>)`.

The situation is more problematic if we use the special variables `l` and `rl` inside the Lisp-code part. For instance, let us consider the rule that disallows duplicates:

```
(* ?1 (?if (not (member ?1 (rest rl))) "no duplicates")
```

In this case we cannot infer the arcs automatically from the pattern-matching part, but have to treat these rules as special cases. For instance, this particular rule should be translated so that the graph is fully connected, i.e. each search-variable has a binary constraint with all other search-variables.[1]

Still more problematic are cases where a rule is not able to know in advance to which and to how many search-variables a given search-variable should be connected. This situation occurs during search when we constrain search-variables dynamically depending on the partial solution.[2]

To summarise, there are three main cases when translating rules: (1) automatic translation (we infer the arcs from the pattern-matching part);[3] (2) manual translation; (3) rules that cannot be translated to arcs in advance.

Next, we redefine the `PMC` function so that it supports forward checking. This is done by adding one optional argument, `fwc-rules` (for forward checking rules), to the argument list:

```
(defun PMC (search-space rules
  &key (heuristic-rules ()) (fwc-rules ()) (sols-mode :once) (rnd? nil))
...)
```

`fwc-rules` is given as an optional argument, i.e. by default `PMC` uses pure backtrack search.[4] This choice was made because we have the experience that pure backtrack search is quite often surprisingly efficient.[5] This optional argument allows the user to fine-tune the problem at hand by incrementally adding forward checking rules. Because forward checking does not affect the end

1. `V1` is connected with `V2`, `V3`, `V4`, `V5` and `V6`. `V2` is connected with `V3`, `V4`, `V5`, and `V6` , etc. This graph is already so densely connected that we make no attempt to draw it.

2. We will give examples of rules of this type in section 5.4.

3. Automatic translation is only possible for index variable rules and for wild card rules that do not use the special variables l and rl in the Lisp-part of the rule.

4. It is important to note that the fwc-rules argument can only consist of PWConstraints rules that can be automatically translated.

result, the user can add only those rules which are the most crucial ones for the optimisation of the search.

To conclude this section, let us demonstrate the effect of adding forward checking to pure backtrack search. We do this with the help of two musical examples. In the first one these we use only pure backtrack search. The task is to calculate all 3856 instances of the *all-interval series*.[1] An all-interval series is a twelve-tone row that does not contain interval duplicates. The intervals are calculated as the modulo 12 difference between two adjacent pitch-classes in the row.[2] The search is carried out using the following expression:[3]

```
(PMC '((0) (1 2 3 4 5 7 8 9 10 11) (1 2 3 4 5 7 8 9 10 11)
      (1 2 3 4 5 7 8 9 10 11) (1 2 3 4 5 7 8 9 10 11) (1 2 3 4 5 7 8 9 10 11)
      (1 2 3 4 5 7 8 9 10 11) (1 2 3 4 5 7 8 9 10 11) (1 2 3 4 5 7 8 9 10 11)
       (1 2 3 4 5 7 8 9 10 11) (1 2 3 4 5 7 8 9 10 11) (6))
      '((* ?1 (?if (not (member ?1 (rest rl)))) "no pitch-class duplicates")
          (* ?1 ?2 (?if (unique-int? (mod (- ?2 ?1) 12) (rest rl)
                                 :key #'(lambda (n) (mod n 12))))
             "no (modulo 12) interval duplicates"))
       :sols-mode :all)4
```

The second example is a modification of the first one: we change the search-space and add two index variable rules. The search-space is now defined so that the tenth and eleventh domains are `(1 7 11)` and `(5)`. The two index variable rules accept only certain set-classes at the given indexes (see the index-rules "`indexes 1, 5 and 10 -> 3-5b`" and "`indexes 2 and 10 -> 2-5`"). We add forward checking by giving the two index variable rules also as forward checking rules (see the argument for the keyword `:fwc-rules`):

5. Actually almost all PWConstraints examples discussed until now have been solved with pure backtrack search. The only exception is the "chains" example (section 5.1.5.2). There we noticed a considerable improvement when using forward checking.

1. An alternative way of producing all-interval rows is found in Morris and Starr (1974).

2. Morris and Starr (1974:364).

3. The search-space list of our example starts with (0), because we are not interested in the different transpositions of the resulting rows. The last item of the search-space list is fixed at (6), as it is known that the modulo 12 difference between the first and the last item of an all-interval series is always 6. Morris and Starr (1974:365).

4. The function *`unique-int?`* used in the second rule "no (modulo 12) interval duplicates" checks that there are no modulo 12 interval duplicates in the partial solution.

```
(PMC '((0) (1 2 3 4 7 8 9 10 11) (1 2 3 4 7 8 9 10 11) (1 2 3 4 7 8 9 10 11)
       (1 2 3 4 7 8 9 10 11) (1 2 3 4 7 8 9 10 11) (1 2 3 4 7 8 9 10 11)
       (1 2 3 4 7 8 9 10 11) (1 2 3 4 7 8 9 10 11) (1 7 11) (5) (6))
       '((* ?1 (?if (not (member ?1 (rest rl)))) "no pitch-class dups")
         (* ?1 ?2 (?if (unique-int? (mod (- ?2 ?1) 12) (rest rl)
             :key #'(lambda (n) (mod n 12)))) "no interval dups")
         (i1 i5 i10 (?if (member (sc-name (list i1 i5 i10)) '(3-5b)))
             "indexes 1,5 and 10 -> 3-5b")
         (i2 i10 (?if (member (sc-name (list i2 i10)) '(2-5)))
             "indexes 2 and 10 -> 2-5"))
       :fwc-rules
        '((i1 i5 i10 (?if (member (sc-name (list i1 i5 i10)) '(3-5b)))
             "indexes 1, 5 and 10 -> 3-5b")
         (i2 i10 (?if (member (sc-name (list i2 i10)) '(2-5)))
             "indexes 2 and 10 -> 2-5"))
     :sols-mode :all)  =>

((0 8 11 10 7 2 4 9 3 1 5 6) (0 2 11 10 1 8 4 9 3 7 5 6))
```

The addition of forward checking rules results in a search that is almost 10 times faster than a pure backtrack search. This improvement is understandable if we look at the graph representation of our problem (figure 5.37):

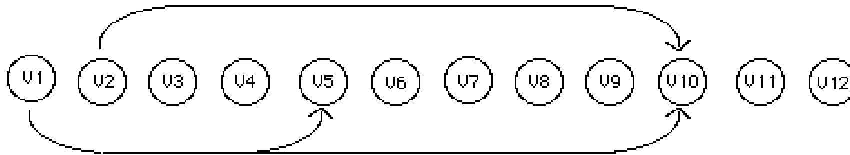


Fig.5.37: Arcs for the index variable rules.

As the domain of `V10` is quite sparse, containing only the values `(1 7 11)`, the forward checking algorithm can reduce the search-space heavily already at the stage when possible values are selected at `V2` and `V5`.

5.3 First Case Study: Counterpoint Rules

In the first PWConstraints case study we study the problem of translating counterpoint rules to our rule formalism. We start by extending the simple queue structure used by the search-engine to a full-scale score representation scheme. After this we show how the PWConstraints rule formalism can be used to write "typical" counterpoint rules found in counterpoint textbooks. Finally we examine how the score representation scheme affects the definition of the search-space and the interface between PWConstraints and PW.

Our main focus in this section is to develop tools that permit *polyphonic search problems*. Therefore it is not our purpose to simulate some specific historical style or to produce musical examples. There are several reasons for avoiding these topics.

Firstly, it is clear that the task of listing and formalising the most crucial rules constituting a given musical style is far from trivial. Typically, counterpoint textbooks give a large number of rules, but these are not always clearly formalised. A specific problem in translating textbook rules to a formal system like ours, is to control and detect conflict situations, i.e. situations where two or more rules conflict. We should also have a way to distinguish between important and unimportant rules, to handle exceptional cases, etc. Secondly, producing an extensive musical example in a given style would require a thorough discussion of the "compositional choices" to be made for the particular example in question. We would have to discuss aspects like large scale formal segmentation, choice of cadences, imitation techniques, etc. Thirdly, although this section does not include musical end results, later in section 5.4, we will use the tools developed here to produce a large-scale musical example. The rules to be discussed there rely largely on the ideas presented in this section.

5.3.1 PWConstraints Compared to CHORAL

Perhaps the most ambitious and closest predecessor to our presentation is the CHORAL project by Ebcioglu (1992:295-332). It is an expert system for harmonising chorales in the style of J.S. Bach. It is based on a logic programming language called *BSL* (*Backtracking Specification Language*). CHORAL has some features similar to PWConstraints. Like PWConstraints, it contains a backtrack search-engine. A set of *absolute* rules filter unwanted candidates from a partial solution. It includes also heuristic rules that affect the ordering of the accepted candidates.

Besides obvious similarities, there are some major differences between CHORAL and PWConstraints. For example, in CHORAL the rules are written in a remotely Lisp-like syntax resembling first-order predicate calculus. In PWConstraints, in turn, the rules are always defined by the pattern-matching part and the Lisp-code part. The latter part is written in standard Common Lisp. CHORAL supports the idea of having multiple viewpoints by providing different predefined *views* like *chord skeleton view*, *melodic string view*, *time-slice view*, etc. These views work in a rhythmically relatively simple style like chorales. In PWConstraints, however, the score representation scheme allows the search problem to handle scores that can be rhythmically arbitrarily complex. Finally, when CHORAL generates Bach chorales it uses about 350 rules. This is in contrast with PWConstraints examples. These contain typically around 5 to 10 rules (the largest examples include about 20 rules).[1]

5.3.2 Score Representation

Until now we have assumed that the basic data structure used by the search-engine is a vector of search-variables. This simple queue structure has many favourable features. It is ideal when designing a search-engine. The implementation of the basic operations, like moving forwards and backwards in a search-space, is straightforward. Also, this structure affects the way a search prob-

---

1. Understanding how an enormous set of 350 rules affects the end result seems to be an impossible task for an average user. One can of course claim that one needs many rules to get interesting results. On the other hand there is also a limit to how complex a system can grow before the user looses control over the system. Probably adding more and more rules does not automatically guarantee that one gets better results.

lem is formulated. The definition of a search-space is easy, as it clearly reflects the underlining structure used by the search-engine. The pattern-matching part of a rule, depending on the order in which search-variables are processed, helps us to formulate rules in an efficient and clear way.

The main problem with the current scheme however is that it does not allow polyphonic search problems. In a polyphonic search problem, like in a traditional polyphonic score, we should be able to operate with several layers (parts, voices) of events (notes). Each event, in turn, has its own onset and offset time.

In the following discussion we will combine the tools already defined earlier in this chapter (sections 5.1 and 5.2) with extensions that allow polyphonic search problems.

We start by assuming that the user gives the rhythmic structure of the search problem in advance. This is achieved by first preparing a score in a standard PW `RTM-editor`. The contents of the `RTM-editor`, or the *input score*, is then given as an argument to the Lisp function that performs the search. The search, in turn, aims at filling the input score with pitch information.[1]

5.3.2.1 Score-Sort

Before the search we have to translate the input score to a structure that can be used by the search-engine. The basic idea is to collapse any given polyphonic score to 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 the *score-sort* algorithm.

Score-sort works as follows. We read a score from left to right and sort notes in the order they appear in it. If two or several notes share the same attack time, they are sorted so that the longest notes are placed before the shorter ones.[2] If two or more notes have the same attack time *and* the same duration, we can order them freely. We will adopt the convention that notes having the highest part number are considered first.[3]

To summarise, score-sort is a hierarchical sort algorithm where we order first by attack time, then by duration and finally by part number.

