UCLouvain

École polytechnique de Louvain

Melodizer : A Constraint Programming Tool For Computer-aided Musical Composition

Author: **Damien Sprockeels**
Supervisor: **Peter Van Roy**
Readers: **Jean Bresson, Pierre Schaus**
Academic year 2021–2022
Master [120] in Computer Science and Engineering

Abstract

This master's thesis presents the design of a tool aiming to assist composers in their creation process. It uses constraint programming to enforce musical rules through two constraint satisfaction problems (CSP). This thesis is built on an already existing thesis focused on rhythm [11], and is as a result more pitch oriented.

Gecode, a powerful constraint solver, is used to model the musical problems in terms of constraints. OpenMusic, a visual programming and composition environment based on Lisp is used to represent the musical aspect of this thesis, and as a support for Melodizer. GiL, an interface between Gecode and Lisp, is used to connect Gecode and OpenMusic and allow for the creation of constraint problems in a musical environment.

A tool, Melodizer, is designed to help composers increase their expression and composition abilities by using constraints to generate melodies. It provides an intuitive and interactive interface making it easy for composers to customize their problem by giving input and setting parameters such as solution variety or optional constraints, and allowing him or her to edit any proposed solution.

Two CSPs allow Melodizer to model a variety of problems using a set of mandatory and optional constraints. It does not aim to generate ready-to-use music but rather to provide interesting out-of-the-box ideas for composers. It provides a solid base towards a bigger picture allowing composers to create symphonies from a similar tool.

If you are a composer, we suggest you look at chapter 7 first. It contains a variety of concrete examples of what Melodizer can be used for. Each of these examples comes with a video allowing you to hear the result, along with a small explanation on how to reproduce the same result using Melodizer in OpenMusic.

We want to thank
Peter Van Roy,
Karim Haddad from IRCAM,
Jean Bresson and Pierre Schaus,
Vanessa Maons and the INGI System Team,
Pascal Maes and Christophe Perzyna,
Baptiste Lapière
for the help they have provided us with this master's thesis.

Contents

1 Introduction **1**
- 1.1 Context and motivation . . . 1
- 1.2 Contributions . . . 2
- 1.3 Melodizer Example . . . 3
- 1.4 Roadmap . . . 3

2 Theoretical background **5**
- 2.1 Musical notions . . . 5
  - 2.1.1 Staff . . . 5
  - 2.1.2 Tonality . . . 7
  - 2.1.3 Chords . . . 7
- 2.2 Constraint programming . . . 8
  - 2.2.1 What is constraint programming? . . . 8
  - 2.2.2 Example of a CSP . . . 9
- 2.3 Gecode . . . 9
- 2.4 OpenMusic . . . 11
  - 2.4.1 Boxes . . . 11
  - 2.4.2 Representation . . . 13

3 Model **14**
- 3.1 Variables and their domains . . . 14
- 3.2 Musical constraints . . . 14
  - 3.2.1 Relation between pitch and intervals . . . 15
  - 3.2.2 In tonality constraint . . . 15
  - 3.2.3 Note on chord constraint . . . 16
  - 3.2.4 Interval between adjacent notes constraint . . . 19
  - 3.2.5 Harmonic interval chord constraint . . . 19
  - 3.2.6 Range restriction constraint . . . 20
  - 3.2.7 Melody direction constraints . . . 20
  - 3.2.8 All different notes constraint . . . 23
  - 3.2.9 At least n constraint . . . 23
- 3.3 Branching . . . 23
  - 3.3.1 Random number generators . . . 25
  - 3.3.2 Variable selection heuristic . . . 25
  - 3.3.3 Value selection heuristic . . . 25

4 Searching for a solution **27**
- 4.1 How to search? . . . 27
  - 4.1.1 Searching algorithm . . . 27
  - 4.1.2 Improving the variety amongst the solutions . . . 28
  - 4.1.3 Improving the speed of the search . . . 28
- 4.2 Imposing variety between solutions . . . 28
  - 4.2.1 How to compare two solutions . . . 28
  - 4.2.2 The constraint . . . 28
  - 4.2.3 Implementation . . . 28

4.2.4 Limitation . . . 29
4.3 Separating the search from the rest of the tool . . . 29
4.3.1 Why we should separate the search from the rest . . . 29
4.3.2 How it works in practice . . . 29
4.4 Example of a full musical CSP . . . 30

5 Design and implementation of Melodizer **33**
5.1 Representation of the musical elements . . . 33
5.2 Reminder on Voice objects . . . 34
5.3 Melodizer object design . . . 35
5.3.1 Input attributes . . . 35
5.3.2 Inner workings attributes . . . 35
5.3.3 Output attributes . . . 35
5.4 Melodizer object editor . . . 36
5.4.1 Input panel . . . 36
5.4.2 Additional constraints panel . . . 37
5.4.3 Search panel . . . 37
5.4.4 Solution assembly panel . . . 38
5.5 Melodizer object implementation . . . 39
5.5.1 Melodizer-utils . . . 39
5.5.2 Melodizer-csts . . . 39
5.5.3 Melodizer-csp . . . 40
5.5.4 Melodizer . . . 40

6 How to use Melodizer **41**
6.1 Setting up the environment . . . 41
6.1.1 Create a patch . . . 41
6.1.2 Load Melodizer and GiL . . . 41
6.1.3 Creating a melodizer object . . . 42
6.2 Motif, phrase, period . . . 42
6.3 Giving input to the object . . . 43
6.3.1 Giving input from the editor window . . . 43
6.3.2 From the patch . . . 44
6.4 Specifying the optional constraints . . . 44
6.5 Searching for a solution . . . 44
6.6 Assembling a melody using generated solutions . . . 45
6.7 Making Melodizer interact with other elements in OpenMusic . . . 45

7 Composer use scenarios for Melodizer **47**
7.1 Reminder . . . 47
7.2 Finding a melody to harmonize with a sequence of chords . . . 48
7.3 Revisiting an existing piece . . . 48
7.4 Finding a melody to play on top of a sequence of chords using Rhythm-Box to generate the rhythm of the melody . . . 48
7.5 Generating motifs, phrases and periods using optional constraints . . . 50
7.6 Generating variations of a melody while keeping the overall feel of the melody . . . 50
7.7 Using Melodizer to generate different musical phrases, creating variations of them and combining them together to create a musical piece . . . 53

8 GiL **55**
8.1 Structure . . . 55
8.1.1 C Wrapper . . . 55
8.1.2 Lisp Wrapper . . . 56
8.2 Contribution . . . 56
8.3 Example of use . . . 57
8.4 How to add to GiL . . . 57

9 Conclusion **59**
9.1 GiL . . . 59
9.2 Two musical CSPs . . . 59
9.3 Melodizer, an OpenMusic tool to assist composers in their creation process . . . 60
9.4 Future work . . . 60
9.4.1 Developing GiL . . . 60
9.4.2 Adding other CSPs to model a variety of problems . . . 60
9.4.3 Improving Melodizer . . . 61

Bibliography **62**

A How to install Melodizer **63**
A.1 Download . . . 63
A.2 Load the libraries in OpenMusic . . . 63

B GiL source code **65**
B.1 C wrapper . . . 65
B.1.1 space_wrapper.hpp . . . 65
B.1.2 space_wrapper.cpp . . . 73
B.1.3 gecode_wrapper.hpp . . . 84
B.1.4 gecode_wrapper.cpp . . . 89
B.2 Lisp Wrapper . . . 96
B.2.1 gecode-wrapper.lisp . . . 97
B.2.2 gecode-wrapper-ui.lisp . . . 108

C Melodizer source code **114**
C.1 melodizer-csts.lisp . . . 114
C.2 melodizer-csp.lisp . . . 119
C.3 melodizer-utils.lisp . . . 123
C.4 melodizer.lisp . . . 127

Chapter 1

Introduction

1.1 Context and motivation

Computers have an ever increasing role in our society, and music composition is no exception. They are found in many parts of the music production process, from writing simple phrases to composing complex pieces. This master's thesis will focus on computer-aided composition, which is just one of the many applications for computers in the musical world, with the help of constraint programming. The goal is to create a tool in OpenMusic[1], a composer workbench developed at IRCAM[2], that uses constraints programming with Gecode[3], a powerful framework for constraint based applications, to generate interesting and innovative melodies to increase a composer's exploration and expression abilities. It is built on the fact that musical rules, such as harmony and rhythm, can be expressed as mathematical expressions on numbers, and constraint programming is therefore a convenient solution to express those rules. The main advantage is that a solver is not influenced by personal biases or social context and can therefore suggest melodies that are very different from what a composer would come up with.

Past work on computer-aided composition with constraints is abundant ([1], [2], [16], [17]), so the goal of this work is not so much to explore musical rules expressed with constraints, but rather to provide composers with a practical musical composition tool. The focus has been to offer a wide set of useful tools and to create an interface allowing composers to easily handle their musical ideas and to generate new ideas. Compared to these works, we take advantage of progress in computer speed (100x) and sophistication in constraint solving (Gecode). We believe that the current situation is better suited for a successful tool than the previous works of twenty years ago.

This thesis has two main axes : developing an interface between Gecode and Lisp to allow the use of constraint programming in OpenMusic, and developing a constraint-based tool in OpenMusic that composers can actually use as a creative tool, though the latter is more important. It is built on the work of Baptiste Lapière [11], who made the first step towards such a tool by implementing the start of an interface between Gecode and Lisp, and a tool, Rhythm-Box, allowing for rhythm generation from the user's input. The aim of this work is to continue developing the interface between the constraints world and the musical world, while producing a powerful and useful tool for composers. Since work has already been done regarding rhythm generation, this thesis will be more pitch-oriented. Rhythm-Box[4] can be used to generate rhythm to use with Melodizer, or composers can invent rhythmic phrases themselves. The end goal is to be able to use Baptiste's work on rhythm along with the results of this thesis to provide an interesting tool for composers to experiment with.

In order to express constraints on musical elements as mathematical expressions, we need a way to express these musical elements as numbers. Pitch will be represented using the MIDI notation, while rhythm will be expressed using rhythm-trees.

[1] http://repmus.ircam.fr/openmusic/home
[2] https://www.ircam.fr
[3] https://www.gecode.org
[4] https://github.com/blapiere/Rhythm-Box

1.2 Contributions

This thesis has many contributions, that can be organised in three different categories :

Developing GiL, an interface between Gecode and Lisp The first contribution is the addition of a variety of Gecode features to GiL[5], to allow for more complex and interesting Constraint Satisfaction Problems (CSP) in Lisp. The work done in [11] was the start of an interface between Gecode and Lisp, and though it was complete enough to create CSPs in Lisp, it was lacking several features to give more flexibility to the user. The following features were added:

- Multiple branching strategies: before, there were only one default strategy for variable and value selection. GiL now has a few possibilities for both, and it is easy to add strategies that you need if they are not already supported.
- Search engines : additions were made to the search engine wrapping to make it more flexible, and a function to detect if the search has been stopped has been added as well.
- Search options : The possibility to add a time limit to the search and to specify the number of threads to use has been added to GiL as well, making it easier to customize the search for a specific problem. Other options can be easily added due to the design of the wrapping.
- Constraints : A few additional constraints have been added to GiL, making its catalogue of constraints even larger.
- Bugs : Finally, some bugs were found and fixed to make GiL as reliable as possible.

A first approach to a Constraint Satisfaction Problem for music The second contribution of this master's thesis is a constraint satisfaction problem that can find solutions respecting a set of musical constraints. It is implemented in Lisp using GiL. It takes as input a chord sequence and a rhythm for a melody. It then applies basic musical constraints to make good-sounding melodies that can be played on top of the input chords without restraining the set of solutions too much. The CSP in itself contains multiple contributions :

- A set of musical constraints allowing the composer to customize the solutions while having some mandatory constraints making sure the melody will sound good. These constraints include : a constraint forcing the notes to be in a given tonality, a constraint restraining the interval between two consecutive notes, a set of constraints giving the melody a general direction, and more.
- A searching strategy that allows to have a wide variety in the solutions using a branch-and-bound search engine to post a constraint when a solution is found.
- Control of the variety of the solutions through the interface so the composer can adapt it to their needs.
- It often happens that the solver takes a large amount of time. We have introduced a technique where the user (the composer) can stop the solver after a desired time.

Melodizer, a tool in OpenMusic to assist composers in their creation process The third and final contribution of this master's thesis is an interface allowing to use the constraint satisfaction problem depicted above. It does not aim to generate ready-to-use music but rather to stimulate composer's creativity and opening his or her mind to new horizons. Here are some of its main features :

- Melodizer provides a large number of constraint-based tools for the composer, allowing them to customize the problem.
- Melodizer allows the composer to easily navigate through and interact with the different solutions.
- Melodizer allows the composer to keep interesting results and use them in further exploration. There are three levels in which results can be stored, namely Motifs, Phrases, and Periods. [10]
- Melodizer offers a way to export results out of the interface to be used with other components of OpenMusic.

[5] https://github.com/sprockeelsd/GiLv2.0

1.3 Melodizer Example

Figure 1.1 shows an example of one of many things that can be done using Melodizer. In this particular example, a composer takes an already existing song, "Puisque tu pars" by Jean-Jacques Goldman, and wants to find a different melody to play on top of the same chord sequence. To do so, they separate the melody from the chords and give the melody's rhythm along with the chords as input to the Melodizer tool. They can select the tonality they want for the solution (D major in this case) and then start the search. One of the solutions generated by Melodizer is displayed at the bottom of the figure. The chords, rhythm, original song and new variation can be played by scanning the QR code.

1.4 Roadmap

This thesis covers the following topics:

- Chapter 2 covers the basic knowledge necessary to understand the discussion in this master's thesis. It starts with a few notions of music theory to allow the reader to understand the musical constraints as well as the examples. It also covers the basics of constraint programming, explaining what it is and the main components of a constraint satisfaction problem. Finally, it talks about Gecode, the framework for constraint programming, and OpenMusic, the software in which Melodizer can be used.

- Chapter 3 covers the model used for the constraint satisfaction problem used in Melodizer. It explains what variables are used, and what are their initial domains. It then details every constraint, optional and non-optional, that are supported in Melodizer, and discusses the branching as well.

- Chapter 4 describes the search strategy for solutions, as well as a discussion on how to improve the variety of the solutions that are suggested to the user and how to make the search faster. It also covers how and why the search is separated from the rest of the tool. It concludes with a concrete example of a musical CSP using the model of chapter 3 and the search described in this chapter.

- Chapter 5 talks about the design and implementation of the Melodizer tool in OpenMusic. It discusses the representation used for the musical elements, and gives a reminder on Voice objects. It then discusses the design of Melodizer, starting with the attributes of the melodizer object. After that, it discusses the melodizer object editor, which is the interface for the user. Finally, it details the implementation of the object and the interface.

- Chapter 6 explains all the components of the interface of Melodizer, and gives a tutorial on how to use it. It starts by a tutorial on how to set up the environment, and details the different steps of creating and solving a musical problem. It explains how to give input to the object to set the parameters of the problem, how to add optional constraints, how to search for and modify solutions, and how to assemble different solutions into one musical piece. Finally, it explains how Melodizer can interact with other features of OpenMusic inside a patch.

- Chapter 7 shows some simple examples of composer use scenarios for Melodizer. These are actual situations that composers might find themselves in. It explains the scenario, provides visual representation of the input and output of Melodizer, and provides a small video showing the result produced by the tool. It concludes with a video tutorial on how to use the main features of Melodizer. If you are a composer, we suggest you start with this chapter.

- Chapter 8 talks about the interface between Gecode and GiL. It first talks about the structure of the library, as well as the features already in GiL before the start of this master's thesis and how they are wrapped. It then talks about this thesis' contribution and gives an example of how to use GiL in a side-by-side comparison with Gecode. Finally, it explains how to add other features to GiL.

- Chapter 9 summarizes the work of this master's thesis and gives a few ideas on what can be done to improve Melodizer and GiL.

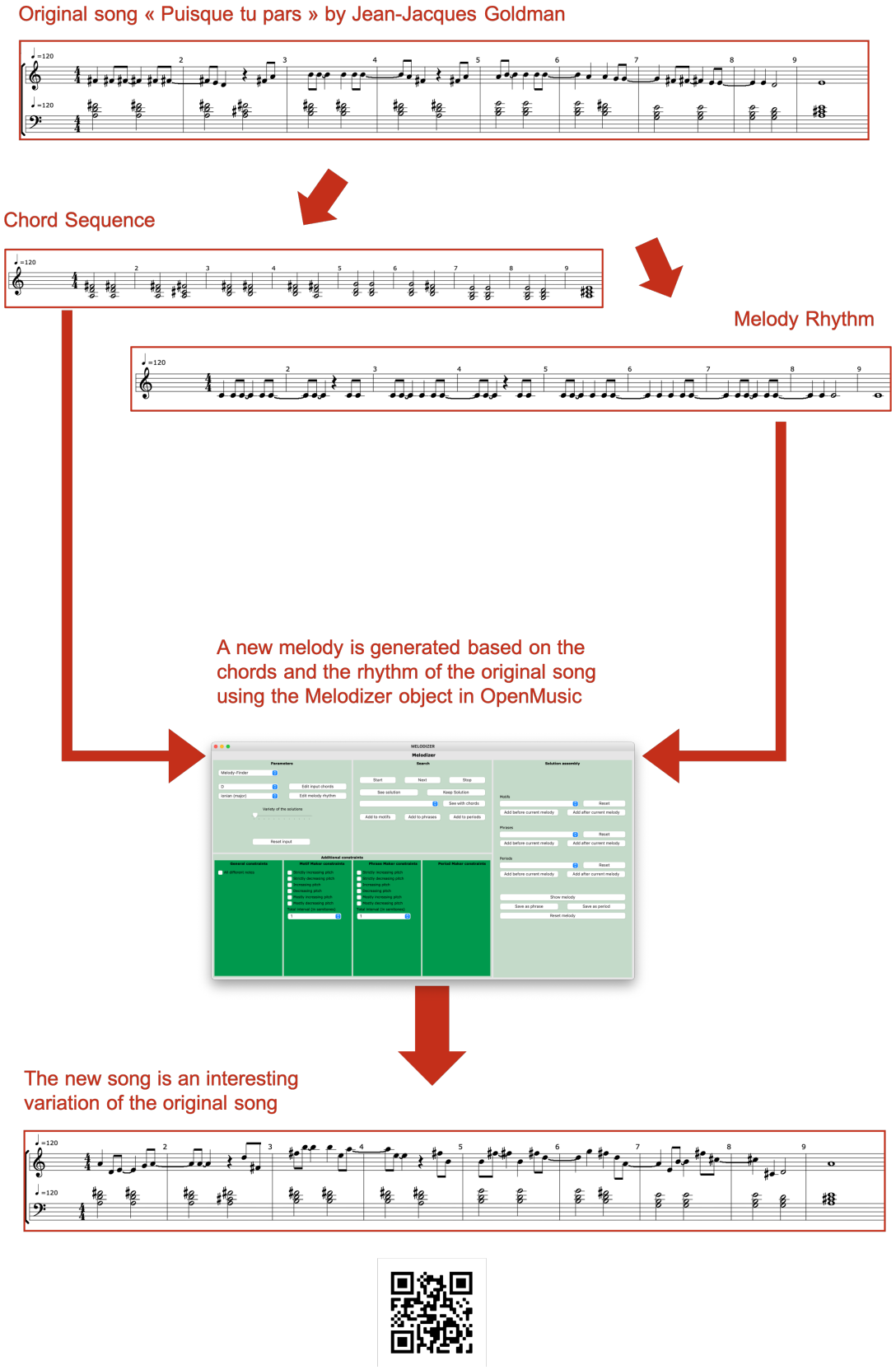


Figure 1.1: Example of the use of Melodizer to generate a different melody with the same rhythm for “Puisque tu pars” by Jean-Jacques Goldman [9]

Chapter 2

Theoretical background

This chapter introduces the fundamental notions necessary to understand the discussions of this master's thesis. It will first cover the basics of music theory, including the different symbols used to represent music, a definition of tonality and a description of a few types of chords. It will then discuss constraint programming, and explain models and a few different types of search. It will also talk a little bit about Gecode, the constraint programming framework used for the constraint-programming part of this thesis, and explain how to use it to solve CSPs. Finally, it will talk about OpenMusic, the software in which Melodizer can be used, and discuss the main components of an OpenMusic project.

2.1 Musical notions

In order for the reader to understand the musical aspect of this master's thesis, basic notions of occidental music theory will be briefly explained in this section, including vocabulary and basics on reading music. This section is not a music theory class, therefore it will only discuss relevant concepts in the context of this master's thesis. The curious reader can read [6] for more information on music theory.

2.1.1 Staff

A staff is the most common way to represent notes with their pitch in occidental music. It consists of five horizontal lines, evenly spread, counted from the bottom up. A staff is divided in bars, according to the time signature. A variety of symbols can be found on staves, the most important ones will be discussed here.

Notes Notes have two main characteristics : a duration and a pitch. Duration is expressed in the shape of the note, while pitch is expressed in the position of the note in a staff, with respect to its clef. Figure 2.2 shows the most commonly used shapes for durations and the relations between them. The whole note, on top of the figure, is usually used as a reference. There are only seven names used to express the pitch of a note, but alterations can be used to describe all twelve notes of occidental music. Those names are A, B, C, D, E, F, G. The twelve notes used in occidental music are depicted in figure 2.1.

Clef There are two clefs that are most commonly used, the G clef and the F clef. The G clef, found on the first staff of figure 2.1, is typically used to represent higher pitch. The swirl in the center of the clef wraps around the second line, that represents the G. Every step up or down, to a space or another line, goes to the
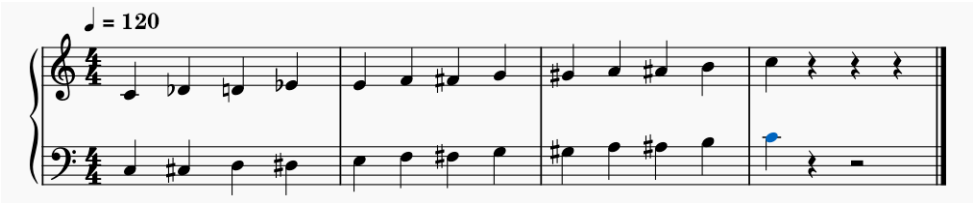


Figure 2.1: Example of 2 staves with different notes and symbols

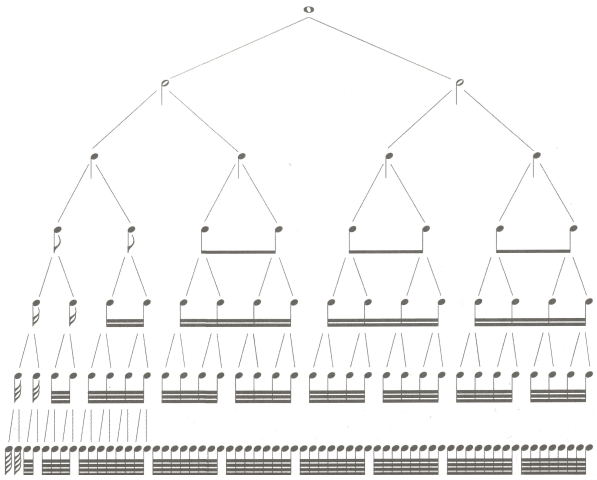


Figure 2.2: Relationship between the different shapes of notes [6]

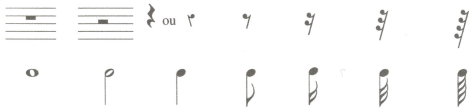


Figure 2.3: Equivalence between the length of rests and the corresponding note shape [6]

next note. For example, the space between the second and third line represents an A, one note above G, and the third line represents a B, one note above A and two notes above G.
The F clef, found on the second staff of figure 2.1, is used to represent lower pitch. The two dots are around the fourth line of the staff, that represents an F. A step up or down works the same way as with the G clef. The space between the third and fourth lines represents an E, and the third line represents a D.

Rests Rests are symbols that represent the absence of a note. They are related to the different shapes of notes discussed before, and figure 2.3 shows the relation between the shapes of notes and the rests.

Alterations There are three main alterations used to modify the pitch of a note: the "sharp" (♯), the "flat" (♭) and the "natural" (♮). The sharp symbol moves the pitch up by a semitone, the flat symbol moves the pitch down by a semitone, and the natural symbol cancels all previous alterations.

Other rhythmic symbols There are symbols that can be used on a note to alter its duration, in addition to the shape of the note. A note can be "dotted", meaning it has a dot on its right, and that increases the note's duration by half. Two notes having the same pitch can also be linked together, meaning that the duration of the note is the duration of the two notes linked together. Finally, three notes of the same shape can be linked together with a "3" on the line linking them, and that will mean that these 3 notes combined will have a duration equivalent to only two of the notes. This is called a triplet. Figure 2.4 shows an example of use of these rhythmic symbols.

Time signature The time signature gives the rhythmic duration of a measure. It is composed of two numbers, one above the third line of the staff and one below the third line of the staff. The first number gives the number of beats in a bar while the second number gives the duration of a beat with respect to a whole note. For example, a time signature of (3 4) means that a measure has the length of three quarter notes.

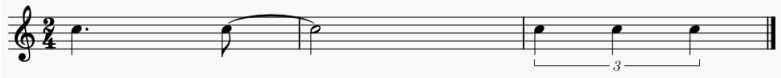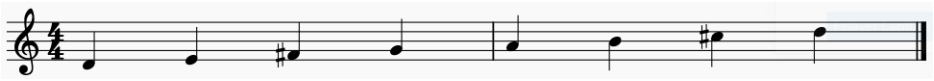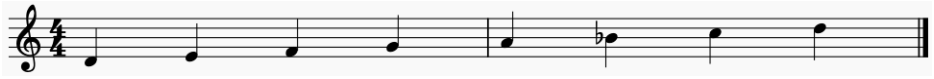


Figure 2.4: Example use of doted notes, linked notes and triplet
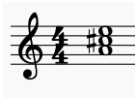


(a) Notes in the D major tonality
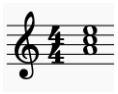


(b) Notes in the D minor tonality
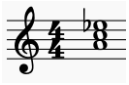
Figure 2.5: Comparison between the D major and D minor scales
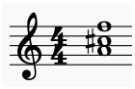


(a) A major chord
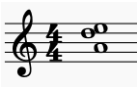


(b) A minor chord



(c) A diminished chord



(d) A augmented chord



(e) A suspended fourth chord

Figure 2.6: Different types of A chords

Tempo The tempo translates rhythmic duration into time. It is usually expressed in beats per minute (bpm). For example, figure 2.1 has a tempo of 120bpm, meaning that each beat is half a second long. Since the time signature is (4 4), it means that a quarter note is half a second long.

2.1.2 Tonality

A tonality is defined by two elements : a key and a mode. The key is one of the 12 notes, and is used as a base to build the tonality with the mode. The mode specifies the sequence of tones and semitones that make the tonality. For example, the major mode is defined as the succession of two tones, one semitone, three tones and one semitone. There are many modes, but the most common are the major mode and the minor mode. Figure 2.5 shows the notes used in D major (figure 2.5a) and D minor (figure 2.5b).

2.1.3 Chords

Chords are identified by a root note and a type, that can be recognized by the intervals between the notes of the chord. This section will describe the most common three notes chords.

Major chords A major chord is composed of a major third (4 semitones) and a major fifth (7 semitones) from the root (figure 2.6a).
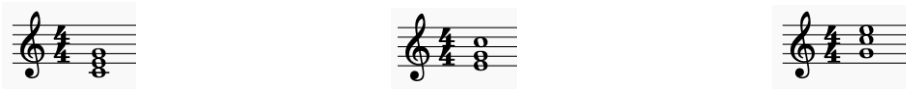
Minor chords A minor chord is composed of a minor third (3 semitones) and a major fifth (7 semitones) from the root (figure 2.6b).

Diminished chords A diminished chord is composed of a minor third (3 semi-tones) and an augmented fourth (6 semitones) from the root (figure 2.6c).

Augmented chords An augmented chord is composed of a major third (4 semitones) and an augmented fifth (8 semitones) from the root (figure 2.6d).