Let us consider the following example to demonstrate how score-sort operates:

1. This means that the rhythmic structure is fixed during search. This may seem an artificial limitation. However, we have to consider the complexity of solving both aspects - the polyphonic rhythmic structure *and* the pitch structure - efficiently in one search. The rhythmic structure can nevertheless be controlled *indirectly* by defining special melodic rules that control "melodic unisons" (i.e. melodic successions where a note is followed by one or several notes with identical pitch). If the melodic unisons are interpreted so that the first note of this group is considered as an "attack" note while the remaining notes are considered "tied" notes, we are in fact controlling also the rhythmic structure of the result.

2. This ordering has important consequences as it means that long notes are always considered first when proceeding in the search. The reverse is true when we backtrack during search.

3. We assume that the first or highest part gets number 1, the second highest number 2, etc.

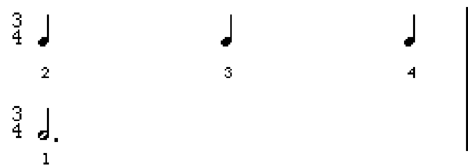


Fig.5.38: The rhythmic structure of a two-part score sorted to a flat list.

Figure 5.38 gives a two-part score, one measure in length.[1] The upper part consists of three notes while the lower one has one. The task is to order the four notes. The numbers below each note indicate the order, or *note index*, given by score-sort.

We start from the beginning of the score. We find first two notes with an identical attack time. The dotted half note of part 2 is sorted before the first quarter note of part 1. Thus the dotted half note gets number `1` and the quarter note number `2`. Note `3` is found by reading the next attack (second note of part 1). The final note, number `4`, is the third note of part 1.

Figure 5.38 shows also how the search proceeds in the score. We begin with note `1`. Then we go on to notes `2` and `3`, and finally finish the search at note `4`. Backtracking is done in reversed order: `4`, `3`, `2` and `1`.

Figure 5.39 below gives the beginning of the rhythmic structure of a four part chorale by J.S. Bach. The numbers below each note are given by score-sort.

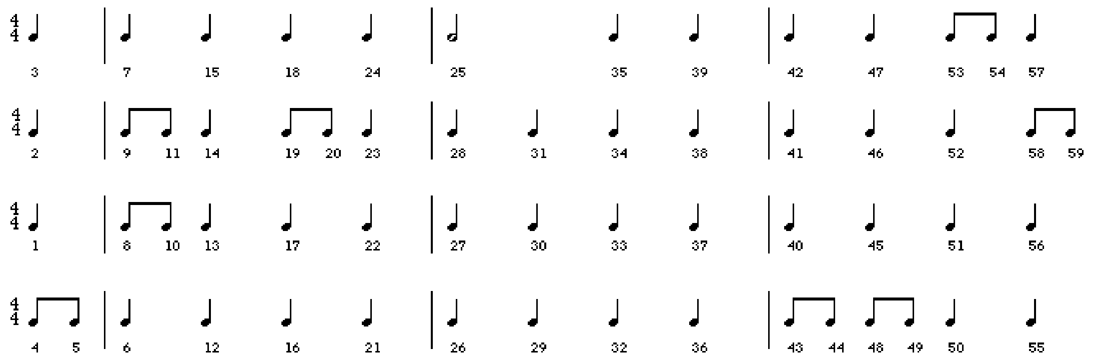


Fig.5.39: The rhythmic structure of the opening of a J.S. Bach chorale.

We find first four notes with the same attack time. This group consists of three quarter notes and one eighth note (part 4). We start with the quarter notes and order them by part number. The eighth note is the fourth item in our sort. Now we proceed to the next attack and find in part 4 a

1. We are concerned here only with the rhythmic structure. Consequently we do not show any pitches or staff lines.

single eighth note that gets number 5. We continue to the next attack and find a group of four notes: two quarter notes and two eighth notes. The quarter notes (parts 1 and 4) are first ordered by part number followed by the eighth notes (parts 2 and 3), also ordered by part number, etc.

Figure 5.40 below shows a more complex rhythmic structure. This example is taken from Anton Webern's Op. 16. No. 3 and is given without further comments:

Fig.5.40: Score-sort applied to the beginning of Anton Webern's Op. 16. No. 3.

5.3.2.2 Restoring Musical Information

When translating a score to a flat list we loose of course a lot of relevant musical information needed later by the rules. For instance, after score-sort, a note belonging to a given part does not know about its preceding note in the same part. Our next task is thus to restore the musical information found in the input score.

5.3.2.3 "Musical Map"

A practical solution to this problem is to store local knowledge in the search-variables. This knowledge, in turn, should enable us to collect dynamically the melodic, harmonic and voice leading information associated with each search-variable. For instance, typical melodic rules should be able to access easily previous notes of the melodic line (i.e. notes that have been "composed" already). Harmonic rules, in turn, should know which notes are already sounding when accepting or rejecting the current candidate. Harmonic progression rules should know about the current chord, the previous chord, etc.

What we need is a kind of "musical map" of the input score. Like any ordinary map the musical map consists of main "roads" or "paths" which are connected in turn to more local ones.

In order to move around this map fluently, we need a new data structure. This structure will be used later to define the main paths of a musical map. A standard Lisp list is not ideal for our purposes as it is awkward to move backwards in it. This is because a Lisp list has no backward links. Therefore we define a new class, called ***`linked-list-item`***:[1]

1. A similar idea of using linked-lists to represent musical scores can be found in Ames (1985:251).

```
(defclass linked-list-item ()
  ((prev-item :initform nil :accessor prev-item)
   (next-item :initform nil :accessor next-item)))
```

linked-list-item contains two slots prev-item and next-item. These slots provide the links to both directions. Figure 5.41 below gives a *linked-list chain* consisting of four linked-list-items L1, L2, L3 and L4.[1]

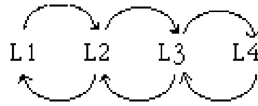


Fig.5.41: A chain of linked-list-items.

Next, we define some useful methods for linked-list-item:

```
(defmethod n-prev-items ((self linked-list-item) count) ...)
```

n-prev-items returns count previous items as a list. For instance if we send n-prev-items to L3 and ask for two items, we get the list (L2 L1) (figure 5.41).

```
(defmethod n-next-items ((self linked-list-item) count) ...)
```

n-next-items, in turn, returns count next items.

```
(defmethod all-prev-items ((self linked-list-item)) ...)
```

all-prev-items returns all previous items.

```
(defmethod all-next-items ((self linked-list-item)) ...)
```

all-next-items returns all next items.

Finally, the function *link-list* links a list of linked-list-items into a chain:

```
(defun link-list (linked-list-items) ...)
```

For instance, if we give the linked-list-items L1, L2, L3 and L4 as the argument to link-list, it links them together as shown in figure 5.41.

1. The chain in figure 5.41 is linked in the following way. L1's prev-item is nil and L1's next-item is L2 (see the arrow pointing from L1 to L2). L2's prev-item is L1 and L2's next-item is L3 (arrows pointing from L2 to L1 and to L3). L3's prev-item is L2 and L3's next-item is L4. Finally, L4's prev-item is L3 and L4's next-item is nil.

5.3.2.4 Melodic-Context

After this we discuss in more detail how the most important aspects of the input score are preserved. These aspects include *melodic-context*, *harmonic-context*, *harmonic-slice* and *metric-context.*

With melodic-context we mean that each note in a horizontal melodic succession is able to refer to its predecessor and successor. In order to preserve the melodic-context of a score, we redefine the `search-variable` class definition as follows:[1]

```
(defclass search-variable (key-item linked-list-item) ...)
```

Since we inherit from `linked-list-item,` we are able to link `search-variables` together. We collect all notes of a part in a list and link it by calling the function `link-list`. This process is repeated for each part in the score. The *horizontal melodic successions* will form the first main paths in our musical map.

5.3.2.5 Harmonic-Context

Preserving the harmonic surrounding of each note is more complex. We will split the discussion into two parts: harmonic-context and harmonic-slice. The former is useful in rules that control the step-by-step construction of harmonies during search. The latter is used typically in situations where we are interested in the *complete* harmonic status of the current note.

First we examine the concept of harmonic-context. It is defined as a list of those notes sounding at the attack time of the current note. It is important to observe that we exclude from this list all notes that still have no value, i.e. the search has not yet found solutions for them.[2]

Let us, for example, define the harmonic-context for each note in the Bach example given in figure 5.39 above. We start with note `1` having no harmonic-context (i.e. it is `nil`). Next we proceed to note `2`. This note has the harmonic-context `(1)`, because note `1` is sounding and it has already a value. We proceed to note `3`. The harmonic-context is now `(1 2)`. The harmonic-context of note `4` is in turn `(1 2 3)`. We continue to note `5` and the harmonic-context is still `(1 2 3)`. This is because notes `1`, `2` and `3` are still sounding at the attack time of note `5`, etc.

Figure 5.42 below shows the Bach example with harmonic-context. The note index is found under each note. The harmonic-context of each note is given as a list of note indices below the note index.[3]

1. The original `search-variable` class definition was given in section 5.2.1.1. The `key-item` class was defined in section 5.2.3.

2. The excluded notes have higher note index values than the current one (i.e. they will be processed by the search *after* the current note).

3. The harmonic-context indices in figure 5.42 are not in any specific order.

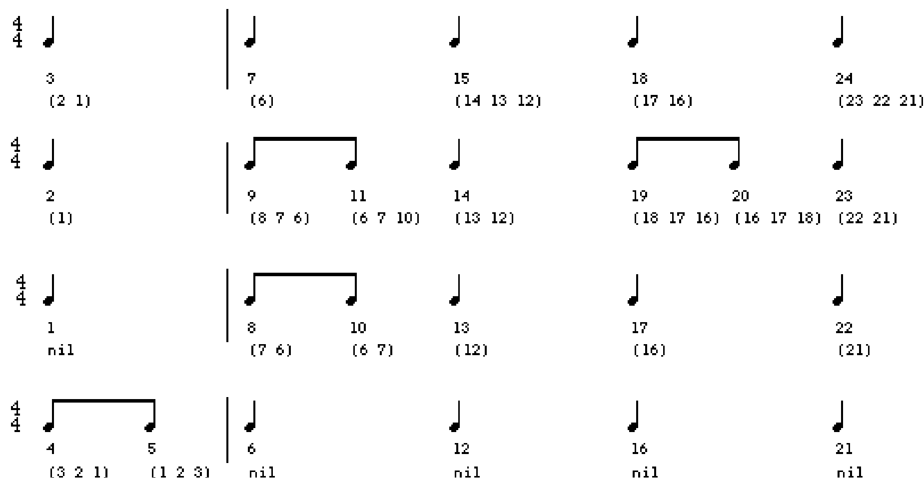


Fig.5.42: Beginning of the Bach example with harmonic-context added.

We conclude this section by giving in figure 5.43 the Webern example with note indices and harmonic-context:

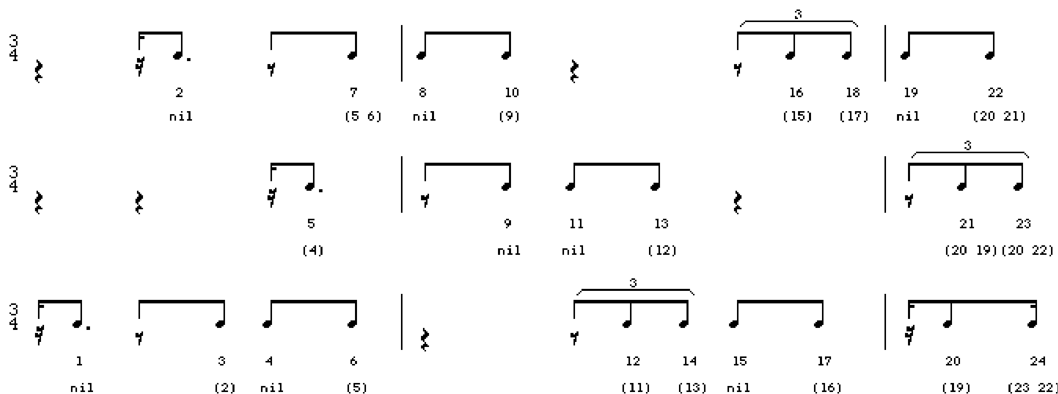


Fig.5.43: The Webern example with harmonic-context added.