Suspended fourth chords A suspended fourth chord is composed of a major fourth (5 semitones) and a major fifth (7 semitones) from the root (figure 2.6e).

Inversions Three notes chords have three possible forms : the fundamental, the first inversion and the second inversion. These inversions are represented in figure 2.7 for the C major chord.



(a) Fundamental form of the C major chord (b) First inversion of the C major chord (c) Second inversion of the C major chord

Figure 2.7: Different forms of the C major chord

2.2 Constraint programming

This section will explain what constraint programming is, and what it can be used for. It will cover the important elements such as how to model a problem to use constraint programming to solve it, and how a solver works. It will also talk about selection heuristics and provide a concrete example of the use of constraint programming to solve a sudoku.

2.2.1 What is constraint programming?

Constraint programming is a paradigm for solving combinatorial problems. Users state the constraints that a set of decision variables must follow, and they specify guidelines on how to search for a solution. They do not have to worry about solving the problem, the solver is doing the work. Constraints differ from imperative programming in that they do not specify a sequence of steps to follow but rather a "property" of a solution. A good model is therefore really important to maximize the chances of finding a solution relatively fast [5].

Constraint Satisfaction Problem (CSP) A CSP is a triplet $< X, D, C >$ where X is the set of decision variables, each with domain $D(x)$, and C is a set of constraints. A solution to a CSP is a candidate solution (assign to each variable a value from its domain) that satisfies all the constraints in the CSP [13].

Constraints are used to reduce the domain of variables. That process is called constraint propagation. In constraint programming, the user focuses on modeling the problem, and not so much on finding the solution. That is the responsibility of the solver. The solver searches for solutions by assigning values from their domain (updated by constraint propagation) to variables, and checking if they still satisfy all constraints. The user can specify how the search is done by using different types of solvers, and they can specify how the variables and values from the domain are selected by defining search heuristics. The quality of the model can often make the difference between finding a solution or not.

Solvers There are many different solvers, so we will only cover the ones that are relevant for this master's thesis. The first solver is the depth-first search engine. It works by selecting the first variable according to the variable heuristic, then selecting the value for that variable according to the value heuristic, and so on. The second search-engine we will see is the branch-and-bound search engine. It works like the depth-first search engine in Gecode (see section 2.3), except that when a solution is found it posts a new constraint that is specified by the user. It is very useful when we are looking for the best solution for example.

Variable selection heuristics There are many ways to choose the order in which we should select the next variable to assign, that are more or less effective depending on the problem. Here are a few possible heuristics :

- Select the variable who's current domain is the smallest
- Select the variable that has the biggest degree[1]
- Select the variable randomly

[1] the degree of a variable is the number of constraints where this variable appears

Value selection heuristics There are also many ways to choose the value that will be assigned to a variable, some of them are listed below:

- Select the smallest value of the domain
- Select a random value in the domain
- Split the domain and keep all the values not greater than the mean of the smallest and largest value
- Split the domain and keep all the values greater than the mean of the smallest and largest value
- Select the greatest value that is not greater than the median

2.2.2 Example of a CSP

Figure 2.8 shows a CSP written in C++ with Gecode to solve a sudoku. The constraints applied are :

- $\forall i, j \in [1, 9]\ st\ i \neq j\ \ x_{i,k} \neq x_{j,l}$, all the tiles on a given row must have a different values
- $\forall k, l \in [1, 9]\ st\ k \neq l\ \ x_{i,k} \neq x_{j,l}$, all the tiles on a given column must have different values
- $\forall i, j, k, l \in [1, 9]\ st\ \frac{i}{3} = \frac{j}{3}\ and\ \frac{k}{3} = \frac{l}{3}\ \ x_{i,k} \neq x_{j,l}$ all the tiles in a given square must have different values.

where $x_{i,j}$ is the tile at position (i,j) in the grid. The main steps are highlighted in blue, including :

- The creation of the variables
- The posting of the different constraints stated above
- The specification of the branching heuristics (largest accumulated failure count by domain size for the variables, smallest value for the values)
- The specification of the solver (Depth-First Search)

2.3 Gecode

Gecode [14] is an open source C++ toolkit for developing constraint-based systems and applications. It is very comprehensive, efficient and flexible, which makes it a perfect tool for this master's thesis. This section will present the most important features of Gecode. Figure 2.8 shows an example of a CSP to solve a sudoku using Gecode.

Space In Gecode, every CSP is modelled by creating a class extending the Space class. Variables are created in the constructor of the class, and constraints are posted in the constructor too. The branching is also specified in the constructor. A search engine is then created in another function (e.g. a main function) to search for solutions of the problem. It takes an object representing the problem as an argument.

Variables Four types of variables are supported in Gecode: Integer variables, boolean variables, float variables and set variables. They are created by calling the corresponding constructor, with the first argument always being the current space. Gecode also supports arrays of variables, and argument variables (intermediate variables that are not updated during the search).

```
1    IntVarArray vars(*this,5, 1, 10); // create an array of 5 integer variables with a
     domain of [1,10]
```

Constraints Constraints are posted in the space using a predefined function that also takes the space of the problem as the first argument. There is a wide variety of constraints in Gecode, the full list of constraints is available on the Gecode website [2].

```
1    distinct(*this, vars); // post the all-different constraint on the IntVarArray vars
```

[2] www.gecode.org

```
class Sudoku : public Script {
protected:
  /// The size of the problem
  const int n;
  /// Values for the fields
  IntVarArray x;
public:
  /// Constructor
  Sudoku(const SizeOptions& opt)
    : Script(opt),
      n(example_size(examples[opt.size()])),
       x(*this, n*n*n*n, 1, n*n)  {// define the variables
    const int nn = n*n;
    Matrix<IntVarArray> m(x, nn, nn);
    // Constraints for rows and columns
    for (int i=0; i<nn; i++) {
     distinct(*this, m.row(i), opt.ipl());
     distinct(*this, m.col(i), opt.ipl());
    }
    // Constraints for squares
    for (int i=0; i<nn; i+=n) {
      for (int j=0; j<nn; j+=n) {
       distinct(*this, m.slice(i, i+n, j, j+n), opt.ipl());
      }
    }
    // Fill-in predefined fields (post constraints)
    for (int i=0; i<nn; i++)
      for (int j=0; j<nn; j++)
        if (int v = sudokuField(examples[opt.size()], nn, i, j))
              rel(*this, m(i,j), IRT_EQ, v );
     branch(*this, x, INT_VAR_AFC_SIZE_MAX(opt.decay()), INT_VAL_MIN());
  }
...
}
// \brief Main-function
int
main(int argc, char* argv[]) {
  ...
   Script::run<Sudoku,DFS,SizeOptions>(opt);  // specify the solver (Depth-First Search)
  return 0;
}
```

Figure 2.8: Example of a CSP to solve a Sudoku using Gecode [14]

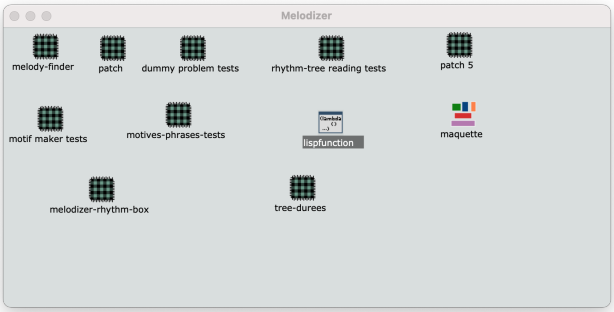


Figure 2.9: Example of an OpenMusic workspace

Branching Branching strategies are expressed using the branch function. They are also posted in the space. Gecode provides a wide variety of variables and value selection heuristics, a full list can be found in [14].

```
branch(*this, vars, INT_VAR_SIZE_MIN(), INT_VAL_MIN()); // select the variable with the smallest domain and select the smallest value of the domain
```

Search Search engines are created as instances of a class. There are three types of search engines in Gecode : Depth-First Search engines (DFS), Branch-And-Bound Search engines (BAB), and Limited Discrepancy Search-Engines (LDS). They take as argument an instance of the class of the problem.

```
Problem* problem = new Problem(); // create an instance of the problem
DFS<Problem> se(problem); // create the search engine (DFS) for the problem
```

Search options Options for the search can be specified to a search engine, including the number of threads to use for parallel search, a certain limit to stop the search and more. They are passed to the search engine as an argument through a Gecode::Search::Options object that has a field for every option.

```
Gecode::Search::Options opts; // create the options object
opts.threads = 0;// use as many threads as possible
Gecode::Search::TimeStop maxTime(1000); // specify a time limit of 1000 ms
opts.stop = &maxTime; // add it to the options
DFS<Problem> e(problem, opts); // give the options to the solver
```

2.4 OpenMusic

OpenMusic (OM) is a visual programming language for computer-assisted music composition created at IRCAM [12]. It is build on top of Common Lisp, and can be used to create any type of programs though it provides a variety of features and libraries to be used as a musical composition environment. OpenMusic is composed of many elements, the most important ones in the context of this master's thesis are detailed in this section.

Workspace The first interface that appears when you start an OM project is called the workspace, and it can contain maquettes, patches and lisp files. Figure 2.9 shows an example of a workspace containing patches, lisp files and maquettes. This thesis only uses patches, so we will not discuss maquettes and lisp functions any further.

Lisp listener The Lisp listener provides the result of the evaluation of boxes, information when errors occur,... It allows the user to see the result of an evaluation.

Patch A patch is the equivalent of a program. It contains data, objects and functions, connected to each other through inlets and outlets that represent the data flow. Figure 2.10 shows a patch with a simple program to create a voice object.

2.4.1 Boxes

Boxes are the main components of a patch. They can be anything from values (numbers, lists,...) to functions or objects. They can have inlets allowing them to receive input and outlets allowing their values to be used. Figure 2.10 shows the different types of boxes.

Data boxes They are the simplest boxes. They represent primitive data types and do not have any inlets. They only have one outlet allowing their value to be used.

Function boxes They represent functions. The inlets represent the input and the outlets are the return values of the function. Evaluating a function box is equivalent to calling the function and will display the result in the listener.

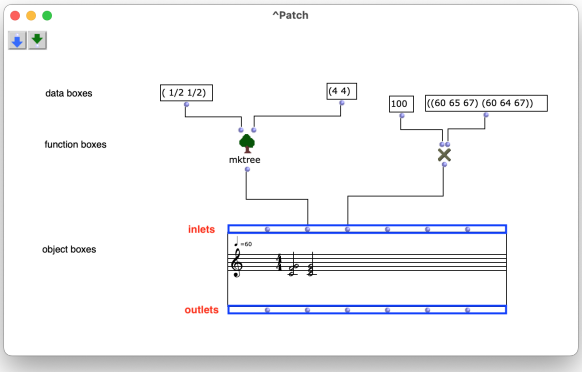


Figure 2.10: Simple patch showing the different boxes

Object boxes Object boxes in OpenMusic are factories to produce instances of that object. Their first inlet and outlet are always a reference to the object itself, and other inlets and outlets are the setters and getters respectively.

Score boxes Score boxes are a group of objects that are related to the musical aspect of OpenMusic. They have an editor that can be used to modify them and play them.

Voice and poly objects Voice and poly objects are important for this master's thesis, so they will be discussed in more detail. Since a poly object is just a list of voice objects, and the way they can be edited is the same, we will only discuss poly objects. Figure 2.11 shows the interface of the poly object.
Voice objects represent a musical voice on a staff, with a tempo, a time signature, a rhythm and a pitch. The main attributes are tree, the rhythm-tree of of the voice, chords, the list of pitches for the notes, and tempo, the tempo of the voice. Each of these attributes can be set by connecting a data box to the appropriate inlet of the voice object, or they can be edited from the editor of the object by adding notes, changing their duration, and so on. Modifications to the voice can be done using the editor, the main ones are listed below[3] :

- The pitch of a note can be adjusted by selecting it using the note tool or the chord tool and dragging it up/down or using the arrow keys of the keyboard.
- Measures : Measures can be added by clicking on the measure tool and then clicking while pressing the "Cmd" key down where the new measure needs to be, or deleted by selecting it and pressing the "backspace" key.
- A rest can be transformed into a note and vice versa by selecting it using the chord tool and pressing the "esc" key.
- An event can be split into a group of smaller events of equal value by selecting it using the chord tool and entering a number representing the number of smaller events that need to replace this event.
- Notes can be added to a chord by selecting it using the note tool or the chord tool and pressing the "cmd" key.
- A voice or measure can be transposed by selecting it using the voice tool or the measure tool respectively, and using the arrow keys.

[3] for more information on how to edit voice and poly objects, see https://support.ircam.fr/docs/om/om6-manual/co/Editor-Rhythm.html

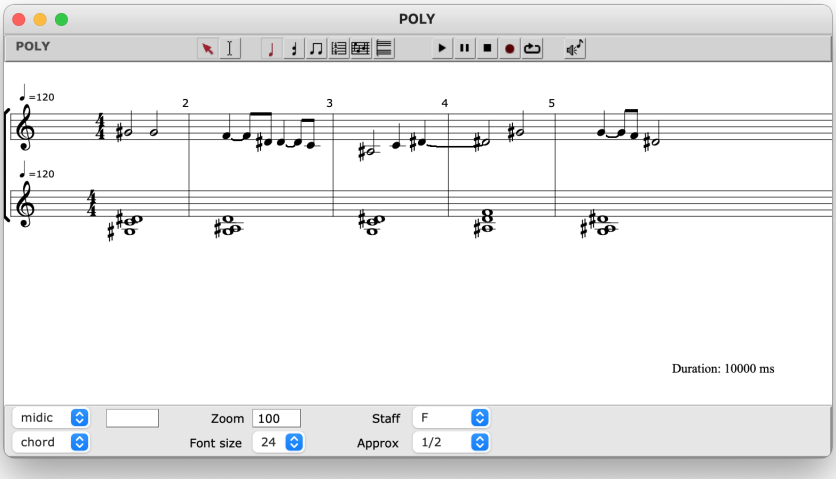
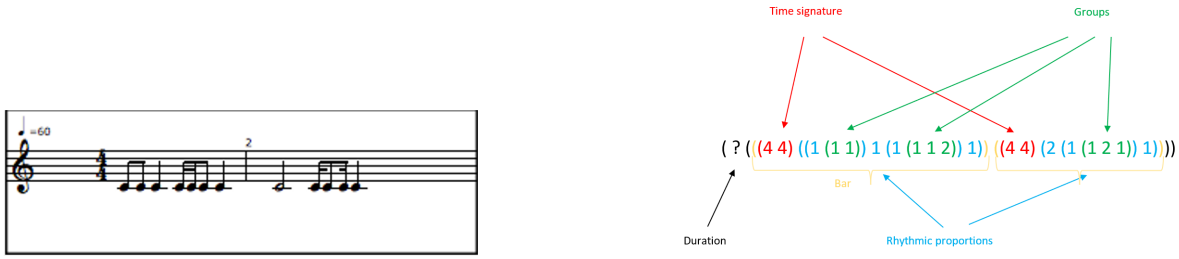


Figure 2.11: Poly object representing the melody and chords of "Fix you" by Coldplay [4]



(a) A simple rhythm

(b) The corresponding rhythm tree

Figure 2.12: A simple rhythm and the corresponding rhythm tree

Box evaluation Connecting boxes is like writing a program, but it doesn't run it. To do so, one must evaluate the boxes. Once a box is evaluated, all the boxes connected directly or indirectly to the inlets of the evaluated box are also evaluated, unless they are locked. They are evaluated from the top of the tree-like structure to the bottom, and from left to right. If a box should not be evaluated, it can be locked by selecting it and pressing the "b" key.

2.4.2 Representation

OpenMusic is using MIDIcent representation for the pitch of the variables, and rhythm trees to represent rhythm. Those notions are detailed in this section.

MIDI and MIDIcent representation MIDI representation encodes the pitch of notes with seven bits, allowing to represent notes ranging from C-1 to G9. C4 has the value of 60, and adding or removing one is equivalent to a pitch that is one semitone higher or lower. OpenMusic uses MIDIcent representation, and a pitch in MIDIcent is the pitch in MIDI multiplied by a hundred. This allows to represent more pitches than the traditional tones and semitones.

Rhythm trees Rhythm trees are a a list of elements that define a rhythm. The first element is the total duration of the rhythm, if it is unknown it is a "?". Then follows a list of bars, each of the form (time-signature events). Within a bar, if there are events shorter than one beat, they are expressed as a group of events. Figure 2.12 shows a simple rhythm and the corresponding rhythm tree.

Chapter 3

Model

This chapter will cover the model used for the musical CSPs. Two different CSPs were developed in the context of this master's thesis, but they only differ in small parts and will be discussed together. Parts specific to one CSP will be referred to explicitly. This section will describe the model of the musical problem, including the decision variables, the different constraints (optional and mandatory) that are used to shape the melody, and the different branching strategies that are used to efficiently find a solution.

3.1 Variables and their domains

The choice has been made to focus the attention on pitch only and to rely on other sources (RhythmBox [11] or the composer) to generate the rhythm. Variables and constraints will therefore only consider pitch. To represent pitch, the MIDI representation will be used, even though OpenMusic uses the MIDIcent representation. This is motivated by the fact that using MIDIcent notation would result in much larger domains in terms of size, with a lot of unused values, and would massively decrease the performance of the tool as a result.

Pitch variable array The first array of variables used in the CSP is an array to represent the pitch of the notes from the melody that the user wishes to find. It is of size n, where n is the number of musical events in the input rhythm, and is an array of integer variables. They have a domain ranging from 0 to 127, the maximal range of the MIDI notation, though in practice it is very unlikely that notes in a melody will cover that range. The domain is kept as generic as possible to focus all the restrictions in the constraints part. Note that with this domain, it is impossible for the CSP to assign an absence of value to a given note. That is because rests are considered to be part of the rhythm, and they are therefore not part of the CSP. The only rests in a solution will be the ones that are in the input rhythm.

Intervals variable array The second array of variables used in the CSP is an array to represent the intervals between consecutive notes. It is of size n-1, where n is the number of musical events in the input rhythm, and is also an array of integer variables. Their domain ranges from -127 to 127, where a negative value represents an interval going down in pitch. It is kept as large as possible to keep all the restrictions in the constraints part, even though such large intervals are never used. It is connected to the pitch variable array through a constraint described in section 3.2.1.
This second array is not strictly necessary since all of its elements can be expressed with variables from the pitch array, however it makes it a lot easier to post constraints on intervals between notes. It also has the advantage of allowing us to branch on intervals as well, which is convenient since a significant portion of the constraints are related to intervals. This results in a much better overall performance than only using the pitch variable array.

3.2 Musical constraints

This section will cover the design and implementation of all the constraints used in the CSPs. Some constraints are always applied, some are optional, and some are only used in one of the two CSPs. Calls to

Default constraints (both CSPs)	Melody-finder CSP	Variator CSP
In tonality (3.2.2) Note on chord (3.2.3) Interval between adjacent notes (3.2.4) Harmonic interval chord (3.2.5) Range restriction (3.2.6)	Melody direction (opt) (3.2.7) All different notes (opt) (3.2.8)	At least n (3.2.9)

Figure 3.1: The category of the different constraints

```
1 ; temporary variables to make it easier to apply the linear constraint
2 (setq temp (gil::add-int-var-array sp 3 -12 108))
3
4 (gil::g-rel sp (first temp) gil::IRT_EQ (nth j pitch)); temp[0] = pitch[j]
5 (gil::g-rel sp (second temp) gil::IRT_EQ (nth (+ j 1) pitch)); temp[1] = pitch[j+1]
6 (gil::g-rel sp (third temp) gil::IRT_EQ (nth j intervals)); temp[2] = intervals[j]
7 ; -pitch[j] + pitch[j+1] - intervals[j] = 0
8 (gil::g-linear sp '(-1 1 -1) temp gil::IRT_EQ 0)
9
```

Figure 3.2: Code linking the pitch and interval arrays together

GiL are highlighted in the code snippets. The currently supported constraints are :

- In tonality constraint (3.2.2) : Ensures that the notes are in the given tonality
- Note on chord constraint (3.2.3) : Ensures that a note played at the same time as a chord follows a set of rules based on the type of the chord
- Interval between adjacent notes constraint (3.2.4) : Ensures that the interval between consecutive notes follows a set of rules
- Harmonic interval chord constraint (3.2.5) : Ensures that the notes in the context of a given chord have a maximal interval of an octave
- Range restriction constraint (3.2.6) : Restrains the ambitus of the melody
- Melody direction constraints (3.2.7) : Give a general direction to the melody
- All different notes constraint (3.2.8) : Ensures that all the notes are different
- At least n constraint (3.2.9) : Ensures that at least n notes are equal to their value in a given melody

Table 3.1 shows what constraints are default constraints, and shows what constraints are used for each CSP.

3.2.1 Relation between pitch and intervals

The pitch variable array and the intervals variable array are connected together using the following formula:

$$\forall i \in [0, n[\ intervals[i] = pitch[i+1] - pitch[i]$$

where n is the length of the pitch variable array. It means that any interval is the difference between the second note and the first note of that interval. Figure 3.2 shows how this formula is applied in Lisp using constraints.

3.2.2 In tonality constraint

This constraint ensures that, given a certain tonality represented by a key (A to A♭) and a mode (see figure 3.3), all the notes from the solution will be in that tonality. Notes belonging to a tonality can be deduced from their key and mode as a series of tones and semitones starting from the key and repeating over and over. In MIDI notation, a semitone is 1 and a tone is 2. We can then find all the notes from a given key using the following formula :

$$dom(X) = N_{tonality}$$

mode	series of tones/semitones
major (ionian)	(tone / tone / semitone / tone / tone / tone / semitone)
dorian	(tone / semitone / tone / tone / tone / semitone / tone)
prhygian	(semitone / tone / tone / tone / semitone / tone / tone)
lydian	(tone / tone / tone / semitone / tone / tone / semitone)
myxolydian	(tone / tone / semitone / tone / tone / semitone / tone)
natural minor (aeolian)	(tone / semitone / tone / tone / semitone / tone / tone)
locrian	(semitone / tone / tone / semitone / tone / tone / tone)
harmonic minor	(tone / semitone / tone / tone / semitone / 3 semitones / semitone)
pentatonic	(tone / tone / 3 semitones / tone / 3 semitones)
chromatic	(semitone / semitone / semitone / semitone / semitone / semitone / semitone / semitone / semitone / semitone / semitone / semitone)

Figure 3.3: Sequence of tones and semitone defining the supported modes

where $N_{tonality}$ is the set of notes from a specified tonality.

Figure 3.3 shows the tone and semitone sequence for the supported modes. The following Gecode constraint is then posted for each of the variables:

```
1 dom(home, IntVar variable, IntSet domain);
```

where home is the computation space, variable is an integer variable and domain is the set of notes that are in the tonality specified. Figure 3.4 shows the code for this constraint.

3.2.3 Note on chord constraint

This constraint ensures that, given a list of three note chords (in their fundamental form or their inversions), the notes from the melody that are played at the same time as those chords will respect a set of rules depending on the type of the chord.

Finding notes played on top of a chord The first step is to identify which notes from the melody are played on top of chords. To do so, lists of starting times for the chords and for the notes of the melody are computed. We then iterate through the list of chords starting times, and for each chord we iterate through the list of notes starting time from the melody that have not been explored yet to check if one of them is played simultaneously with the current chord. If there is a note who's starting time corresponds to the chord's starting time, a constraint is posted on the domain of that variable according to table 3.6, and if it finds a note that is played after the chord it switches to the next chord.

Identifying the chord Once we know that a given note is played on top of a chord, we must be able to identify it in order to decide what values that note can take. To do so, the first step is to identify the type of the chord and its inversion. This can be done using the intervals between the different notes of the chord using table 3.5. Once we know the type of the chord and the inversion, we are able to identify the chord. [8] gives us rules for the notes that can be played on top of a given chord depending on it's type. They can be found in table 3.6. The Gecode constraint used is

```
1 dom(home, IntVar variable, IntSet domain);
```

where home is the computation space, variable is the integer variable representing the note played on top of the chord, and domain is the domain according to the rules stated above.
Additionally, a constraint is posted so that the interval between the highest note of the chord and the note from the melody is at most 1 octave. The Gecode constraint used is

```
1 rel(home, IntVar variable, rel-type, int number);
```

where home is the computation space, variable is an integer variable, rel-type is a relation and number is an integer. Figure 3.7 shows the code for this constraint. The code of some auxiliary functions will not be presented here as it is too long, but it can be found in appendix C.

```
; <sp> is the space,<notes> is the variable array on which the constraint is executed
; <key> is the key,<mode> is the mode
(defun in-tonality (sp notes key mode)
    (let (admissible-notes)
        ; call the function to get the list of notes in that tonality
        (setf admissible-notes (notes-from-tonality key mode))
        ; set the domain of each variable to the set of notes from the tonality
        (loop :for j :from 0 :below (length notes) :do
            (gil::g-dom sp (nth j notes) admissible-notes)
        )
    )
)
; returns the set of notes that are in the tonality specified by the arguments
(defun notes-from-tonality (key mode)
    (let (admissible-notes note scale i)
        (setq scale (get-scale mode)); get the scale in semi-tones
        ; then, create a list and add the notes in it
        (setq note key) (setq i 0)
        (setq admissible-notes (list))
        ; add all notes above the key, then add all notes below the key
        (om::while (<= note 127) :do
            (setq admissible-notes (cons note admissible-notes)); add it to the list
            (if (>= i (length scale)) (setq i 0))
            (incf note (nth i scale)); note = note + scale[i mod 6]
            (incf i 1); i++
        )
        (setq note key) (setq i 1)
        ; note = note - scale[6-i mod 6]
        (decf note (nth (- (- (length scale) 1) 0) scale))
        (om::while (>= note 0) :do
            (setq admissible-notes (cons note admissible-notes)); add it to the list
            (if (>= i (length scale)) (setq i 0))
            ; note = note - scale[6-i mod 6]
            (decf note (nth (- (- (length scale) 1) i) scale))
            (incf i 1); i++
        )
        admissible-notes
    ))
(defun get-scale (mode)
    (cond
        ((string-equal mode "ionian (major)") (list 2 2 1 2 2 2 1))
        ((string-equal mode "dorian") (list 2 1 2 2 2 1 2))
        ((string-equal mode "phrygian") (list 1 2 2 2 1 2 2))
        ((string-equal mode "lydian") (list 2 2 2 1 2 2 1))
        ((string-equal mode "mixolydian") (list 2 2 1 2 2 1 2))
        ((string-equal mode "aeolian (natural minor)") (list 2 1 2 2 1 2 2))
        ((string-equal mode "locrian") (list 1 2 2 1 2 2 2))
        ((string-equal mode "harmonic minor") (list 2 1 2 2 1 3 1))
        ((string-equal mode "pentatonic") (list 2 2 3 2 3))
        ((string-equal mode "chromatic") (list 1 1 1 1 1 1 1 1 1 1 1 1))
    ))
```

Figure 3.4: code for the in-tonality constraint

	Fundamental form	First inversion	Second inversion
Major chords	4-3	3-5	5-4
Minor chords	3-4	4-5	5-3
Diminished chords	3-3	3-6	6-3

Figure 3.5: Intervals between notes from different types of chords in semitones

Type of the chord	Accepted notes
Major chords	the fundamental, the third or the fifth
Minor chords	the fundamental, the third or the fifth
Diminished chords	the third