5.3.2.6 Harmonic-Slice

A harmonic-slice is a list consisting of all notes sounding concurrently with the current note. In this case we are not interested whether they have values or not. The current note itself is included in this list.

If we consider again the Bach example given in figure 5.42, we notice that the four first notes have the same attack time. The harmonic-slice for each one is `(1 2 3 4)`. For note `5` it is `(1 2 3 5)`. Next we find again four notes with the same attack time. Each of them has the harmonic-slice `(6 7 8 9)`. At the next attack we find notes `10` and `11`. The harmonic-slice for them is `(6 7 10 11)`, etc. We see that all notes with the same attack time share the same harmonic-slice information.

Figure 5.44 below shows the Bach example with note indices, harmonic-context and harmonic-slices (given below the harmonic-context).[1]

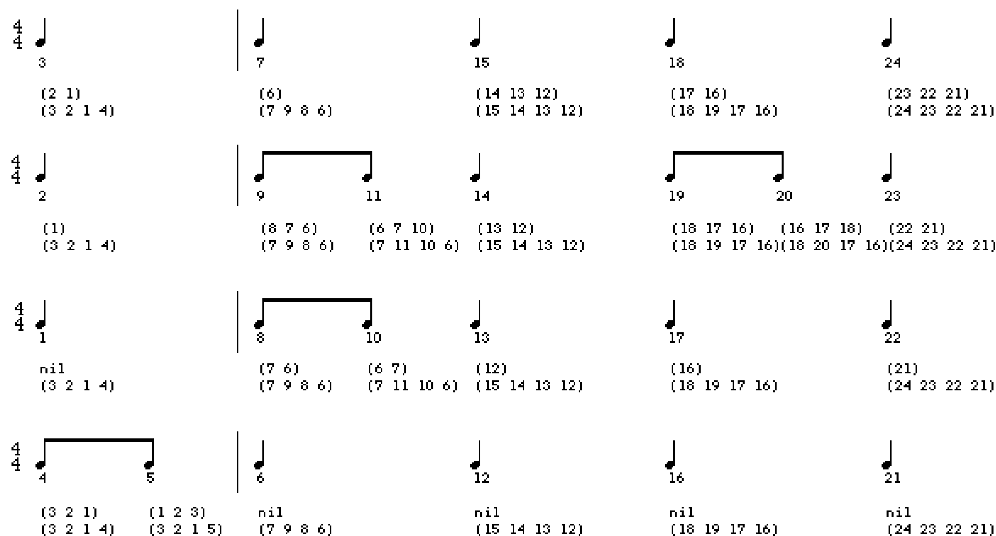


Fig.5.44: The Bach chorale with note indices, harmonic-context and harmonic-slices.

After this we define a new class, called ***`harmonic-slice:`***

```
(defclass harmonic-slice (key-item linked-list-item) ())
```

`harmonic-slice` inherits from `linked-list-item`. This means that we are able to link a list of `harmonic-slice` objects.

We need also a constructor, ***`make-harmonic-slice,`*** making instances of `harmonic-slice` objects:

```
(defun make-harmonic-slice (time s-variables) ...)
```

1. The harmonic-context and harmonic-slice indices in figure 5.44 are not in any specific order.

The time argument is the attack time, given in ticks, of the harmonic-slice object. s-variables, in turn, is a list of search-variables defining the harmonic-slice. For instance the first harmonic-slice object in figure 5.44 contains the search-variables (1 2 3 4). After collecting all harmonic-slice objects of a score we link this list. The *chain of harmonic-slices* constitute the second main path in our musical map.

5.3.2.7 Metric-Context

Many counterpoint rules assume that we are able to access the metric surrounding of a given note. For example, the contents of a melodic rule may depend on whether an eighth note is found on the downbeat or not. Some harmonic rules may allow specific dissonances, if the current note is on the upbeat, etc.

The metric-context of a given note is defined firstly by the *metric pattern* of the note and secondly by its *position* within this pattern.

Our starting point will be the beat-list.[1] If, for example, a beat consists of one quarter note, the beat-list of this beat is (1 (1)). If a beat consists of two eighth notes, the beat-list is (1 (1 1)), etc. Thus each beat and each note within the beat is associated with a beat-list, or a *rtm-pattern*. Also, all notes of the beat have a particular position, or *rtm-position*, within the rtm-pattern. In order to preserve the metric-context we must store in each note the information about its rtm-pattern and rtm-position.

Let us suppose that we want to define a rule with the following pattern-matching part: (* ?1 ?2 <Lisp-code>). Moreover, the Lisp test function of the rule should only be called in cases where the variables ?1 and ?2 belong to the rtm-pattern (1 (1 1)) (figure 5.45):[2]

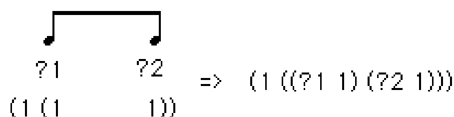


Fig.5.45: Matching ?1 and ?2 with (1 (1 1)).

The rule is defined as follows:

```
(* ?1 ?2                                         ; 1
   (if? (if (match-rtm? (1 ((?1 1) (?2 1))))    ; 2
            <Lisp-test>
            t)))
```

1. See for more details section 4.6.1.1.
2. We also assume that ?1 is the first note of the rtm-pattern and ?2 is the second.

We first give the two variables `?1` and `?2` in the pattern-matching part (1). Then we call in the Lisp-code part the macro ***match-rtm?*** with the argument `(1 ((?1 1) (?2 1)))` (2). This argument combines the variables `?1` and `?2` with the rtm-pattern `(1 (1 1))` resulting in `(1 ((?1 1) (?2 1)))` (see figure 5.45).

The `match-rtm?` macro is defined as follows:

```
(defmacro (match-rtm? (rtm-pat+vars) ...)
```

`match-rtm?` returns `true` if the following conditions are fulfilled. Firstly, all variables mentioned in the `rtm-pat+vars` argument must share the same rtm-pattern. Secondly, the rtm-positions of the variables must match the positions given by `rtm-pat+vars`.

Let us take another example where we are interested to match all eighth note triplets found in the score (figure 5.46):

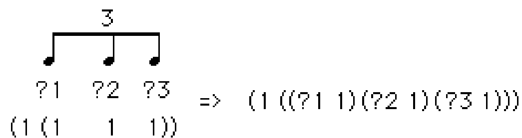


Fig.5.46: Matching `?1`, `?2` and `?3` with `(1 (1 1 1))`.

Thus we have three variables in the pattern-matching part and the rtm-pattern is `(1 (1 1 1))`. We combine this information to obtain the following rule:

```
(* ?1 ?2 ?3
  (if? (if (match-rtm? (1 ((?1 1)(?2 1)(?3 1))))
           <Lisp-test>
           t)))
```

As a final example, let us define a rule for the variables `?1` and `?2` and the rtm-pattern `(1 (-1 (1 (1 -1 1)))` (figure 5.47):

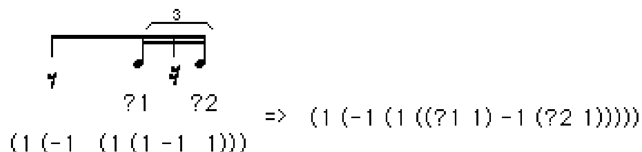


Fig.5.47: Matching `?1` and `?2` with `(1 (-1 (1 (1 -1 1)))`.

```
(* ?1 ?2
  (if? (if (match-rtm? (1 (-1 (1 ((?1 1) -1 (?2 1))))))
           <Lisp-test>
           t)))
```

5.3.2.8 Completing the Score Representation Scheme

The score representation still requires class definitions for `linked-beat`, `linked-measure` and `linked-part.` Each of these inherit from `linked-list-item` and are thus used to define new paths in the musical map.

We start by defining `linked-beat`:

```
(defclass linked-beat (key-item linked-list-item) ())
```

The *horizontal beat successions* of a score are constructed as follows. We first collect all `beat-objects` found in a given part. For each `beat-object` we make an instance of `linked-beat` and link these to a chain of beats. This process is repeated for each part in the score.

The `linked-measure` class definition, in turn, is as follows:

```
(defclass linked-measure (key-item linked-list-item) ())
```

The *horizontal measure successions* of a score are constructed in a similar way to the horizontal beat successions. We collect all `measure-objects` found in a given part and make an instance of `linked-measure` for each `measure-object`. We then link the `linked-measure` objects to a chain of measures. This process is repeated for each part in the score.

As the last class definition inheriting from `linked-list-item`, we define the `linked-part` class:

```
(defclass linked-part (key-item linked-list-item) ())
```

For each part found in the score we make an instance of `linked-part` and link these to a *chain of parts*.

Finally we define a class, called `part-collection`, that collects all parts of a score together:

```
(defclass part-collection (key-item) ())
```

The `part-collection` object is mainly a collection of parts, but it contains also other global information of the score. The current `part-collection` object is stored under the global variable `*part-collection*`.

Figure 5.48 below shows the Bach example where the double arrows indicate the linked-list chains of the score. These chains include all horizontal melodic successions, beat successions (labelled `B1`, `B2, ...`) and measure successions (`M1`, `M2, ...`). Below the parts we find a chain of harmonic-slices (indicated by the note index lists). Finally the parts themselves are also linked together (the double arrows at the left side of figure 5.48).

(3 2 1 4) (3 2 1 5) (7 9 8 6) (7 11 10 6) (15 14 13 12) (18 19 17 16)(18 20 17 16)(24 23 22 21)

Fig.5.48: The Bach example containing linked-lists of parts, measures, beats, notes and harmonic-slices.

The figure 5.48 is however not complete. We must for instance show pointers from notes to beats and from beats to measures.[1] Let us take a more detailed look at the second measure of part 2 of figure 5.48:

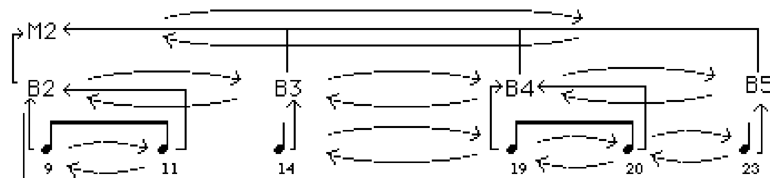


Fig.5.49: Linked-list chains and pointers.

1. Pointers differ from linked-list-items in that they can point only in one direction. Hence they are given as simple arrows.

As before, the double arrows in figure 5.49 indicate linked-list chains while the simple arrows are pointers. We see, for example, that notes `9` and `11` point to `B2`, and `B2`, in turn, points to `M2`. Note `14` points to `B3`, `B3` to `M2`. Notes `19` and `20` point to `B4`, `B4` to `M2`, etc.

Figure 5.50 below shows how the different components of a score interact together. We give a catalogue of keywords for `search-variable`, `linked-beat`, `linked-measure`, `linked-part`, `part-collection` and `harmonic-slice`:[1]

Search-variable:

:note-index

:mel-index

:time+dur

:rtm-pattern

:rtm-pos

:beat

:measure

:part

:part-num

:harmonic-context

:harmonic-slice

Linked-beat:

:`s-variables`

:measure

:b-num

:rtm-pattern

:time

Linked-measure:

:beats

:part

:m-num

:sign

:time

Linked-part:

:measures

:part-num

1. The keywords allow us to read information from an object using the method read-key. This is possible as all objects in figure 5.50 inherit also from `key-item`.