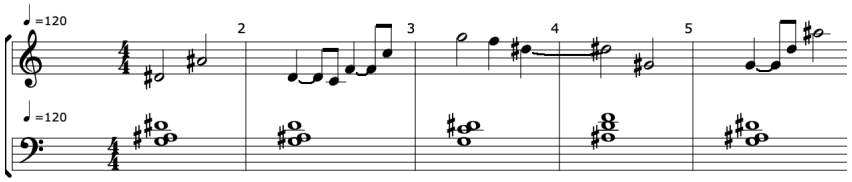
Figure 3.6: Notes that can be played on top of a given chord based on its type

```
; <sp> is the space,<notes> is the variable array on which the constraint is executed
; <input-rhythm> is the voice object of the rhythm for the melody we want to find
; <chords> is the voice object of the chords
(defun note-on-chord (sp notes input-rhythm chords)
    ; get the starting time of each of the notes of the melody and the chords
    (let ((melody-starting-times (voice-onsets input-rhythm))
        (chords-starting-times (voice-onsets chords))
        (chord-counter 0); counter to know which chord we are currently looking at
        (variable-counter 0)); counter to keep track of the note
        (dolist (c chords-starting-times); go through the chords starting times
            ;go through the input-rhythm starting times
            (dolist (m (subseq melody-starting-times variable-counter))
                (cond
                    ((< m c) ; increment the counter for variables
                        (setf variable-counter (+ variable-counter 1))
                    )
                    ((= m c) ; post the constraint on that specific variable
                        (apply-constraint-noc sp notes variable-counter chord-counter chords
    )
                        (setf variable-counter (+ variable-counter 1))
                        (return )
                    )
                    ((> m c) ; break the loop and go to the next chord
                        (return )
                    )
                )
            )
            (setf chord-counter (+ 1 chord-counter))
        )
    ))
(defun apply-constraint-noc (sp notes variable-id chord-id input-chords)
    (let ((chord-pitch (to-midi (om::lmidic (nth chord-id (om::chords input-chords))))); get
     the values of the notes of the chord
            intervals mode inversion admissible-notes)
        (sort chord-pitch #'<) ; sort the note values in increasing order
        ; get the intervals in semitones between the different notes of the chord
        (setf intervals (list
            (- (second chord-pitch) (first chord-pitch))
            (- (third chord-pitch) (second chord-pitch))))
        ; get the mode of the chord from the intervals between the notes
        (setf mode (first (get-mode-and-inversion intervals)))
        ; get the inversion of the chord
        (setf inversion (second (get-mode-and-inversion intervals)))
        ;get the notes playable on that chord
        (setf admissible-notes (get-admissible-notes chord-pitch mode))
        ; post the constraint that the domain of that variable is the admissible notes
        (gil::g-dom sp (nth variable-id notes) admissible-notes)
        ; post the constraint that the interval between the note of the melody and the
    highest note of the chord is maximum 1 octave
        (gil::g-rel sp (nth variable-id notes) gil::IRT_LQ (+ (max (first chord-pitch)) 12))
    ))
```
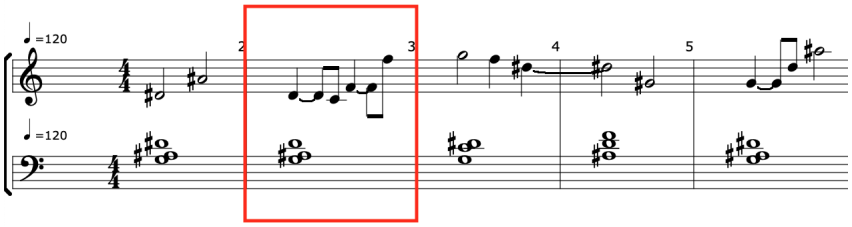
Figure 3.7: Code for the note-on-chord constraint

```
; <sp> is the space, <notes> is a list of IntVars representing the pitch of the notes
; <intervals> is a list of IntVars representing the intervals between successive notes from
    the notes argument
(defun interval-between-adjacent-notes (sp notes intervals)
    (let ((valid-intervals '(-1 -4 -5 -7 -8 -9 -12 0 1 2 3 4 5 7 8 12)))
        ;restrain the interval domains to acceptable values
        (dolist (interval intervals)
            (gil::g-dom sp interval valid-intervals)
        )
    )
)
```

Figure 3.8: Code for the interval-between-adjacent-notes constraint



(a) Example satisfying the harmonic interval chord constraint



(b) Example breaking the harmonic interval chord constraint

Figure 3.9: Examples for the harmonic interval chord constraint

3.2.4 Interval between adjacent notes constraint

This constraint ensures that the interval between two consecutive notes respects the rule given in [8], more precisely the interval can be everything up to a minor sixth, except a tritone, and can also be an octave. This can be expressed mathematically using the MIDI representation:

$$dom(Interval) \in \{-1, -4, -5, -7, -8, -9, -12, 0, 1, 2, 3, 4, 5, 7, 8, 12\}$$

The Gecode constraint used is

```
dom(home, IntVar variable, IntSet domain);
```

Where home is the computation space, variable is the integer variable representing the note played on top of the chord, and domain is the domain according to the rule stated above. Figure 3.8 shows the code for this constraint.

3.2.5 Harmonic interval chord constraint

This constraint ensures that all the notes played in the harmonic context of a given chord have a maximum interval of an octave. That is, the maximum interval between all the notes played for the duration of the chord should not exceed an octave. Figure 3.9 shows two examples: figure 3.9a is a melody that satisfies the constraint, while figure 3.9b shows a melody breaking the constraint.

The constraint's algorithm is the following: We iterate through the chord's starting times, and for each chord we iterate through the starting times of the notes from the melody, starting from where we left out with the previous chord, until we reach a note that is played in the harmonic context of another chord or the last note. Now that we have the set of notes belonging to the harmonic context of this chord, we can apply the following constraint:

$$max(notes\text{-}set) - min(notes\text{-}set) <= 12$$

where *notes-set* is the set of notes in the chord's harmonic context. To do so, the following Gecode constraints are used :

```
1 max(home, IntVarArray variables, IntVar max-val);
2 min(home, IntVarArray variables, IntVar min-val);
3 linear(home, IntArgs coefficients, IntVarArray min-max, rel_type, int constant);
4 rel(home, IntVar var, rel_type, IntVar var);
```

where home is the computation space, variables is the array of integer variables representing the pitch of the notes in the harmonic context of the chord, var is an integer variable, rel_type is a relation type, max-val and min-val are variables to store the maximum and minimum value of variables, coefficients is an array of coefficients for the linear equation and min-max is a variable array with the variables max-val and min-val. Figure 3.10 show the Lisp code for this constraint.

3.2.6 Range restriction constraint

This constraint ensures that the melody is always above the lowest note of the chords and below the highest note of the chord plus two octaves. This corresponds to the following mathematical formula:

$$\forall i \in [0, n]\ min(chords) <= x[i] <= max(chords) + 24$$

where x is the array of variables representing the pitch and chords is the array containing all the notes from the chords. The Gecode constraint used is

```
1     rel(home, IntVar variable, rel-type, int number);
```

where home is the computation space, variable is an integer variable, rel-type is a relation type and number is an integer. Figure 3.11 shows the code for this constraint.

3.2.7 Melody direction constraints

This section will cover a group of constraints that have similar purpose, that is giving a global direction to the melody.

Strictly increasing/decreasing pitch and increasing/decreasing pitch constraint

These constraints ensure that the notes specified as an argument have strictly increasing/ strictly decreasing values or increasing/decreasing values. That is, their pitch is going (strictly) up/down. The following constraints are applied for strictly increasing pitch constraint, strictly decreasing pitch constraint, increasing pitch constraint and decreasing pitch constraint respectively :

$$x_0 < x1 < ... < x_{|x|-1}$$

$$x_0 > x1 > ... > x_{|x|-1}$$

$$x_0 \leq x1 \leq ... \leq x_{|x|-1}$$

$$x_0 \geq x1 \geq ... \geq x_{|x|-1}$$

The Gecode constraint used for these is

```
1 rel(home, IntVar variables, (IntRelType) rel_type);
```

where home is the computation space, variables is the array of variables representing the pitch of the notes, and rel_type is the mathematical relation. The code for these constraints can be found in figure 3.12.

```
; <sp> is the space,<notes> is the variable array on which the constraint is executed
; <input-rhythm> is the voice object of the rhythm for the melody we want to find
; <chords> is the voice object of the chords
(defun harmonic-interval-chord (sp notes input-rhythm chords)
    ; get the starting time of each of the notes of the melody
    (let ((melody-starting-times (voice-onsets input-rhythm))
        ; get the starting time of each of the chords
        (chords-starting-times (voice-onsets chords))
        (chord-counter 0); counter to know which chord we are currently looking at
        (variable-counter 0); counter to keep track of the note
        ; counter to keep track of which variable is the start of the melody over the given
    chord
        (local-starting-note-melody 0)
        (starting-time-next-chord 0))
        (dolist (c chords-starting-times); go through the chords starting times
            (setf starting-time-next-chord (nth (+ 1 chord-counter)
                chords-starting-times)); get the starting time of the next chord
            ; keep a track of the first note in the context of this chord
            (setf local-starting-note-melody variable-counter)
            (cond
                ((>= (+ 1 variable-counter) (length melody-starting-times)); last note
                        (return )))
            ; go through all the notes that we haven't seen yet
            (dolist (m (subseq melody-starting-times variable-counter))
                (cond
                    ; last note
                    ((= variable-counter (- (length melody-starting-times) 1))
                        (hic-constraint sp notes local-starting-note-melody
                            variable-counter))
                    ((typep starting-time-next-chord 'null); last chord
                        ; post the constraint on all remaining notes
                        (hic-constraint sp notes local-starting-note-melody
                            (- (length melody-starting-times) 1))
                        (return ))
                    ; if the note is in the context of this chord
                    ((< m starting-time-next-chord)
                        (incf variable-counter 1))
                    ; the note is not in the context of this chord
                    ((>= m starting-time-next-chord)
                        ; post the constraint on the corresponding notes
                        (if (>= (- variable-counter 1) local-starting-note-melody)
                            (hic-constraint sp notes local-starting-note-melody
                                (- variable-counter 1)))
                        (return ))))
            (setf chord-counter (+ 1 chord-counter)))))
; post the constraint that max(pitch[starting-note],..., pitch[ending-note]) - min(pitch[
    starting-note],..., pitch[ending-note]) <= 12 (an octave)
(defun hic-constraint (sp notes starting-note ending-note)
    (let (notes-array max-min-array)
        ; create an array to represent all the notes played with that chord
        (setf notes-array
            (gil::add-int-var-array sp (+ (- ending-note starting-note) 1) 0 127))
        ; create a table where the first value is the max and the second is the min
        (setf max-min-array (gil::add-int-var-array sp 2 0 127))
        ;for each element of note-array, make it equal to the note it represents
        (loop :for j :from 0 :below (+ (- ending-note starting-note) 1) :do
            ; notes-array[j] = notes[j + starting-notes]
            (gil::g-rel sp (nth j notes-array) gil::IRT_EQ
                (nth (+ starting-note j) notes))
        )
        (gil::g-lmax sp (first max-min-array) notes-array); max-val = max(notes-array)
        (gil::g-lmin sp (second max-min-array) notes-array); min-val = min(notes-array)
        ; max(notes-array) - min(notes-array) <= 12)
        (gil::g-linear sp '(1 -1) max-min-array gil::IRT_LQ 12)
    )
)
```

Figure 3.10: code for the melodic-interval-chord constraint

```
;<sp> is the space,<notes> is the array of variables representing the pitch
;<input-chords> is the input chords in the form of a voice object
(defun range-restriction (sp notes input-chords)
    ; get the list of pitches for all the chords in the voice object
    (let ((chords (om::chords input-chords))
        vals min-note max-note)
        ; collect all the notes and put them in one big list
        (setf vals
                (to-midi (om::flat
                    (loop :for n :from 0 :below (length chords) :by 1 collect (om::lmidic (
    nth n chords)))
                ))
        )
        (setf min-note (min-list vals)); get the lowest note
        (setf max-note (max-list vals)); get the highest note

        (dolist (note notes);restrain the interval domains to acceptable values
            (gil::g-rel sp note gil::IRT_GQ min-note) ; note >= min-note
            ; note <= max-note + 2 octaves (24 semitones)
            (gil::g-rel sp note gil::IRT_LQ (+ max-note 24))
        )
    ))

```

Figure 3.11: Code for the range-restriction constraint

```
; <sp> is the space,<notes> is a list of IntVars representing the pitch of the notes
(defun strictly-increasing-pitch (sp notes)
    (gil::g-rel sp notes gil::IRT_LE nil)
)

; <sp> is the space,<notes> is a list of IntVars representing the pitch of the notes
(defun increasing-pitch (sp notes)
    (gil::g-rel sp notes gil::IRT_LQ nil)
)

; <sp> is the space,<notes> is a list of IntVars representing the pitch of the notes
(defun strictly-decreasing-pitch (sp notes)
    (gil::g-rel sp notes gil::IRT_GR nil)
)

; <sp> is the space,<notes> is a list of IntVars representing the pitch of the notes
(defun decreasing-pitch (sp notes)
    (gil::g-rel sp notes gil::IRT_GQ nil)
)

```

Figure 3.12: Code for the melody direction constraints

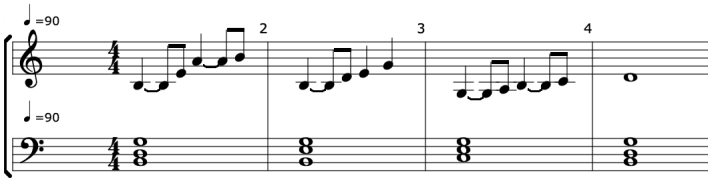


Figure 3.13: Example of the use of the mostly increasing pitch constraint

Mostly increasing/decreasing pitch constraint

These constraints give a general direction for the pitch (up/down) while still allowing some intervals to go in the opposite direction. They can be used on short melodies to make them go up or down, or they can be used on larger melodies to give them a general direction (see figure 3.13). It works by enforcing that the overall interval of the melody is at least what is given as an argument, and by enforcing that at least eighty percent of the intervals are positive/negative depending on the constraint. It can be expressed mathematically as follows :

$$\sum_{i=0}^{n} intervals[i] = global_interval$$

$$\#\{i \in [0,n] \,|intervals[i] > 0\} \geq 0.8 \times n$$

$$\#\{i \in [0,n] \,|intervals[i] < 0\} \geq 0.8 \times n$$

For the mostly increasing pitch and the mostly decreasing pitch constraints respectively. The Gecode constraints used are :

```
1     rel(home, IntVar sum, rel-type, expr(home, sum(IntVarArray vars)));
2     rel(home, IntVar sum, (IntRelType) rel_type, int global-interval);
3     count(home, IntVarArray vars, IntSet set, rel-type, int val);
```

where home is the computation space, sum is the variable that takes the value of the sum of all the elements of vars, global-interval is the global interval of the melody as specified, vars is the array of intervals, set is the set of values representing positive/negative intervals, and val is the number of variables in vars that must take a value within set. Figure 3.14 shows the code for those two constraints.