Part-collection:	**Harmonic-slice:**
:parts	:s-variables
:part-s-variables-ls-ls	:time
:sorted-s-variables	
:harmonic-slices	

Fig.5.50: The keywords of search-variable, linked-beat, linked-measure, linked-part , part-collection and harmonic-slice.

Figure 5.50 has two main purposes. Firstly, it shows what kind of information is stored into each respective object.[1] Secondly, the keywords are used in the next section to write utility functions and macros. These functions and macros, in turn, are used intensely by the counterpoint rule examples defined in section 5.3.3.

As can be observed from figure 5.50 the PWConstraints score representation scheme allows both a *top-down* and a *bottom-up* approach. In the former we start from the current part-collection and go into more detailed information in the hierarchical structure, i.e. parts, measures, beats and notes (or search-variables). The search-variables, in turn, allow us to refer to the score information in the opposite direction. We start from a single note and proceed upwards in the hierarchy to beats, measures, parts and end up with the part-collection constituting the whole score.

5.3.2.9 Utility Functions and Macros

In this section we list a number of general utility functions and macros. These will be our basic repertoire when writing counterpoint rules in section 5.3.3.[2]

1. For instance, a search-variable knows about its note index (see keyword :note-index), its position in the melodic succession within its part (:mel-index), its start-time and duration (:time+dur), rtm-pattern and rtm-position (:rtm-pattern and :rtm-pos), its beat (:beat), its measure (:measure), its part (:part), the part number of its part (:part-num), its harmonic-context (:harmonic-context) and its harmonic-slice (:harmonic-slice). A linked-beat, in turn, contains a list of its search-variables (:s-variables), its measure (:measure), its beat number (:b-num), its rtm-pattern (:rtm-pattern) and its start-time (:time). Part-collection has a list of parts (:parts), a list of lists of search-variables where each sublist is associated to a part (:part-s-variables-ls-ls), a list of search-variables sorted by score-sort and a list of harmonic-slices (:harmonic-slices), etc.

2. We will not give any further comments on the functions or macros, as they are relatively simple. Most of them simply access some information from a search-variable using the method read-key.

```
(defmacro m (s-variable)
 "returns the value (m for midi) of s-variable"
  `(value ,s-variable))

(defun l->ms (s-variables)
 "returns values of a list of s-variables"
  (mapcar #'(lambda (s-variable) (value s-variable)) s-variables))

(defmacro beat (s-variable)
 "returns the linked-beat of s-variable"
  `(read-key ,s-variable :beat))

(defmacro measure (s-variable)
 "returns the linked-measure of s-variable"
  `(read-key ,s-variable :measure))

(defmacro nindex (s-variable)
 "returns the note index (given by score-sort) of s-variable
 (starting from 0)"
  `(read-key ,s-variable :note-index))

(defmacro mindex (s-variable)
 "returns the mel-index (melodic index) of s-variable
 (starting from 1)"
  `(read-key ,s-variable :mel-index))

(defmacro beatnum (s-variable)
 "returns the beat number of the beat of s-variable
  (starting from 1)"
  `(read-key (read-key ,s-variable :beat) :b-num))

(defmacro measurenum (s-variable)
 "returns the measure number of the measure of s-variable
  (starting from 1)"
  `(read-key (read-key ,s-variable :measure) :m-num))

(defmacro partnum (s-variable)
 "returns the part number of the part of s-variable
  (starting from 1)"
  `(read-key ,s-variable :part-num))

(defmacro hc (s-variable)
 "returns the harmonic-context of s-variable as a list s-variables"
  `(read-key ,s-variable :harmonic-context))

(defmacro hslice (s-variable)
 "returns the harmonic-slice of s-variable as a list s-variables"
  `(read-key ,s-variable :harmonic-slice))

(defun startt (s-variable)
 "returns the start time (in ticks) of s-variable"
  (first (read-key s-variable :time+dur)))

(defun durt (s-variable)
 "returns the duration (in ticks) of s-variable"
  (second (read-key s-variable :time+dur)))

(defun endt (s-variable)
 "returns the end time (in ticks) of s-variable"
  (let ((l (read-key s-variable :time+dur)))
     (+ (first l) (second l))))

(defun attack-item? (hslice s-variable)
```

```
 "returns true if s-variable, being a member of hslice,
  is an attack s-variable"
  (= (time hslice) (startt s-item)))

(defun downbeat? (s-variable)
 "returns true if s-variable is downbeat note,
  i.e. its rtm-position is 0"
  (= (car (last (read-key s-variable :rtm-pos))) 0))

(defun prev-item-on-downbeat (s-variable)
"returns a downbeat s-variable of the previous beat or nil"
  (let ((prev-beat (prev-item (read-key s-variable :beat)))
         downbeat-var)
    (when prev-beat
      (setq downbeat-var (first (read-key prev-beat :s-variables)))
      (when (and downbeat-var (downbeat? downbeat-var))
        downbeat-var))))
```

5.3.2.10 Melodic Pattern-Matching

After this we must redefine the relationship between the pattern-matching part of a rule and the flat list of search-variables given by score-sort. So far the pattern-matching part has reflected clearly the order in which the search-variables were processed by the search-engine. After score-sort, however, the order the search-variables has no obvious musical meaning anymore. It is used mainly internally by the search-engine.

We will adopt the convention that the pattern-matching part of a rule in a polyphonic search problem corresponds to the horizontal melodic successions of a score.

For instance, let us take the following rule: `(* ?1 ?2 <Lisp-code>)`. The variables `?1` and `?2`, mentioned in the pattern-matching part, refer to all (adjacent) melodic two note successions.

5.3.2.11 Non-Attack Search-Variables

In score-sort, when sorting the flat list of search-variables, we consider only cases when a new note appears in a score. Thus this approach can be called *attack oriented*. This scheme, however, does not capture cases that fulfil two conditions. Firstly, the number of harmonic notes in a polyphonic texture decreases because of rests. Secondly, the score does not contain new attacks at this moment. These cases are important to observe as the harmonic situation of the score has changed. This is why we insert special search-variables, called *non-attack search-variables*, in the sorted list of search-variables. Figure 5.51 below gives two non-attack cases indicated with arrows. The harmonic-slices are given below the score.[1] The non-attack search-variables have harmonic-slices that are marked with a "+" sign.

1. Because this example contains rather complex rhythms, we do not draw the harmonic-slices as lists but as columns, in order to save space.

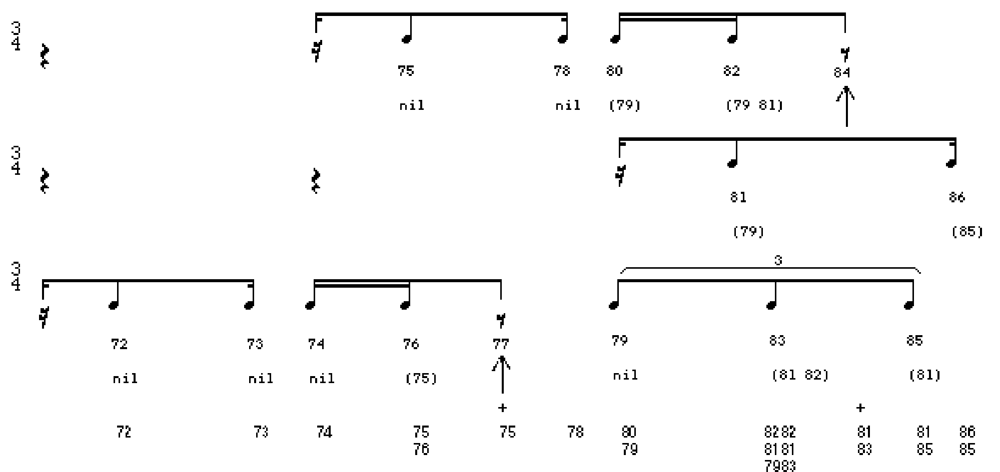


Fig.5.51: The Webern example, m. 7, with non-attack search-variables.

The non-attack search-variables are "passive" in a sense that they do not contain notes. Thereby they have no values that can be tested by the rules. This means that we have to create a new category of rules, called *non-attack rules*. Typically these rules refer only to the harmonic-slice of the current non-attack search-variable. For instance, let us assume a non-attack rule disallowing the set-classes `2-4` and `2-5` in all non-attack cases:

```
(* ?1 (?if (not (eq-SC? '(2-4 2-5) (l->ms (hslice ?1)))))
"2-4 or 2-5 disallowed in non-attack cases")
```
[1]

5.3.3 Translating Counterpoint Rules

After defining the score representation scheme, we proceed to the second main subject of this section. We will adopt a number of counterpoint rules found in textbooks and translate them to the PWConstraints rule formalism.[2] There are however several reasons why exact translations may be difficult to achieve. For instance, the texts often do not give the exact conditions under which a given rule should be applied. Instead, the conditions have to be "inferred" or "interpreted" from the given musical examples. Furthermore, textbook rules may include expressions like "rare cases", "common cases", "strange cases", "exceptional cases", etc. which are difficult to translate into our formal system.[3] Finally, cases where several rules given in different places in the text are in conflict with each other are especially problematic. Thus some of the PWConstraints translations

1. `eq-SC?` was defined in section 5.1.5.2 and `l->ms and hslice` in section 5.3.2.9.

2. Our main source will be de la Motte (1981). In particular we concern ourselves in the third chapter that deals with the counterpoint style of Josquin Desprez.

given below can only be considered as some kind of "interpretations" or "approximations" of the discussed textbook rules.

We will divide the rules in three main groups: melodic rules (labelled from M1 to M8), voice leading rules (V1 V2 V3) and harmonic rules (H1 H2 H3 H4).

5.3.3.1 Melodic Rules

Melodic rules will be quite straightforward as they use the pattern-matching capabilities of PW-Constraints. We start with a group of melodic rules that deal with allowed intervals, contrary motion and treatment of jumps in the same direction (M1 M2 M3 M4 M5). The first rule determines the allowed melodic intervals:

```
M1:
(* ?1 ?2 (?if (member (- (m ?2) (m ?1))
                  '(1 -1 2 -2 3 -3 4 -4 5 -5 7 -7 8 12 -12)))
     "allowed melodic intervals")
```
[1]

We calculate the interval between the last two melodic notes and check if it is found among the list of allowed intervals.[2]

```
M2:
(* ?1 ?2 ?3 (?if (not (eq-SC? '(3-11b 3-11a 3-10 3-12)
                              (m ?1) (m ?2) (m ?3))))
    "disallowed three note melodic successions")
```

We take the last three melodic notes and check that they do not form one of the following set-classes: `3-11b` (major), `3-11a` (minor), `3-10` (diminished) and `3-12` (augmented).

3. We discussed some solutions to these problems already in sections 5.1.5.4 and 5.2.4. In the former section we showed how to generate musical material with the help of statistical distribution of the musical properties in the result. In section 5.2.4 we discussed how heuristic rules can be used to favour (or not to favour) certain results.

1. It is important to note that the variables ?1 and ?2 are search-variable objects not values. This is why must access the values of ?1 and ?2 with the expressions (m ?1) and (m ?2). For more details see section 5.2.3. The macro m was defined in section 5.3.2.9.

2. A similar rule is found in de la Motte (1981:60). The difference is that we omit the "rare" intervals 9 (major 6th), 15 and 16 (minor and major 10th).

M3:
```
(* ?1 ?2 ?3
   (?if (let ((n1 (m ?1)) (n2 (m ?2)) (n3 (m ?3)))
      (not (and (up-down-movem? n1 n2 n3)                      ; 1
                (member (abs (- n3 n1)) '(6 10 11 13 14)))))) ; 2
        "avoid reaching 6, 10, 11, 13 or 14")
```
[1]

We first test if `?1, ?2` and `?3` form a melodic movement that is ascending or descending (1). If this is the case the absolute interval between `?1` and `?3` should not be a tritone, 7th or 9th (2).