3.2.8 All different notes constraint

This constraint ensures that all the notes of the melody are different. That is,

$$\forall i, j \in [0, |x|] \; st \; i \neq j \;\; x[i] \neq x[j]$$

This is enforced using the following Gecode constraint:

```
1 distinct(home, IntVarArray x);
```

where home is the computation space and x is the array of variables. Figure 3.15 shows the code for this constraint.

3.2.9 At least n constraint

This constraint ensures that given an array of variables and an array of integers, at least m of those variables will take the value in the integer array corresponding to their index in the array. That is,

$$\#\{i \in [0,n] \mid x[i] = c[i]\} \geq m$$

where x is the array of variables, c is the array of integers and n is the length of these arrays. The Gecode constraint used is

```
1 count(home, IntVarArray vars, IntArgs c,rel_type, int val);
```

where home is the computation space, vars is the array of variables, c is the array of integers, rel-type is a relation type and val is an integer. Figure 3.16 shows the code for this constraint.

3.3 Branching

This section will discuss the different strategies for variable and value selection that are used for the different CSPs. The variable and value selection strategies differ for both CSPs, so they will be discussed separately. Branching is performed on both the pitch variable array and the intervals variable array since constraints are relatively spread out on the two arrays to increase performance.

```
; <sp> is the space,<notes> is a list of IntVars representing the pitch of the notes
; <intervals> is a list if IntVars representing the intervals between consecutive notes
; <global-interval> is the global interval that the melody spans
(defun mostly-increasing-pitch (sp notes intervals global-interval)
    (let (sum interval-domain-greater-than-zero)
        ;  variable to hold the result of the sum of all intervals (cant' be bigger than
    127)
        (setq sum (gil::add-int-var sp -127 127))
        (setq interval-domain-greater-than-zero
            (loop :for n :from 1 :below 12 :by 1 collect n)); [1..12]

        (gil::g-sum sp sum intervals); sum = sum(intervals)
        ; sum >= global-interval
        (gil::g-rel sp sum gil::IRT_GQ (parse-integer global-interval))
        ; (nb of intervals > 0) >= 0.8*size(intervals)
        (gil::g-count sp intervals interval-domain-greater-than-zero gil::IRT_GQ
            (ceiling (* 80 (length intervals)) 100))
    )
)

; <sp> is the space,<notes> is a list of IntVars representing the pitch of the notes
; <intervals> is a list if IntVars representing the intervals between consecutive notes
; <global-interval> is the global interval that the melody spans
(defun mostly-decreasing-pitch (sp notes intervals global-interval)
    (let (sum interval-domain-smaller-than-zero)
        ; variable to hold the result of the sum of the intervals
        (setq sum (gil::add-int-var sp (- (* 127 (length intervals))) -1))
        (setq interval-domain-smaller-than-zero
            (loop :for n :from 1 :to 13 :by 1 collect (- n))); [-12...-1]

        (gil::g-sum sp sum intervals); sum = sum(intervals)
        ; sum <= global-interval
        (gil::g-rel sp sum gil::IRT_LQ (- (parse-integer global-interval)))
        ; (nb of intervals < 0) >= 0.8*size(intervals)
        (gil::g-count sp intervals interval-domain-smaller-than-zero gil::IRT_GQ
            (ceiling (* 80 (length intervals)) 100))
    )
)
```

Figure 3.14: Code for the mostly increasing pitch and mostly decreasing pitch constraints

```
; <sp> is the space,<notes> is a list of IntVars representing the pitch of the notes
(defun all-different-notes (sp notes)
    (gil::g-distinct sp notes)
)
```

Figure 3.15: Code for the all different notes constraint

```
;<sp> is the space,<notes> is the array of variables representing the pitch
;<values> is the array of values
;<n> is the number of variables such that notes[i] = values[i]
(defun at-least-n (sp notes values n)
    (gil::g-count-array sp notes values gil::IRT_EQ n)
)
```

Figure 3.16: Code for the at least n constraint

3.3.1 Random number generators

The results need to be reproducible for many reasons, including if a composer deletes a solution by accident and they want to find it again. This poses a problem with the random heuristics that are explained below. The way to ensure that results are reproducible while still being able to use a random approach is to use random number generators. They are Gecode objects that take an unsigned integer seed, and using the same seed will generate the same numbers, hence the same "randomness". Currently, the seed cannot be changed and the random number generation happens in GiL.

3.3.2 Variable selection heuristic

Melody-finder CSP The branching for pitch and intervals is based on a first fail approach. The variable selection heuristic for the pitch variable array will be based on the degree of the variables, since some variables are involved in more constraints than others (typically the notes played on top of a chord). For the intervals variable array however, all variables have the same degree and using the same heuristic would not be of any use. The chosen heuristic will then be based on the domain of the variables, since some intervals will be more restrained due to the domain of the corresponding pitch variables.
The variable selection heuristic for the pitch variable array is to choose the variable with the biggest degree first. In the event of a tie, the first variable with that degree will be selected first. This heuristic is expressed in Gecode as

```
1 INT_VAR_DEGREE_MAX();
```

For the intervals variable array, the variable selection heuristic will be to choose the variable with the smallest current domain. This is expressed in Gecode as

```
1 INT_VAR_SIZE_MIN();
```

Variator CSP Since the goal of this CSP is to have variations of an existing solution, variables will be selected at random to ensure that all notes are equally likely to be modified. This is not optimal in terms of time complexity, but solutions are usually still found within a second. This heuristic is expressed in Gecode as

```
1 INT_VAR_RND(r); // r is a random number generator
```

3.3.3 Value selection heuristic

In order to improve the efficiency of the solver, the value selection heuristics differ for the two CSPs but also depending on the optional constraints that are selected.

Melody-finder CSP The value selection heuristic for the pitch variable array is to choose randomly the next value to assign, to guarantee a certain variety amongst the different notes. The random value is chosen with a seed, so running the CSP twice with the same input will produce solutions in the same order. When it comes to the value selection heuristic for the intervals variable array, there are different cases :

- Default behaviour : if no optional constraints are selected, the value selection heuristic is to select a random value to increase the diversity of the solutions.
- the increasing or strictly increasing constraints are selected: In this case, the intervals are always positive and choosing smaller intervals first is more likely to produce a solution than choosing big intervals, considering that the pitch is limited in height. The smallest possible value will be chosen first.
- the decreasing or strictly decreasing constraints are selected: Similarly, all intervals will be negative and therefore choosing the biggest intervals first (i.e the ones that are closer to 0) is more likely to find a solution quickly.
- the mostly increasing constraint is selected: in this case, most intervals will be positive even though some will might be negative. The selected strategy is to split the domain in half according to the mean of the minimum and maximum values of the domain, and to select the upper half.

- the mostly decreasing constraint is selected: similarly, most intervals will be negative and the selected strategy is to branch on the lower half of the domain first.

The corresponding Gecode strategies are listed below.

```
1 INT_VAL_RND(r); // default behaviour
2 INT_VAL_MIN(); // for the increasing or strictly increasing constraints
3 INT_VAL_MAX(): // for the decreasing or strictly decreasing constraints
4 // select values greater than the minimum of the smallest and largest values
5 INT_VAL_SPLIT_MAX(); // for the mostly increasing constraint
6 // select values smaller than the minimum of the smallest and largest values
7 INT_VAL_SPLIT_MIN(); // for the mostly decreasing constraint
```

These value selection heuristics, coupled to the variable heuristics described above, ensure that a solution is often found within a couple of seconds, though it is usually much faster than this. It can happen however that the solver doesn't find a solution in an appropriate amount of time in the case of the use of the mostly increasing/decreasing constraints with a large number of variables.

Variator CSP Since the purpose of this CSP is to provide variations of a given solution, it makes sense that the branching heuristic for value selection is random. Again, this is not ideal in terms of time complexity, but results are still found relatively quickly.

Chapter 4

Searching for a solution

Now that we have a model, the next step is to find solutions to the problem. In our case, a solution is a list of values for the pitch array that are in the domain of the variables and that satisfy all the constraints stated in the model. There are a lot of candidate solutions, typically $\prod_{i=0}^{n} |pitch[i]|$, and going through each of them one by one without a strategy would not be efficient. The branching heuristics described in the model will help guide the search towards a solution, but the search can still be improved and this section covers how we can do that. This chapter will first discuss how we will search for solutions, and will then discuss how to make the search faster and the solutions more diverse. It will then talk about how and why we should separate the search from the tool.

4.1 How to search?

4.1.1 Searching algorithm

The first question that comes to mind is what algorithm will we use for the search? Gecode offers three possibilities :

- Depth-first search (DFS) : The solver assigns a value to a variable according to the branching heuristics, and keeps doing that until it finds a solution or it finds that there is no solution possible with the values already assigned. In that case, it goes back to the last assigned value and changes it, and repeats the same steps.
- Branch-and-bound search (BAB) : When a solution is found, an additional constraint is posted on the future solutions based on the last found solution. This is typically used to find a better solution than the previous one.
- Limited-discrepancy search (LDS) : the discrepancy is the number of times that a choice has been made that was not the choice suggested by the search heuristics. Limited discrepancy search is searching for solutions by starting with a discrepancy of zero and incrementing it by 1 everytime it has searched every path with the previous discrepancy.

Using depth-first search, consecutive solutions will be very similar because of the nature of the search, regardless of the variable/value heuristics, even if some heuristics might have slightly more variability than others. This isn't ideal as the tool must offer a variety of solutions to the composer to increase their creativity and give them different ideas, and there are typically a lot of solutions that only differ by one or two notes which would make the tool less efficient and unpleasant to use. However, depth-first search guarantees that all the solutions will be presented to the user if they take the time to go through all of them.
The same problem appears with limited discrepancy search, even though the variety is slightly higher after some time, and it also guarantees that all the solutions will be presented to the user.
With branch-and-bound search, the variety can be better depending on the constraint posted after each solution. However, it requires to post an additional constraint after each solution, or more than one if the solutions are not comparable (one solution is not better than the other) and posting said constraint will reduce the amount of solutions to the problem, meaning that some solutions to the original problem will disappear every time the branch-and-bound constraint is posted.

4.1.2 Improving the variety amongst the solutions

To increase the variety of consecutive solutions, three different methods will be examined : the first one consists of searching for all of the solutions and then shuffling them. For this approach, depth-first-search could be used since it will find all the solutions to the problem, and an efficient shuffling algorithm could be used. This option is great for consecutive solution variety but has two big drawbacks : space and time complexity. Indeed, the bigger the problem the more memory space it will need to store all the solutions, and the more time it will take to find and sort them all. However this guarantees that all the solutions are visible to the user if they take the time to go through all of them.
The second method is to use a branch-and-bound search-engine and to post a constraint after each solution. This guarantees a good variety in the solutions provided that the constraint is appropriate, and has the advantage of not needing a lot of memory to store all the solutions. The user can also request one solution at a time, which considerably reduces the time they have to wait for a first solution. However, the drawback is that this method discards solutions that might have been interesting to the user because they are too similar according to the constraint.
The third method we will examine is to use a large-neighbourhood-search approach with a depth-first-search search-engine. If used with depth-first-search, it can help making consecutive solutions more different but it doesn't guarantee that it will always be the case, and it requires a way to tell if one solution is "better" or not than another, which is complicated as the quality of a solution is subjective in this case. This method would result in a better time and space complexity than the previous depth-first-search approach, however the advantage of finding all of the solutions disappears.

4.1.3 Improving the speed of the search

To increase the speed of the search, multiple threads will be used to allow the search engine to use parallel search. This will allow for a more diverse exploration of the search tree, which increases the odds of finding a solution faster. It also slightly increases the variety of consecutive solutions, though the contribution is minor.

4.2 Imposing variety between solutions

The branch-and-bound approach will first be considered, as it doesn't require a way of ranking solutions from better to worse, and it can guarantee good diversity in consecutive solutions if the constraint is appropriate.

4.2.1 How to compare two solutions

For branch-and-bound search to work, it needs a constraint that can ensure that two solutions are different enough. We can either post a constraint on solutions as a whole, which has a time complexity of $\mathcal{O}(1)$, or we can post a constraint per variable which has a time complexity of $\mathcal{O}(n)$ where n is the number of variables, independently of the constraint that is being applied. The method of comparing solutions as a whole will be preferred since it has better time complexity and it is more practical.

4.2.2 The constraint

We must now come up with a way of imposing that two solutions are different enough according to a criteria, which must also be defined. The similitude between two solutions will be defined as the number of variables that take the same value in both solutions. An upper bound can be set, that is proportional to the total number of variables, and a constraint can then ensure that the number of variables that take the same value in both solutions is lower than that upper bound.

4.2.3 Implementation

We will define a constraint called *solutionVariety* that will take as arguments the array of variables, an integer array with the values of the variables for the previous solution, and an integer representing the maximum number of variables that may take the same value as in the previous solution. The code for said constraint can be found in figure 4.1.

```
void solutionVariety(vector<int> prev_sol, int min_variety){
    // create a variable with domain between 0 and nVariables-1
    IntVar z(*this, 0, n-1);
    // cast from vector to IntArgs
    IntArgs p_sol(prev_sol);
    // count the number of elements such that pitch[i] = prev_sol[i]
    // and put it in z (pitch is the array of variables,
    //it is an attribute of the class of the problem)
    count(*this, pitch, p_sol, IRT_EQ, z);
    // constrain z to be smaller than or equal to the number
    //of elements that are allowed to be similar (n - variety)
    rel(*this, z, IRT_LQ, n-min_variety);
}
```

Figure 4.1: code for the solutionVariety constraint, that is applied every time a new solution is found

The constraint does the following: first, it creates a variable $z \in \{0, ..., n_{variables} - 1\}$. Then, it enforces the following :

$$z = \#\{i \in \{0, ..., |x| - 1\} | x_i = c_i\}$$

$$z <= n_{variables} - v$$

where x is the array of variables, c is the array of integer values for the previous solution and v is the minimum number of variables that must take a different value.