M4:
```
(* ?1 ?2 ?3
   (?if (let ((n1 (m ?1)) (n2 (m ?2)) (n3 (m ?3)))
     (not (and (or (> (abs (- n2 n1)) 7)        ; 1
                   (> (abs (- n3 n2)) 7))
              (up-down-movem? n1 n2 n3)))))     ; 2
        "contrary motion before/after jumps greater than 7")
```

We check if the absolute interval between `?1` and `?2` or `?2` and `?3` is greater than `7` (perfect 5th) (1). If this true then `?1`, `?2` and `?3` should not form an ascending or descending melodic movement (2). In other words a note before or after a large jump should move in contrary motion.[2]

M5:
```
(* ?1 ?2 ?3
    (?if (ballistic? (m ?1) (m ?2) (m ?3)))
       "ballistic rule")
```

We take the last three melodic notes and check that `?1, ?2` and `?3` form a "ballistic" melodic movement. A ballistic movement allows two jumps in the same direction, but the larger jump has to be below the smaller one.[3]

The next melodic rules (M6 M7 M8) form a group as they use the metric-context capabilities defined in section 5.3.2.7.[4]

1. The function *up-down-movem?* returns true if the arguments form an ascending or descending movement.

2. de la Motte (1981:69). A more strict version of the rule can be defined by simply changing the interval 7 to a smaller value.

3. de la Motte (1981:70). The function *ballistic*? returns true if the arguments form a ballistic movement.

4. All so called "Schwarze Noten", or "black notes", are given as eighth notes, not as quarter notes as in de la Motte (1981:94).

```
M6:
(* ?1 ?2
   (?if (if (match-rtm? (1 ((?1 1) (?2 1))))  ; 1
          (<= -4 (- (m ?2) (m ?1)) 2)         ; 2
          t))
        "Within a group of two 1/8 notes, never upward
        jumps on downbeat. Upbeat downward jumps less
        than or equal to 4.")
```
[1]

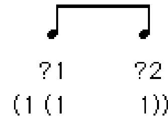


Fig.5.52: Matching ?1 and ?2 with (1 (1 1)).

We take the last two melodic notes and check that both ?1 and ?2 belong to the rtm-pattern (1 (1 1)) (figure 5.52). If this is the case then ?1 is a *downbeat note*, ?2 is a *upbeat note* and both ?1 and ?2 are eighth notes (1).

If match-rtm? matches, we check that the melodic interval is in the range -4 and 2 (2). In other words ?1 can jump only downwards and this jump should never exceed 4 (major 3rd).

```
M7:
(* ?1 ?2
   (?if (if (match-rtm? (1 (1 (?1 1))))
          (<= -4 (- (m ?2) (m ?1)) 4)
          t))
        "never greater 1/8 note upbeat jumps than 4")
```

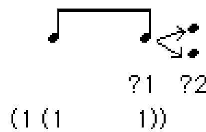


Fig.5.53: Matching ?1 with (1 (1 1)).

We take the last two melodic notes and check if ?1 is the *second* note in the rtm-pattern (1 (1 1)) (figure 5.53). If match-rtm? matches, we check that the melodic interval between ?1 and ?2 is in the range -4 and 4. In other words ?1 can jump a maximum of a major 3rd up or down.

1. The macro match-rtm? was defined in section 5.3.2.7.

```
M8:
(* ?1 ?2 ?3
   (?if (if (or (match-rtm? (2 (3 (?2 1))))      ; 1
                (match-rtm? (1 (1.0 (?2 1)))))
          (if (and (or                           ; 2
                     (= (- (m ?2) (m ?1)) -1)
                     (= (- (m ?2) (m ?1)) -2))
                    (< (durt ?2) (durt ?3)))     ; 3
           (> (- (m ?3) (m ?2)) 0)          ; 4
             (<= (abs (- (m ?3) (m ?2))) 4))     ; 5
          t))
      "single upbeat 1/8 note melodic rule after a long note")1
```

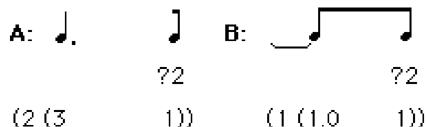


Fig.5.54: Matching ?2 with a single upbeat eighth note.

We take the last three melodic notes and examine if ?2 is the second note in the rtm-pattern (2 (3 1)) or (1 (1.0 1)) (figure 5.54) (1). If this is the case we check that the melodic movement between ?1 and ?2 is downward and stepwise (2). Also we check that the next note, ?3, is longer than an eighth note (3). If these checks succeed ?2 can jump upwards (4). If they fail, however, the jump between ?2 and ?3 can be a maximum of 4 (major 3rd) (5).[2]

5.3.3.2 Voice Leading Rules

We proceed to the next group of rules dealing with voice leading. Compared to melodic rules, voice leading rules are difficult to write. This is of course not surprising because we have to consider simultaneously melodic, rhythmic and harmonic aspects of a score.[3]

All voice leading rules to be discussed in this section are defined within a four note context. These notes can be thought of forming a kind of two by two "matrix" (see figure 5.55 below). The horizontal relations of this matrix define two melodic successions, where the pair ?11->?12 belongs to one part and ?21->?22 to another. The vertical relations of the matrix in turn give two harmonic intervals: int1 and int2. In all cases we assume that ?12 and ?22, forming the second harmonic interval int2, are "synchronised", i.e. they will share an identical attack time. However, the timing of ?11 and ?21 is "free", they are allowed to be synchronised or not.

1. The function durt was defined in section 5.3.2.9.

2. This rule is found in de la Motte (1981:106). The rule is probably too strict as it does not permit cambiatas. We can correct this by allowing ?2 to jump downwards (see the comment line 4 in the rule M8). Cambiatas will be discussed in section 5.3.3.3.1.

3. This means that from now on we will make considerable use of the harmonic-context, the harmonic-slice and the metric-context of search-variables. See for details sections 5.3.2.5, 5.3.2.6 and 5.3.2.7.

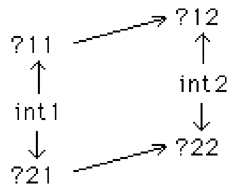


Fig.5.55: Does int1 and int2 form a parallel movement?

We start the discussion by defining a general help function, called ***check-parallels?***:

```
(defun check-parallels? (?11 ?12 ?21 ?22 pc-int-pairs)
  (let ((int1 (mod (abs (- (m ?11) (m ?21))) 12))    ; 1
        (int2 (mod (abs (- (m ?12) (m ?22))) 12)))
    (find-if #'(lambda (int-pair)                      ; 2
                 (and (= int1 (first int-pair))
                      (= int2 (second int-pair))))
             pc-int-pairs)))
```

The first elements of the argument list of check-parallels? are four notes, ?11 and ?12 belonging to one part and ?21 and ?22 to another (figure 5.55). pc-int-pairs, in turn, is a list of harmonic *modulo 12 interval* pairs, i.e. all intervals in pc-int-pairs are given within one octave. pc-int-pairs defines the parallel movements we are interested in. If, for example, a pair is (0 0), it means that check-parallels? checks for parallel unisons and octaves.[1] If it is (7 7), we check for parallel 5ths. If it is (6 7), we check for harmonic successions where the first interval is a tritone and the next one a perfect 5th, etc.

check-parallels? first calculates the harmonic modulo 12 intervals int1 and int2 (1).[2] Next it tries to find an identical interval succession among pc-int-pairs (2). If such a succession is found we know that there is a parallel movement and check-parallels? returns true.[3]

After this we define the first voice leading rule disallowing *open parallels*:[4]

```
V1:
(* ?1 ?2 (?if (not (open-parallels? ?1 ?2))) "no open parallels")
```

1. In other words we check if modulo 12 of both intervals, int1 and int2, is 0 (figure 5.55).

2. Before calculating modulo 12 of the intervals, we take the *absolute* value of them. This is because we do not know if ?11 and ?12 are above or below ?21 and ?22.

3. If voice-crossings are allowed the check-parallels? function should be extended in the following way. Before taking the absolute intervals we check whether the melodic lines are crossing each other. If this is the case then check-parallels? should always return nil.

4. By open parallels we mean parallel 5ths and octaves.

We first take two adjacent melodic notes `?1` and `?2`. Then we call a help function, called *open-parallels?*, that returns `true` in case of open parallels. We call `open-parallels?` inside `not` to disallow such situations.

As our rule gives only two adjacent melodic notes, the first thing we have to do is to find the missing two notes in order to complete the matrix (figure 5.56). This is done by checking whether the harmonic-context of the second note, `?2`, contains a note that is synchronised with it. If such a note is found - let us call it `?x` - we ask for the melodic note which precedes it, called `mel-pair`:

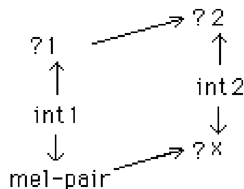


Fig.5.56: Open parallels?

The function `open-parallels?` is defined as follows:

```
(defun open-parallels? (?1 ?2)
  (let (mel-pair)
    (find-if
      #'(lambda (?x)
          (and (attack-item? (hslice ?2) ?x)           ; 1
               (setq mel-pair (prev-item ?x))          ; 2
               (check-parallels? ?1 ?2 mel-pair ?x     ; 3
                                 '((0 0) (7 7) (6 7))))) ; 4
        (hc ?2))))
```

`open-parallels?` checks if there are open parallels among the harmonic-context, `hc`, of `?2`.[1] As we loop through `hc` we test first that the current `hc` note, `?x`, has its attack time simultaneously with `?2` (1).[2] Also we read the previous melodic note of `?x`, called `mel-pair` (2). Then we call the `check-parallels?` help function (3). `?1` and `?2` form the first melodic succession and `mel-pair` and `?x` the second one (figure 5.56). We disallow parallel octaves, parallel 5ths and tritones proceeding to perfect 5ths by giving `((0 0) (7 7) (6 7))` as `int-pairs` (4).

Next we discuss the second voice leading rule which is a variant of the previous one. We disallow parallels that occur on downbeats (figure 5.57). In this case the matrix should contain only downbeat notes. Furthermore, we must guarantee that also the first vertical notes, `?1` and `mel-pair`, are synchronised:

1. The macro `hc` was defined in section 5.3.2.9.
2. The function `attack-item?` was defined in section 5.3.2.9.

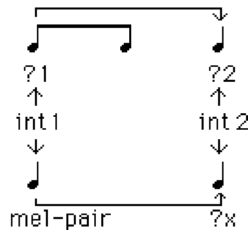


Fig.5.57: Downbeat parallels?

The rule is defined as follows:

```
V2:
(* ?2 (?if (let ((?1 (prev-item-on-downbeat ?2)))  ; 1
               (if (and ?1 (downbeat? ?2))          ; 2
                  (not (downbeat-parallels? ?1 ?2))
                  t)))
        "no downbeat parallels")
```

The pattern-matching part contains only one note `?2`. We ask first for an earlier melodic note of `?2.` This note, `?1`, should be both situated in the previous beat and be on the downbeat (1).[1] We then check whether `?2` is also a downbeat note (2). If this is the case and `?1` is not `nil`, i.e. there is a downbeat note in the previous beat, we call a help function, called ***downbeat-parallels?.***

```
(defun downbeat-parallels? (?1 ?2)
  (let (mel-pair)
    (find-if
     #'(lambda (?x)
         (and (attack-item? (hslice ?2) ?x)
              (setq mel-pair (prev-item-on-downbeat ?x))  ; 1
              (= (startt ?1) (startt mel-pair))           ; 2
              (check-parallels? ?1 ?2  mel-pair ?x
             '((0 0) (7 7) (6 7)))))
     (hc ?2))))
```

`downbeat-parallels?` is almost identical with the function `open-parallels?`. The first difference is at (1), where we ask for the previous *downbeat* note of `?x`, not simply the previous one (see also figure 5.57). Also, we must check that `?1` and `?mel-pair` are synchronised (2).

The final voice leading rule disallows hidden parallels. In this rule we are not interested in the first harmonic interval of the matrix (figure 5.58). We want only to check whether the second harmonic interval, `int2`, is arrived at via parallel motion:

1. The function `prev-item-on-downbeat` was defined in section 5.3.2.9.

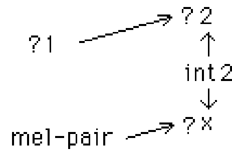


Fig.5.58: Hidden parallels?

The hidden parallels rule is defined as follows:

```
V3:
(* ?1 ?2 (?if (not (hidden-parallels? ?1 ?2))) "no hidden parallels")
```

As in the rule V1 we take two adjacent melodic notes `?1` and `?2`. Then we call the help function *`hidden-parallels?`*.

```
(defun hidden-parallels? (?1 ?2 &optional (5ths-fl t))
  (let ((mel-int1 (- (m ?2) (m ?1)))                          ; 1
        mel-pair mel-int2 int2)
    (find-if
     #'(lambda (?x)
         (and (attack-item? (hslice ?2) ?x)                   ; 2
              (setq int2 (mod (abs (- (m ?x) (m ?2))) 12))    ; 3
              (setq mel-pair (prev-item ?x))                  ; 4
              (setq mel-int2 (- (m ?x) (m mel-pair)))         ; 5
              (or (and (plusp mel-int1) (plusp mel-int2))     ; 6
                  (and (minusp mel-int1) (minusp mel-int2)))
              (if 5ths-fl                                     ; 7
                (or (= int2 0) (= int2 7))                    ; 8
                (= int2 0))))                                 ; 9
     (hc ?2))))
```

`hidden-parallels?` has arguments for `?1` and `?2` forming a two note melodic succession. The optional argument, `5ths-fl`, is a flag used to determine whether we want to detect hidden parallel 5ths or not.

`hidden-parallels?` first calculates the melodic interval, `mel-int1`, of `?1` and `?2` (1). Then we examine if we find among the harmonic-context of `?2, hc,` a note that produces hidden parallels with `?1` and `?2`. First we check if `?x` has its attack time simultaneously with `?2` (2). Then we calculate the harmonic modulo 12 interval, `int2`, of `?2` and `?x` (3) and ask for the previous melodic note of `?x,` called `mel-pair` (4). Next, we calculate the melodic interval, `mel-int2`, of `mel-pair` and `?x` (5). We check whether both melodic motions, `mel-int1` and `mel-int2`, are similar, i.e. both are either positive or negative (6). If this is the case we check the status of `5ths-fl` (7). If it is `true`, we ask if `int2` is equal `0` or `7` (8). Otherwise we are interested only in hidden octaves and check if `int2` is equal to `0` (9).

5.3.3.3 Harmonic Rules

We now move on to deal with harmonic rules. In order to define these efficiently we will work with partial solutions while constructing harmonies: each time the search proceeds to a new note we immediately run all harmonic rules. The harmonic-context of the current note enables us to decide on potential candidates even when we do not know yet the complete harmonic solution at a given moment.

The following discussion is divided into three parts. First we define a help rule that disallows voice crossings, along with a general help function that is able to detect harmonic dissonances. Then we write a harmonic rule for dissonances occurring on the upbeat. Finally we implement two harmonic rules which deal with suspensions.

We simplify our problem somewhat by disallowing voice crossings. In this case we know always whether we are referring to a *bass note*, i.e. the lowest note of the current harmony. This scheme allows us to reduce possible candidates immediately. The rule that disallows voice crossings is defined as follows:

```
H1:
(* ?1 (?if (no-voice-crossings? ?1)) "no voice crossings")
```

The rule calls a help function called ***no-voice-crossings?:***

```
(defun no-voice-crossings? (?1)
  (let ((hc (hc ?1))
        sorted-notes)
    (if hc
     (progn
       (setq sorted-notes                              ; 1
         (sort (copy-list (cons ?1 hc)) #'>
               :key #'(lambda (n) (partnum n))))
       (apply #'<= (mapcar #'(lambda (n) (m n))        ; 2
                          sorted-notes)))
     t)))
```

`no-voice-crossings?` returns `true` if there are no voice crossings in the harmonic-context, or `hc`, of `?1`. If `?1` has a harmonic-context we append `?1` to `hc`. The resulting list is sorted so that notes with the highest part number come first, i.e. we order them from bass to soprano (1). Then we check that all MIDI-values in this sorted list are in ascending order (3).

Let us first define which harmonic intervals are disallowed. A harmonic interval is defined as either the distance between a bass voice and a higher voice, or the distance between two higher voices. In the former case we test whether a harmonic interval is dissonant by calling the function *`dissonant-bass-int?:`*

```
(defun dissonant-bass-int? (int)
  (member (mod int 12) '(1 2 5 6 10 11)))
```

`dissonant-bass-int?` returns `true` if modulo 12 of `int` is a minor or major 2nd, a perfect 4th, a tritone, or a minor or major 7th.

In the latter case, i.e. when the harmonic interval is given by two upper voices, we call the function *dissonant-upper-int?:*

```
(defun dissonant-upper-int? (int)
  (member (mod int 12) '(1 2 10 11)))
```

`dissonant-upper-int?`, in turn, returns `true` if modulo 12 of `int` is a minor or major 2nd, or a minor or major 7th.

Next we need a function to recognise whether a note is a bass note:

```
(defun bass-note? (?1)
"returns t if ?1 belongs to the lowest part
in the harmonic-slice of ?1"
  (= (partnum ?1)
     (apply #'max (mapcar #'(lambda (v) (partnum v))
                          (read-key (hslice ?1) :s-variables)))))
```

bass-note? returns true if `?1` is a bass note. We simply check if `?1`'s part number is equal to the lowest part number in the harmonic-slice of `?1`.

Finally, we are ready to define a general function for testing harmonic dissonances:

```
(defun harmonic-dissonance? (?1)
  "Assumes that the lowest part always has the lowest pitch!
   Returns diss-note or nil."
  (let ((ref-midi (m ?1))                          ; 1
        (ref=bass-note (bass-note? ?1))            ; 2
        item=bass-note int)
   (find-if
     #'(lambda (?x)
         (setq item=bass-note (bass-item? ?x))     ; 3
         (setq int (abs (- (m ?x) ref-midi)))      ; 4
         (if (or ref=bass-note item=bass-note)     ; 5
           (dissonant-bass-int? int)
           (dissonant-upper-int? int)))
     (hc ?1))))
```

harmonic-dissonance? checks if `?1` forms a harmonic dissonance within the harmonic-context of `?1`. We first collect the MIDI-value of `?1` (1). We then check if `?1` is a bass note (2). After this we go through harmonic-context of `?1`, `hc`, and check if the current note of `hc`, `?x`, is a bass note (3). We calculate the harmonic interval, `int`, of `?x` and `?1` (4). Finally, if either `?x` or `?1` is a bass note, we call the function `dissonant-bass-int?`. Otherwise we call `dissonant-upper-int?` (5).

5.3.3.3.1 Passing note, Neighbor Note and Cambiata

We will discuss in this section the following non-harmonic notes: passing notes, neighbor notes and cambiatas. We will call them collectively *unstressed dissonances* or *upbeat dissonances* as in our discussion they all occur on the unstressed part of a beat.

First we have to be able to match *upbeat rhythms*, i.e. rhythms where an upbeat note can potentially become an upbeat dissonance. Figure 5.59 below defines the possible upbeat rhythms and rtm-patterns:

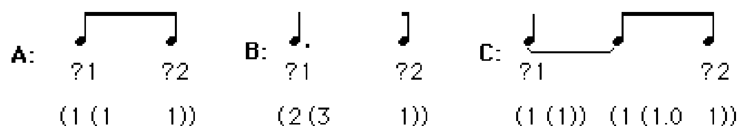


Fig.5.59: Possible upbeat rhythms and rtm-patterns.

The matching is done by the function ***match-upbeat-rtm?:***

```
(defun match-upbeat-rtm? (?1 ?2)
  (or (match-rtm? (1 ((?1 1) (?2 1))))   ; 1
      (match-rtm? (2 ((?1 3) (?2 1))))   ; 2
      (and (match-rtm? (1 ((?1 1))))     ; 3
           (match-rtm? (1 (1.0 (?2 1)))))))
```

`match-upbeat-rtm?` returns true if `?2` is an upbeat note, i.e. `?2` is on the unstressed part of the beat. We start by checking the case A of figure 5.59 (1). If this case fails we check case B (2) and finally case C (3).

Next we need some help functions to be able to recognise cambiatas, neighbor notes and passing notes. The first one, `cambiata-ints?,` checks whether the four notes in the argument list form a proper cambiata succession of intervals:

```
(defun cambiata-ints? (?1 ?2 ?3 ?4)
  (let ((int1 (- (m ?2) (m ?1)))
        (int2 (- (m ?3) (m ?2)))
        (int3 (- (m ?4) (m ?3))))
    (and (or (= int1 -1) (= int1 -2))   ; 1
         (or (= int2 -3) (= int2 -4))   ; 2
         (or (= int3 1) (= int3 2)))))  ; 3
```

The first interval must be a downward 2nd (1), the second a downward 3rd (2) and finally the third interval an upward 2nd (3).

The second cambiata help function, *`cambiata-rtm?`*, is used to match cambiata rtm-patterns.

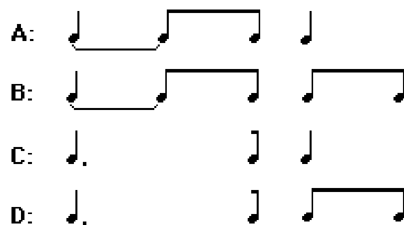


Fig.5.60: Possible cambiata rtm-patterns.

```
(defun cambiata-rtm? (?1 ?2 ?3 ?4)
  (and (or (and (match-rtm? (1 ((?1 1))))
                (match-rtm? (1 (1.0 (?2 1)))))   ; 1
           (match-rtm? (2 ((?1 3) (?2 1)))))
       (or (match-rtm? (1 ((?3 1))))              ; 2
           (match-rtm? (1 ((?3 1) (?4 1)))))))
```

cambiata-rtm? checks whether the four notes in the argument list form a proper cambiata rtm-pattern (see the alternatives A, B, C and D in figure 5.60).[1] The function consists of two `or` clauses. In the first one we check the rtm-patterns for `?1` and `?2` and in the second one those for `?3` and `?4`.

We need still two help functions: one for passing notes, called *scale-movem?*, and one for neighbor notes, called *side-movem?*.

```
(defun scale-movem? (n1 n2 n3) ...)
```

`scale-movem?` returns `true` if `n1`, `n2` and `n3` are in ascending or descending order and the adjacent note intervals between them do not exceed a major 2nd.

```
(defun side-movem? (n1 n2 n3 &optional (only-down t)) ...)
```

`side-movem?`, in turn, returns `true` if `n1` is equal to `n3` and `n2` is maximally a major 2nd higher or lower than `n1`. The optional argument, `only-down`, is a flag indicating if only downward movements are allowed.

Now we are able to write a rule that handles all upbeat dissonance cases: passing notes, neighbor notes and cambiatas:

```
H2:
(* ?1 ?2 ?3 ?4
   (?if
     (if (match-upbeat-rtm? ?1 ?2)                         ; 1
         (let ((n1 (m ?1)) (n2 (m ?2)) (n3 (m ?3)))
            (if (harmonic-dissonance? ?2)                  ; 2
               (or (and (cambiata-rtm? ?1 ?2 ?3 ?4)        ; 3
                        (cambiata-ints? ?1 ?2 ?3 ?4))
                   (scale-movem? n1 n2 n3)                 ; 4
                   (side-movem? n1 n2 n3))                 ; 5
               t))
          t)) "passing notes, neighbor notes and cambiatas")
```

We collect the last four melodic notes. Then we call `match-upbeat-rtm?` to check if `?2` is an upbeat note (1). If this is the case, we check if `?2` forms a harmonic dissonance (2). If the test fails, i.e. `?2` is a consonant note, the rule returns `true`, as we can always accept consonant upbeat notes.