4.2.4 Limitation

As of now, this feature is not available in the Melodizer tool. This is due to a problem using branch-and-bound search in GiL. The branch-and-bound solver only seems to work when the function defining the constraints to post when a solution is found does nothing. This might be related to how variables are handled in GiL. The search engine used for Melodizer is a depth-first search engine.

4.3 Separating the search from the rest of the tool

4.3.1 Why we should separate the search from the rest

The search aspect of the problem shouldn't be a blocking event, meaning that it shouldn't block the interface when the solver is searching for a solution. Additionally, the user should be able to stop the search if it is taking too long or if they want to perform an action, like changing the input and starting over for example. To solve both of these issues, the function to find the next solution will be called in its own separate thread, so that it can work in the background without affecting the rest of the tool. The tool can then periodically check if the user wants to stop the search, and react accordingly.

4.3.2 How it works in practice

Thread creation The thread creation is done in Lisp using the *mp::process-run-function* function. The function to call in a separate thread is passed as an argument to this function, and the function returns immediately an object of type *mp::process*, allowing to get access to the thread (i.e see its state,...). Figure 4.2 shows the code creating a thread and calling the function to find the next solution inside of the thread. Since the solution field of the melodizer object is only modified there, there is no need to use a lock.

Stopping the search In Gecode, the search can be stopped after a given time with the use of TimeStop objects. They can be passed to the search engine through Options object, and the search-engine will then check if the time limit given by the TimeStop object is exceeded or not. If it is, it will stop the search and return a null solution. The *next* function of the tool can then check if the user wishes to stop the search through a boolean set to true by the pressing of a button, and react accordingly: if the user didn't press the button, it will simply reset the timer and resume the search, but if the user wants to stop the search it will throw an error stating that the search has been stopped. The search can then be resumed by the pressing of

```
;get the next solution
(mp:process-run-function ; start a new thread for the execution of the next method
    "next thread" ; name of the thread, not necessary but useful for debugging
    nil ; process initialization keywords, not needed here
    (lambda () ; function to call
        (setf (solution (om::object editor)) (search-next (result (om::object editor))
            (om::tree (input-rhythm (om::object editor))) (om::object editor)))
        (setf (om::tempo (solution (om::object editor)))
            (om::tempo (input-rhythm (om::object editor)))); tempo = tempo(input)
        (om::openeditorframe ; open a voice window displaying the solution
            (om::omNG-make-new-instance
                (solution (om::object editor)); the new solution
                "current solution" ; name of the window
            )
        )
    )
    ; arguments for the function call if necessary
)
```

Figure 4.2: Code creating a thread and calling the function the get the next solution of the problem

```
Simple_problem* m = new Simple_problem();// create an instance of a problem
Gecode::Search::Options opts; // create an options object
opts.threads = 0;// use as many threads as possible
Gecode::Search::TimeStop maxTime(1000); // create a time stop object, maxtime = 1000ms
opts.stop = &maxTime;
BAB<Simple_problem> e(m, opts); // create the search engines with the defined options

```

Figure 4.3: Example of setting a time limit to the search in Gecode

the next button. Figure 4.3 shows the code creating a BAB search engine and setting its number of threads and a time limit for the search in Gecode.

4.4 Example of a full musical CSP

Now that we have a model of the problem and a strategy to search for solutions, we can create a complete musical CSP in Lisp using GiL and find solutions for it. Figure 4.4 shows the code for a simplified version of the melody-finder CSP of Melodizer, and figure 4.5 shows the code for the function that is searching for a solution. The constraints used are the following :

- $\forall i \in [0, n[\ intervals[i] = pitch[i+1] - pitch[i]$
- $\forall i \in [0, n]\ min(chords) <= x[i] <= max(chords) + 24$
- $dom(intervals) \in \{-1, -4, -5, -7, -8, -9, -12, 0, 1, 2, 3, 4, 5, 7, 8, 12\}$
- $dom(X) = N_{tonality}$

And the different steps are highlighted in blue.

```
; <input> is a voice object with the chords on top of which the melody will be played
; <rhythm> the rhythm of the melody to be found in the form of a voice object
; <optional-constraints> is a list of optional constraint names
; <global interval> is the global interval that the melody should cover
; <key> is the key in which the melody is,<mode> is the mode of the tonality
(defmethod melody-finder (input rhythm optional-constraints &optional
    (global-interval nil) (key 60.0) (mode "major"))
    (let ((sp (gil::new-space)); create the space;
        pitch intervals dfs tstop sopts)

        ;initialize the variables
        (setq pitch (gil::add-int-var-array sp (om::n-pulses rhythm) 0 127))
        (setq intervals (gil::add-int-var-array sp (- (length pitch) 1) -127 127))

        ; connect pitches to intervals
        (loop :for j :from 0 :below (length intervals) :do ;for each interval
            (let (temp)
                (setq temp (gil::add-int-var-array sp 3 -12 108))
                ; temp[0] = pitch[j]
                (gil::g-rel sp (first temp) gil::IRT_EQ (nth j pitch))
                ; temp[1] = pitch[j+1]
                (gil::g-rel sp (second temp) gil::IRT_EQ (nth (+ j 1) pitch))
                ; temp[2] = intervals[j]
                (gil::g-rel sp (third temp) gil::IRT_EQ (nth j intervals))
                ; -pitch[j] + pitch[j+1] - intervals[j] = 0
                (gil::g-linear sp '(-1 1 -1) temp gil::IRT_EQ 0)
            )
        )

        ; post the constraints
        (range-restriction sp pitch input)
        (interval-between-adjacent-notes sp pitch intervals)
        (in-tonality sp pitch key mode)

        ; branching
        (gil::g-branch sp pitch gil::INT_VAR_DEGREE_MAX gil::INT_VAL_RND)
        (gil::g-branch sp intervals gil::INT_VAR_SIZE_MIN gil::INT_VAL_RND)

        ;time stop
        (setq tstop (gil::t-stop)); create the time stop object
        (gil::time-stop-init tstop 500); initialize it (time is expressed in ms)
        ;create search options
        (setq sopts (gil::search-opts)); create the search options object
        (gil::init-search-opts sopts); initialize it
        ; set the number of threads to be used during the search
        (gil::set-n-threads sopts 1)
        (gil::set-time-stop sopts tstop); set the timestop object to stop the search
        ; create the search engine
        (setq se (gil::search-engine sp (gil::opts sopts) gil::DFS))
        ; return the search engine, varibale arrays and search options
        (list se pitch tstop sopts intervals)
    )
)

```

Figure 4.4: Code for a simplified version of the melody-finder CSP

```
; <l> is a the output of a CSP
; <rhythm> is the input rhythm as given by the user
; <melodizer-object> is a melodizer object
; this function finds the next solution of the CSP using the search engine given as an
    argument
(defmethod search-next (l rhythm melodizer-object)
    ; get the elements of the list
    (let ((se (first l))
          (pitch* (second l))
          (tstop (third l))
          (sopts (fourth l))
          (intervals (fifth l))
          (check t); for the while loop
          sol pitches)

    ; search until interrupted or no solution
        (om::while check :do
            ;reset the timer before launching the search
            (gil::time-stop-reset tstop)
            (setq sol (gil::search-next se)); search the next solution
            (if (null sol)
                (stopped-or-ended (gil::stopped se) (stop-search melodizer-object)
                    tstop)
                (setf check nil); we have found a solution so break the loop
            )
        )
        ; store the values of the solution
        (setq pitches (to-midicent (gil::g-values sol pitch*)))

        ;return a voice object that is the solution we just found
        (make-instance 'voice
            :tree rhythm
            :chords pitches
            :tempo 60
        )
    )
)
; determines if the search has been stopped by the solver because there are no more
;solutions or if the user has stopped the search
(defun stopped-or-ended (stopped-se stop-user tstop)
    ; if the search has not been stopped by the TimeStop object, there are no more solutions
    (if (= stopped-se 0)
        (error "There are no more solutions.")
    )
    ;otherwise, check if the user wants to keep searching or not
    (if stop-user
        (error "The search has been stopped. Press next to continue the search.")
    )
)
```

Figure 4.5: Code for the search for a solution

Chapter 5

Design and implementation of Melodizer
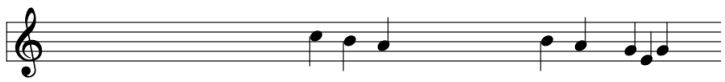
This chapter will cover the details of the design and implementation of the melodizer object in OpenMusic. It will discuss the representation of musical elements that Melodizer will work with, and will give a quick reminder on voice objects and how they can be edited. It will then talk about the design of the melodizer object, and discuss the editor window in detail. Finally, it will detail the implementation.

5.1 Representation of the musical elements
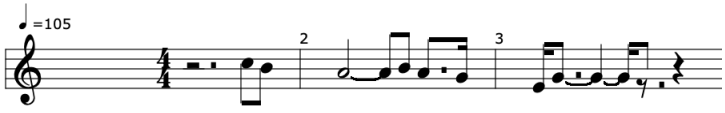
Two representations were considered for the musical elements: using Chord-Seq objects or Voice/Poly objects.

- Chord-Seq objects have a time-based approach to representing duration and rhythm. This makes it easy to access the starting time and duration of musical events, which is more practical for writing constraints on rhythm. However the visual representation is not easy to read. Notes are represented as quarter notes on a staff, and their duration is expressed as the distance between the note and the next note.
- Voice and Poly objects have a proportion-based approach, expressing duration as a fraction of a whole note and rhythm trees to represent rhythm. The visual representation is like the classical way of writing music, on a staff and with the shape of the note determining its duration.

Since the visual representation of Voice/Poly objects is easier for the composer to use, these objects will be used to represent musical elements in Melodizer even though it makes writing constraints involving rhythm more tedious to write. Figure 5.1 shows a simple melody represented using a Chord-Seq object (figure 5.1a) and a Voice object (figure 5.1b).



(a) Simple melody represented with a Chord-Seq object



(b) Simple melody represented with a Voice object

Figure 5.1: Comparison of the melody from "West Coast" by Imagine Dragons [7] represented with a Chord-Seq object and with a Voice object

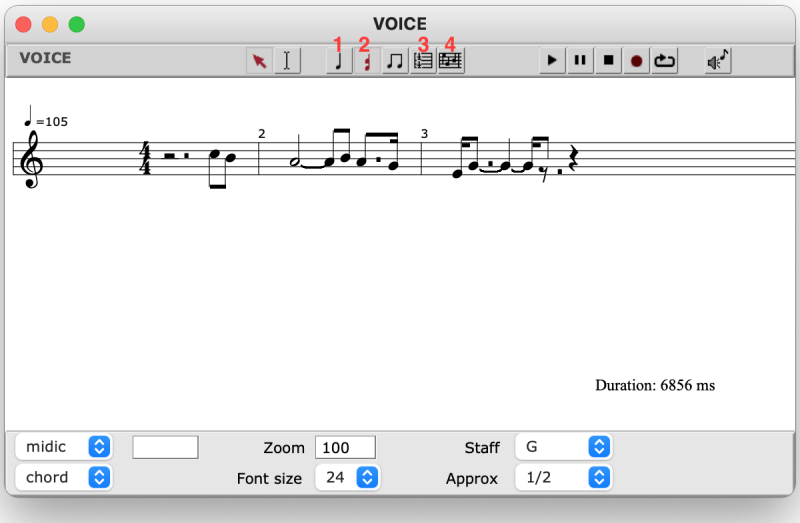


Figure 5.2: A voice object editor

5.2 Reminder on Voice objects

Since Voice objects are an important part of Melodizer, in particular the editor interface of these objects, a reminder is welcome in order to fully understand the implementation of the tool.
Voice objects represent a musical voice on a staff, with a tempo, a time signature, a rhythm and a pitch. The main attributes are tree, the rhythm-tree of of the voice, chords, the list of pitches for the notes, and tempo, the tempo of the voice. Each of these attributes can be set by connecting a data box to the appropriate inlet of the voice object, or they can be edited from the editor window of the object by adding notes, changing their duration,... The editor window of a Voice object is depicted in figure 5.2. Modifications to the voice can be done using the editor, the main ones are listed below :

- The pitch of a note can be adjusted by selecting it using the note tool (1) or the chord tool (2) and dragging it up/down or using the arrow keys of the keyboard.
- Measures : Measures can be added by clicking on the measure tool (3) and then clicking while pressing the “Cmd” key down where the new measure needs to be, or deleted by selecting it and pressing the “backspace” key.
- A rest can be transformed into a note and vice versa by selecting it using the chord tool and pressing the “esc” key.
- An event can be split into a group of smaller events of equal value by selecting it using the chord tool and entering a number representing the number of smaller events that need to replace this event.
- Notes can be added to a chord by selecting it using the note tool or the chord tool and pressing the “cmd” key.
- A voice or measure can be transposed by selecting it using the voice tool (4) or the measure tool respectively, and using the arrow keys.

Poly objects have a similar editor interface, the main difference being that there can be multiple voices. All the manipulations stated above are also valid for these objects.

5.3 Melodizer object design

The end tool is an OpenMusic object with an interface allowing to dynamically change input and parameters to model a variety of musical problems. Its attributes belong to one of three categories. Each of them will be discussed separately. The code for the tool can be found in appendix C. There is one inlet and one outlet for each attribute of the melodizer object, and their name appears when trying to connect something to them. Inlets allow to give input to the object, while outlets allow to take output from the object.

5.3.1 Input attributes

The first category of attributes is the input attributes. They can be set by the user, either through the editor window or by connecting data boxes to the inlets of the object, and allow to specify the problem precisely.

- **input-chords** is the voice object containing the chord sequence containing the chords that the melody will harmonize with.
- **input-rhythm** is the voice object containing the rhythm of the melody that the user wishes to find, or the voice object containing the melody that the user wants to vary.
- **key** is the key of the tonality that the user has chosen for the melody. It has a default value of C.
- **mode** is the mode of the tonality that the user has chosen for the melody. It has a default value of "major".
- **tool-mode** is the mode the tool is currently in. It has a default value of "melody-finder".
- **variety** is the solution variety desired by the user, expressed in musical events. It has a default value of 0.

5.3.2 Inner workings attributes

The second category of attributes is the attributes that allow to transfer data to different parts of the melodizer objects (e.g. from the interface to the CSP). They are not accessible to the user and are there for functionality.