1. Note that the first and the third rtm-pattern (A and C) contain only three notes. This means that the fourth note can be of any rhythmic value.

However, if the harmonic dissonance test succeeds, it means that ?2 is a dissonant note. In this case we first check whether we have a proper cambiata case (3). If this test fails we check if ?2 is a potential passing note (4). If we fail again we finally check if ?2 is a potential neighbor note (5).[1] If all cases fail then the whole rule fails. This is because ?2 does not form a proper upbeat dissonance.

5.3.3.3.2 Suspension

In our discussion we will consider only two harmonic cases for a downbeat note: either it is consonant or, if it is dissonant, it has to form a suspension with its harmonic-context.

Of the rules discussed in this chapter the suspension rule is the most difficult. This is because we must recognise, among other things, special rhythmic conditions before allowing suspensions. For instance, when discussing passing notes, neighbor notes and cambiatas, we were able to define the allowed rtm-patterns locally inside one voice. In case of suspensions we have to deal with situations where the rtm-patterns are distributed between two separate voices.

Figure 5.61 gives two fast and figure 5.62 three slow versions of possible suspension rtm-patterns. In all cases ?1 prepares the suspension, ?D introduces the downbeat dissonance and ?2 resolves the dissonance by moving stepwise downwards:

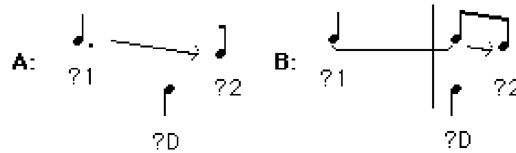


Fig.5.61: Suspension rtm-patterns, fast versions.

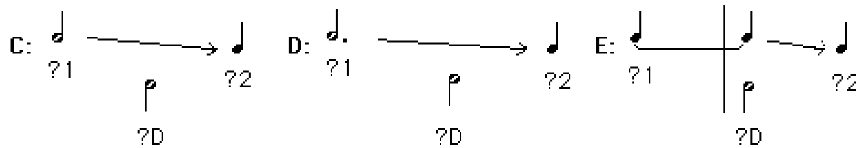


Fig.5.62: Suspension rtm-patterns, slow versions.

Let us start by defining a function, *suspension-rtm-match?*, that matches the possible suspension rtm-patterns:

```
(defun suspension-rtm-match? (?1 ?2 ?D)
  (and
   (downbeat? ?D)                                        ; 1
   (< (startt ?1) (startt ?D) (startt ?2))               ; 2
```

1. We do not give here the optional argument to side-movem? which means that we allow only downward neighbor notes.

```
(or
 (and
  (or  (match-rtm? (2 ((?1 3) (?2 1))))                   ; 3
       (and (match-rtm? (1 ((?1 1))))                      ; 4
            (match-rtm? (1 (1.0 (?2 1))))))
  (match-rtm? (1 ((?D 1)))))                               ; 5
 (and
  (and (or (match-rtm2? ?1 (1 ((?1 1))) (1 (1.0)))   ; 6
           (match-rtm? (2 ((?1 1))))
           (match-rtm? (3 ((?1 1)))))
       (match-rtm? (1 ((?2 1))))                           ; 7
       (match-rtm? (2 ((?D 1)))))))))1                     ; 8
```

suspension-rtm-match? receives ?1, ?2 and ?D as arguments. It returns true if they form one of the rtm-patterns found in figures 5.61 and 5.62. We check first that ?D, i.e. the dissonant note, is on the downbeat (1) and that the start-times of ?1, ?D and ?2 are in ascending order (2). Then we proceed to the fast versions A (3) and B (4) (see figure 5.61). If either A or B produces a match, we test whether or not ?D is a quarter note (5). In the slow versions, in turn, we first go through all possible rtm-patterns for ?1 (6), then for ?2 (7) and finally for ?D (8).

We also need a simplified version of the function harmonic-dissonance called ***harmonic-dissonance2?***. It has two arguments ?1 and ?2, and it returns true if they form a harmonic dissonance:

```
(defun harmonic-dissonance2? (?1 ?2)
  (let ((int (abs (- (m ?1) (m ?2)))))
    (if (or (bass-item? ?1) (bass-item? ?2))
      (dissonant-bass-int? int)
      (dissonant-upper-int? int))))
```

We come now to the second part of our problem. It is not obvious at what stage, in the name of efficiency, the suspension rule should be run. The problem lies again in the fact that the musical context is spread between two voices. One solution is to split the problem in two parts. In the first part we test if a note forms a potential dissonant note in a suspension (see at ?D below in figure 5.63):

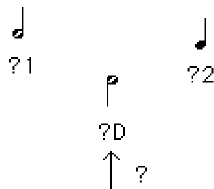


Fig.5.63: Is ?D a potential suspension dissonance?

1. We use in this expression a variant of the match-rtm? macro called match-rtm2?. The difference is that match-rtm2? can accept two adjacent rtm-patterns as argument not just one.

The first part of the suspension rule is defined as follows:

```
H3:
(* ?D (?if (suspension-part1? ?D))
 "Suspension rule, part 1, should be used with suspension-part2?")
```

The rule calls the help function, called *suspension-part1?:*

```
(defun suspension-part1? (?D)
  (if (downbeat? ?D)                                                    ; 1
    (let ((?prev (prev-item ?D)) ?2 harmonic-int)
      (not
       (find-if-not
        #'(lambda (?1)
            (if (harmonic-dissonance2? ?D ?1)                           ; 2
              (and (setq ?2 (next-item ?1))                             ; 3
                   (suspension-rtm-match? ?1 ?2 ?D)                     ; 4
                   (if ?prev (member (abs (- (m ?prev) (m ?1)))
                                     '(1 2)) t)                         ; 5
                   (setq harmonic-int (- (m ?1) (m ?D)))                ; 6
                   (if (plusp harmonic-int)
                 (member (mod harmonic-int 12) '(5 10 11))      ; 7
                     (member (mod (abs harmonic-int) 12) '(1 2))))   ; 8
              t))
        (hc ?D))))
    t))
```

`suspension-part1?` has one argument, `?D`, that is the hypothetical suspension dissonant note. We first check if `?D` is on the downbeat (1). If this is not the case then `suspension-part1?` returns `true` as we are not interested in this rule in upbeat notes.

If, however, `?D` is on the downbeat, then we continue with our assumption. Next we ask if we find among `?D`'s harmonic-context a note that both forms a dissonance with `?D` and is *not* (we use `find-if-not`) a proper candidate for the first note in a suspension (see at `?1` in figure 5.63). If such a note is found then we know that `?D` can never be a proper candidate because we allow on the downbeat only suspensions or consonances. In this case `suspension-part1?` returns `nil` (we use `not` around `find-if-not`). The purpose of `suspension-part1?` is to guarantee that all candidates at `?D` are either consonances or potential suspensions.

We will next take a closer look at the expression inside `find-if-not`. We first check if `?1` (the current note of the harmonic-context of `?D`) and `?D` form a harmonic dissonance (2). If `?D` is a consonance we return simply `true` meaning that `find-if-not` will continue.

If, however, `?D` is a dissonance, we must check that both `?1` and `?D` are valid candidates for a suspension. We read the *next* melodic note of `?1` and call it `?2` (3). We now test if `?1`, `?2` and `?D` have valid suspension rtm-patterns (4). If this is true we check that the melodic movement to `?D` is stepwise (5).[1] Then we calculate the dissonant harmonic interval, called `harmonic-int` (6). Finally, we check that `harmonic-int` is of an allowed type. If `?D` is below `?1`, then the following intervals are allowed: perfect 4th and minor and major 7th (7). If `?D` is above `?1`, then we permit only minor and major 2nds (8).[2]

---

1. The note before the downbeat dissonant note, i.e. the note preceding `?D`, should move stepwise to the dissonant note (de la Motte 1981:83). A less strict version of the rule could disregard this check.

2. de la Motte (1981:83).

We have not solved the whole problem yet. When deciding for the candidates for `?D` we do not yet know the pitch of `?2`.[1] After running the first suspension rule we know that all candidates at `?D` are either consonances or potential suspensions. We have to complete the suspension rule by checking that `?2` resolves the suspension in a correct way (see below figure 5.64).

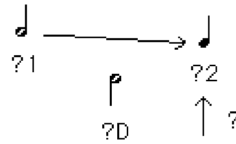


Fig.5.64: Is `?2` resolving a suspension?

Next we define the second part of the suspension rule:

```
H4:
(* ?1 ?2 (?if (suspension-part2? ?1 ?2))
 "Suspension rule, part 2, should be used with suspension-part1?")
```

The second suspension rule calls in turn the help function *suspension-part2?*.

```
(defun suspension-part2? (?1 ?2)
  "?1              ?2
   ?prev   ?D          "
  (if (downbeat? ?1)                                 ; 1
    (not
     (find-if-not
      #'(lambda (?D)
          (if (and (harmonic-dissonance2? ?1 ?D)      ; 2
                   (suspension-rtm-match? ?1 ?2 ?D)) ; 3
            (member (- (m ?2) (m ?1)) '(-1 -2))      ; 4
            t))
      (hc ?2)))
    t))
```

`suspension-part2?` receives as arguments the resolution part of a potential suspension: `?1` and `?2` (figure 5.64). We first check if `?1` is on the downbeat (1). If this is not the case, we return `true` because this cannot be a suspension case.

If, however, `?1` is on the downbeat then we continue with the rule. We look among `?2`'s harmonic-context, `hc`, for a candidate, `?D`, that is dissonant with `?1` (2). Also we check if `?1`, `?2` and `?D` have valid suspension rtm-patterns (3). If these tests return `true` we test if the melodic movement of `?1` and `?2` is downwards and stepwise (4). If the latter test returns `nil` we know that `?2` cannot be accepted as it does not resolve the suspension correctly. This means that the rule fails (again we use `find-if-not` inside `not`). In all other cases `suspension-part2?` returns `true`.

1. `?2` is the note that should resolve the suspension (figure 5.63).

5.3.4 Defining Search-Space for a Polyphonic Search Problem

5.3.4.1 Search-Space Score

So far we have defined the search-space by giving a list of candidates for each search-variable. This scheme is not anymore appropriate for a polyphonic search problem. This is because there is no more a direct relationship between the given search-space list and the order in which the search-variables are being processed during search. We need new tools to specify of the domains of each note in an input score.

A simple way to enter the search-space for a polyphonic search problem is to define a *static range* for each part. This is done by giving a list of lists where each sublist determines the range of a part in MIDI-values. Let us, for example, assume a score having two parts and a search-space that is defined by the following range lists: `((60 74) (48 69))`. The first sublist, `(60 74)`, indicates that the domains of all notes in the first part are in the range `60` to `74`. The second sublist, `(48 69)`, defines the domains of the second part in the same way.

If, however, we want to define *dynamic* `ranges` then this is done by preparing a score in a polyphonic PW `RTM-editor`. This score, called *search-space score,* must have the same number of parts and measures as the input score. Each part, in turn, contains a two-note chord sequence defining the dynamic `ranges` for this particular part.

For instance, let us assume that the user has prepared the following input score (figure 5.65):

Fig.5.65: The user defines the rhythmic structure for a search problem.

Let us next make a search-space score with two parts, each part having five measures. The `ranges` are given by the following two-note chord sequences (figure 5.66):

Fig.5.66: The user defines the search-space for the input score given in figure 5.65.

The domain `ranges` of each note in figure 5.65 is determined in the following way. We first connect in the search-space score all upper notes of part 1. This gives a dynamic high limit for the first part (see the dotted lines between notes in figure 5.66). Then we connect all lower notes of the same chord sequence and get the respective low limit. We repeat the same process for the second part.

Finally, a given note finds its range by reading its parts limit values at its attack time. Figure 5.67 below gives the first parts of the input score and the search-space score of our example. The `ranges` of each note are shown in the search-space score either by two-note chords or two-headed arrows:

Fig.5.67: Determining the dynamic `ranges` for an input score.

5.3.4.2 Constraining Notes

In section 5.1.2.3 we mentioned an attractive PWConstraints feature allowing us to manually "compose" parts of a solution in advance. This was done by constraining notes so that their domains include only one value. In a polyphonic search problem this possibility is of great interest as we can mark the constrained notes directly in the input score.

Figure 5.68 below gives an example where the user has "fixed" a cadence within a rhythmic structure. The constrained notes, i.e. notes that must be found in the result, are marked by a small "" sign. The "free" notes, i.e. notes that will be "composed" by the system, are not marked. In our example the first notes of parts 1 and 2 are free notes and their pitch has no special meaning. However, C5, B4 and C5 of part 1 and D4 and C4 of part 2 are all fixed notes and will remain unaltered in the result.

Fig.5.68: A cadence fixed by the user.

5.3.5 PW Interface

The interface between a polyphonic search problem and PW is defined by the function `score-PMC`:

```
(defun score-PMC (m-lines ranges rules
                  fwc-rules heuristic-rules non-attack-rules
                  prepare-fns+args allowed-pcs
                  sols-mode rnd? print-fl) ...)
```

`score-PMC` has following arguments. `m-lines` is a list of `measure-lines`, defining the input score. `ranges` is either a simple range list, a list of lists of `ranges` or a list of `measure-lines`. In the latter case the argument is given as a search-space score.[1]

The next three arguments define the different rule types given by the user. The first one, `rules`, is a list of ordinary PWConstraints rules. The second, `heuristic-rules`, is a list of heuristic rules.[2] The third, `non-attack-rules`, is a list of non-attack rules.[3]

`prepare-fns+args` is a list of user-definable preparation functions that are called before the search starts. They allow the user to customise the search problem. `allowed-pcs`, in turn, is a list of allowed pitch-classes. It is used to filter unwanted pitch-classes from the domains of the search-variables. For instance, by giving the list `(0 2 4 5 7 9 11)` we permit only "white" notes of the piano keyboard. In other words we exclude the pitch-classes `1, 3, 6, 8` and `10.`

Finally, the last three arguments, `sols-mode`, `rnd?` and `print-fl` are identical with the arguments for the `PMC` function described in section 5.1.4, so we will not comment them here.

`score-PMC` reads its inputs, makes an instance of a search-engine, creates a `part-collection` object and starts the search. When the search is finished `score-PMC` updates the input score by writing the result directly in the polyphonic `RTM-editor`.

Figure 5.69 shows an example of a polyphonic search problem in PW. It contains a `score-PMC` box (see arrow `score-PMC`). The first input for it is a input score containing three parts (arrow `score`). The rhythmic structure of the input score has been prepared by the user in the "`score`" window.[4] The `ranges` input is given by the list `((60 79) (54 75) (52 69))` (arrow `ranges`). The rules are defined in the text-editor window "`Rules.Lisp`" (arrow `rules`). This example uses neither heuristics rules nor non-attack rules, thus the inputs for them are `()`. We give here no preparation functions. The allowed pitch-classes are given by the list `(0 2 4 5 7 9 11)` (arrow `allowed-pcs)`.

---

1. The search-space score was described in section 5.3.4.1.
2. Heuristic rules were described in section 5.2.4.
3. Non-attack rules were described in section 5.3.2.11.
4. One should note that the user has constrained some notes in the cadence manually. These notes are marked in the "`score`" window with a "" sign. See for more details section 5.3.4.2. The pitches of all other notes are still undefined and will be determined by the search.

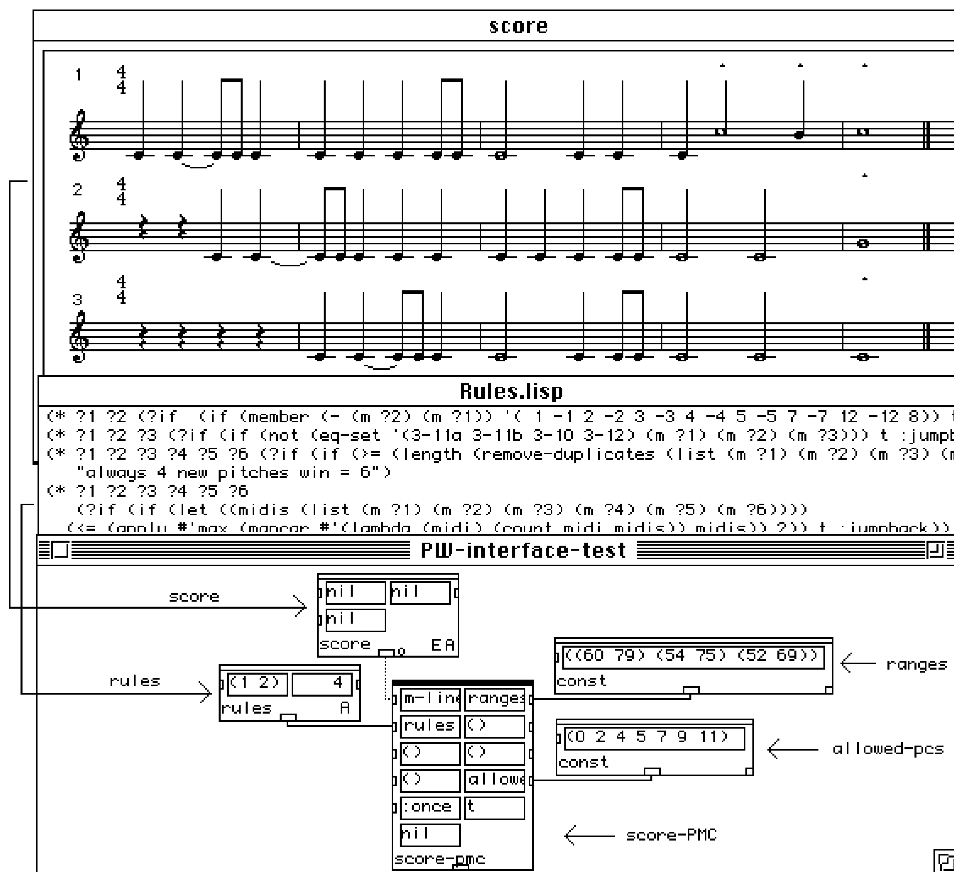


Fig.5.69: A polyphonic search problem in PW.

6 Bibliography

Castrén, M., *Joukkoteorian peruskysymyksiä,* Helsinki: Sibelius Academy, 1989.

Castrén, M., *RECREL: A Similarity Measure for Set-Classes,* Helsinki: Sibelius Academy, 1994.

Clocksin, W. F. and Mellish, C. S., *Programming in Prolog,* Springer-Verlag, 1984.

Ebcioglu, K., *An Expert System for Harmonization of Chorales in the Style of J.S. Bach,* Technical Report no. 89-09, Department of Computer Science, S.U.N.Y. at Buffalo, 1986.

Ebcioglu, K., 'An Expert System for Harmonising Chorales in the Style of J.S. Bach.' In *Understanding Music with AI: Perspectives on Music Cognition*. Editors: Balaban, M., Ebcioglu, K. and Laske, O., MIT Press, 1992, 295-332.

Forte, A., *The Structure of Atonal Music,* New Haven and London: Yale University Press, 1973.

Freuder, E. C. and Wallace, R. J., 'Partial Constraint Satisfaction.' *Artificial Intelligence,* 58, 1992, 21-70.

Goldberg, D. E., *Genetic Algorithms in Search, Optimization & Machine Learning,* Addison-Wesley Publishing Company, 1989.

Laurson, M., *PATCHWORK: A Visual Programming Language and some Musical Applications*, Helsinki: Sibelius Academy, 1996.

Leler, Wm, *Constraint Programming Languages,* Addison-Wesley Publishing Company, 1988.

Luger, G. F. and Stubblefield, W. A., *Artificial Intelligence,* The Benjamin/Cummings Publishing Company, Inc., 1993.

Mackworth, A. K., 'Consistency in Networks of Relations.', *Artificial Intelligence,* 8, 1977, 99-118.

Morris, R. D. and Starr, D., 'The Structure of All-interval Rows.', *Journal of Music Theory*, Vol. 18, 1974, 364-389.

Morris, R. D., 'Combinatoriality without the Aggregate.' *Perspectives of New Music*, Vol. 22, 1+2, 1982-83, 432-486.

Morris, R. D., *Composition with Pitch-Classes,* New Haven: Yale University Press, 1987.

de la Motte, D., *Kontrapunkt*. Kassel: Bärenreiter-Verlag, 1981.

Norvig, P., *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp,* Morgan Kaufmann Publishers, Inc., 1992.

Rueda, C. and Bonnet, A., *PW_Functionals Library, Reference manua,*. Code: Rueda, C.

Paris: IRCAM, 1993a.

Rueda, C. and Bonnet, A., *Situation - Librairie de modules pour la gestion de contraintes musicales, Reference manual,* Code: Rueda, C. Paris: IRCAM, 1993b.

Sabin, D. and Freuder, E. C., 'Contradicting Conventional Wisdom in Constraint Satisfaction' Proceedings of *ECAI*, 1994.

Siskind, P. and McAllester, D. A., 'SCREAMER: A Portable Efficient Implementation of Nondeterministic Common Lisp', Proceedings, *AAAI*, 1993.

Straus, J. N., *Introduction to Post-Tonal Theory*, New Jersey: Prentice Hall, 1990.

Index

A
Anonymous-Variable 41
At Least Property 60
Automatic Rules 56

B
Baboni Schilingi J. 2
Backtrack search-engine 32
Bonnet A. 33

C
Castrén M. 52
CHORAL 33, 87
CLOS 65
Constraint satisfaction problem 32
Counterpoint 86
counterpoint 1, 2, 16, 27, 30, 31, 33, 100
CSP 32

D
DOC 16
Duthen J. 2

E
Ebcioglu K. 32
Eliza 40

F
Fineberg J. 2
Forward checking algorithm 32

G
groups 29

H
Harmonic Rules 112
Harmonic Sequence Generator 31
harmonic-dissonance2? 117
Harmonic-Slice 93
harmonic-slice 100
Heininen P. 31, 48
HSG 31

L
Laurson M. 2
linked-beat 100
linked-measure 100
linked-part 100

M
Malt M. 2
MA-PMC1 4, 14, 15
MA-PMC2 4, 14, 15
Melodic Rules 104
Metric-Context 95
misc 16, 25
MIT 40
Morris R.D. 50
Musical Map 90

P
part-collection 100
Pattern Matching Constraints 44
Pattern-Matching 40
Pattern-matching interface 32
PC-set-theory 12, 13, 16, 19
Permutations 46
PMC 3, 9, 16, 44, 52
PWCs 9

R
RECREL 52
Rueda C. 2, 32, 33

S
score-PMC 2, 3, 4, 9, 10, 11, 15, 16, 22, 25, 27, 29, 122
Score-Sort 88
SCREAMER 33, 65, 76
Search-engine 65
Search-space 30
Search-variable 65
search-variable 100
Situation 32, 33
Splitter 61
Statistical Distribution 56
Suspension 116

V
Variable 41
Voice Leading Rules 107

W
Webern A. 57, 93
Weizenbaum J. 40