- **optional-constraints** is a list of strings representing optional constraints that have been selected by the user and that should be added to the CSP for the search. It has a default value of an empty list.
- **global-interval** is the interval that the melody should cross if a pitch orientation constraint is selected. It is expressed in absolute value and has a default value of 1.
- **result** is a temporary holder for the return value of the CSP. It is a list containing the search-engine, the decision variable arrays, and the options objects.
- **stop-search** is a boolean to know if the user wishes to stop the search or not. It has a default value of nil and a value of t means that the user wants to stop the search.

5.3.3 Output attributes

The last category of attributes is the output attributes. They are the result of the search for a melody, and allow to use the results outside of the Melodizer tool in OpenMusic, for example to integrate in a musical piece. They can be accessed through the corresponding outlet of the melodizer object.

- **solution** holds the last proposed solution of the CSP.
- **solutions-list** is the list of all the solutions saved by the user for the current search.
- **motifs-list** holds the list of motifs saved by the user.
- **phrases-list** holds the list of phrases saved by the user.

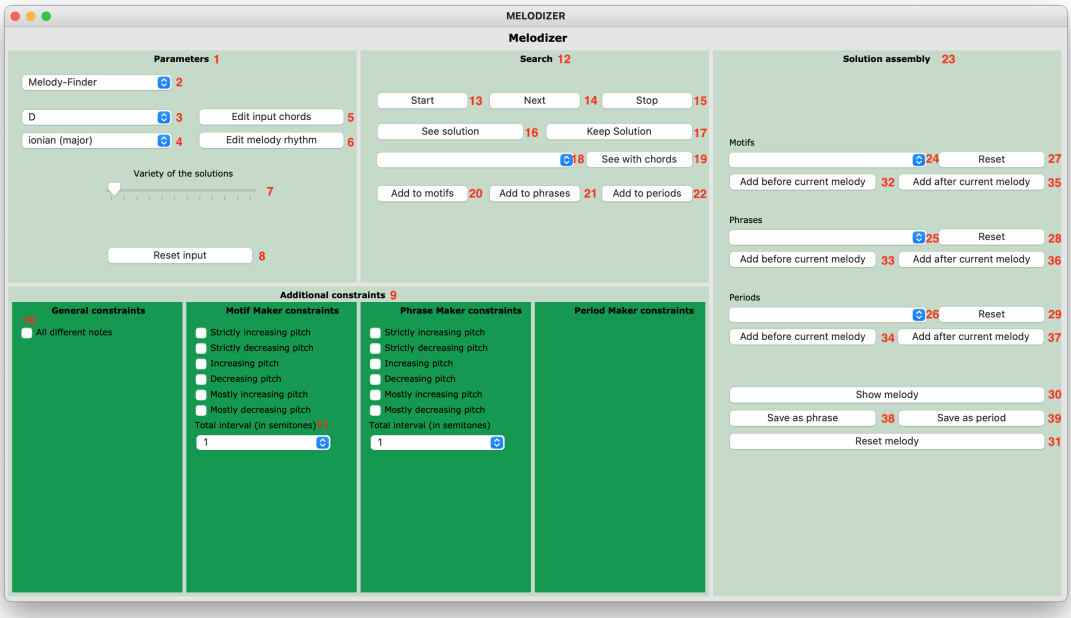


Figure 5.3: Editor window of the melodizer object

- **periods-list** holds the list of periods saved by the user.
- **output-solution** holds the selected solution for the current search in the pop-up menu.
- **output-motif** holds the selected motif for the current search.
- **output-phrase** holds the selected phrase for the current search.
- **output-period** holds the selected period for the current search.
- **melody** holds the melody constructed by the user using the solution-assembly panel.

5.4 Melodizer object editor

The editor window of the melodizer object is the most important part of the tool because it is the interface between the user and the CSP. It allows the user to interact with the problem by specifying optional constraints and with solutions to make them fit their needs. It is composed of four main components, that will be described individually. Figure 5.3 shows the interface of the melodizer object.

5.4.1 Input panel

The first panel that matters when using melodizer to generate melodies is the input panel (1) that allows the user to set the different parameters of the problem. This section covers the different elements that can be found in that panel.

Tool mode selection The first element we see is the tool mode selection pop-up menu (2). It allows the user to set the mode of the tool according to the use they want to make out of it. Currently, there are two modes : melody-finder, that allows the user to generate melodies based on input chords and a rhythm, or the variation-maker mode that allows to generate alternative melodies based on a given melody. Selecting a mode will modify the tool-mode attribute of the object to match the selection.

Tonality selection The two elements below that (3 and 4) allow the user to specify a tonality for the melody that they are looking for. Selecting a value will update the corresponding field of the melodizer object (key/mode).

Input edition The two elements to the right of the tonality selection (5 and 6) open the editor window for the input voice objects for the rhythm/melody and the chord sequence. Any modification is automatically saved since the object itself is edited.

Solution variety selection The slider in the middle of the panel (7) allows the user to set a proportion of musical events that must be different in every solution. A value of 0 means that two solutions can be identical. Selecting a value updates the variety field of the melodizer object. Currently, this feature is not fully supported, but the slider is there for future use.

Reset button The button at the bottom of the panel (8) allows to reset the voice objects for the rhythm/melody and the chords to start with a new blank voice object. Pressing the button will erase the input-chords and input-rhythm fields and replace their value with a new voice object.

5.4.2 Additional constraints panel

The panel at the bottom of the editor (9) is the additional constraints panel. It allows the user to specify additional constraints that they want to add to the problem. Depending on what they are looking for, there is a set of constraints available. Currently, there are no additional constraints for the variation-maker mode. Checking a box (10) adds the name of the corresponding constraint to the optional-constraints field of the melodizer object, and unchecking it removes it from the list. The mostly increasing/decreasing constraints (11) require to provide additional data, by selecting a minimum interval that the melody should cover. Selecting a value will update the global-interval field of the melodizer object. For now, constraints are not mutually exclusive meaning that it is possible to select different pitch orientation constraints even though they are not compatible (e.g strictly increasing and decreasing). Selecting several of them is likely to result in a problem with no solutions.

5.4.3 Search panel

The panel in the center of the top half of the editor window (12) is the search panel. It allows the user to control the search for a solution of the CSP with the parameters specified.

Start button The button on the left most side of the panel (13) is the start button. It allows the user to create the CSP by calling the melody-finder function or the variator function (see section 5.5.3), depending on the current mode of the tool. After pressing the button, any changes to the input panel or the additional constraints panel are not taken into account until the search is restarted by pressing the start button. Pressing the start button resets the solution and solutions-list attributes of the object as well as the stop-search field and updates the solution-display menu. It then stores the result of the function call in the result field of the object.

Next button The button to the right of the start button (14) allows the user to search for a solution. If the CSP has not been created, i.e the result field is nil, it raises an exception. Otherwise, it resets the stop-search field and creates a separate thread to call the search-next function (see section 5.5.3 as well). It then sets the solution field to the return value of the search-next function, and opens the editor window of the solution field to show the solution to the user. The user can modify the solution in the editor and any changes are saved.

Stop button The button to the right of the next button (15) allows the user to stop the search if it is taking too long. When it is pressed, the stop-search field of the melodizer object is set to t and the next time the search will be stopped by the options (max. 500ms after the pressing of the button), it will stop and raise an exception saying that the search has been stopped.

See solution button The button below the start button (16) allows the user to reopen the solution field editor window. It can still be modified and any changes are automatically saved. If there is no current solution, an exception is raised.

Keep solution button The button to the right of the see solution button (17) allows the user to save the current solution to the list of solutions for the current CSP. If there is no current solution, an exception is raised. Otherwise, the current solution is added to the solutions-list field of the melodizer object and the corresponding pop-up menu (18) is updated.

Solution list display The element under the see solution button (18) allows the user to see the list of solutions they saved. Selecting a solution will set the output-solution field of the object to the selected voice object, and open the editor window of the selected element, not the output-solution field. Any changes are saved.

See with chords button The button to the right of the solution list display (19) allows the user to see the solution together with the input chords. If there is no selected solution, i.e the output-solution field is nil, it raises an error. Otherwise, it creates a Poly object with the output-solution and the input-chords fields and opens its editor window. Any changes made at this stage are not saved to the output-solution window as it is a temporary object.

Add to motifs button The button on the bottom right of the panel (20) allows the user to save the selected solution (output-solution field) to the list of motifs (see Solution assembly panel). As always, if the output-solution field is nil, it raises an error. Otherwise, it creates a poly object with the output-solution and input-chords fields, and adds it to the motifs-list field. It then opens its editor window. Any changes are saved since the poly object here is not a temporary object.

Add to phrases/periods buttons These buttons (21 and 22) have the same behaviour as the add to motifs button, except that they add the solution to the phrases-list/periods-list fields respectively.

5.4.4 Solution assembly panel

The panel on the right of the window (23) allows the user to combine different motifs, phrases and periods that they have generated to write more complex pieces of music.

Motifs/Phrases/Periods list display The pop-up menus (24, 25 and 26) display the list of motifs/phrases/periods that have been saved by the user. Selecting one of them will set the output-motif/output-phrase/output-period field to the selected poly object, and open its editor window. Any changes are saved automatically.

Reset buttons The buttons to the right of the pop-up menus (27, 28 and 29) allow the user to erase the list of saved motifs/phrases/periods. Clicking on it will reset the output-motif/output-phrases/output-periods and motifs-list/phrases-list/periods-list fields of the object, and update the corresponding pop-up menu.

Show melody button The button below the list of motifs, phrases and periods (30) allows the user to see the melody that they are currently creating by assembling the different elements they created. If there is no melody, it throws an error. Otherwise, it opens the editor window of the melody field of the melodizer object. It can be edited like every other Poly object, and changes are automatically saved.

Reset melody button The button on the bottom of the panel (31) allows to erase the melody that the user was creating, e.g. when it has been saved to start a new one. When it is pressed, it sets the melody field to nil.

Add before/after current melody buttons The buttons below the pop-up menus showing the list of elements saved by the user (32 to 37) allow to add the selected element before/after the current melody, respectively. When they are pressed, if there is no melody (the melody field is nil) it throws an error. Otherwise, it calls the concat function from OpenMusic to append the melody and the element together.

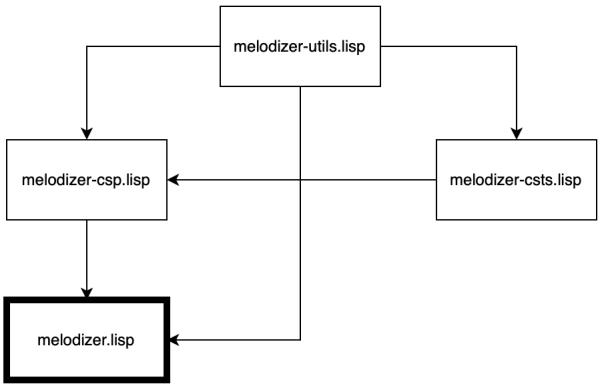


Figure 5.4: Relationship between the different files

Save as phrase/period buttons The last two elements that need to be discussed are the buttons underneath the show melody button (38 and 39). They allow to save the melody constructed by the user as a phrase or period respectively. When they are pressed, they check wether the melody is nil or not. If it is, it throws an exception, otherwise it adds the melody field to the phrases-list or the periods-list field respectively.

5.5 Melodizer object implementation

The code for Melodizer is spread into 4 different files to keep the main features separated. Figure 5.4 shows how these 4 files are interconnected. An arrow from 1 file to another means that functions from the first file are called from the code in the other file. The "melodizer.lisp" file contains the code for the melodizer object as well as the editor, the "melodizer-csp.lisp" file contains the code for the CSPs described in previous chapters, the file "melodizer-csts.lisp" contains the constraints used for the CSPs, and the "melodizer-utils.lisp" file contains generic utility functions.

5.5.1 Melodizer-utils

This file contains a variety of functions that have very different purposes. Here is a list of the main functions and their purpose:

- Conversion functions (from MIDI to MIDIcent and vice versa, from the name of a note to its value and vice-versa,...)
- A function to update a pop-up menu from the interface
- A function to get the starting time of all notes from a Voice object
- A function to get the series of tones and semitones defining a specific mode
- A function to get the notes that can be played on top of a given chord
- A function to get the mode and inversion of a chord based on its notes.

5.5.2 Melodizer-csts

This file contains all the constraint functions used in the CSPs described in chapter 3. Details will not be repeated here, but here is a list of the available constraints:

- All the notes of the melody are different
- Strictly increasing/decreasing pitch
- Increasing/decreasing pitch
- Mostly increasing/decreasing pitch

- At least n notes of the melody take the value they have in the input
- Notes are always higher than the lowest note from the chords and lower than the highest note of the chord + 2 octaves
- The intervals between consecutive notes follow a specific rule
- Notes are in the tonality specified
- The melodic interval in the context of a given chord is a maximum of an octave
- Notes played on top of a chord follow a set of rules

5.5.3 Melodizer-csp

This file contains the code for the two CSPs used in Melodizer, as well as the function that is searching for the solutions. The searching algorithm is the same for both CSPs, so there is only one searching function. The algorithm for the CSPs is explained in chapter 3 and the algorithm for the search is detailed in chapter 4.

5.5.4 Melodizer

This file contains the code for the object itself, as well as the code for the editor window and all of its components. The main components are:

- The declaration of the melodizer class and all the attributes of the object
- The creation of the main view
- The creation of the input panel
- The creation of the search panel
- The creation of the optional constraints panel
- The creation of the solution assembly panel

The code for this file is too long to show here, but can be found in appendix C.

Chapter 6

How to use Melodizer

This chapter will explain exactly how to load and use Melodizer inside OpenMusic. It assumes you already have installed OpenMusic and Gecode and have downloaded the libraries. Appendix A gives a step by step guide on how to install everything you need. This chapter will first explain how to create a patch and load the libraries. It will then explain how to give input to the tool and how to specify aditional constraints to use. It will also cover how to search for solutions, how to modify them and how to save them. Finally, it will explain how to combine generated solutions to create musical pieces.

6.1 Setting up the environment

6.1.1 Create a patch

The first step to be able to use Melodizer is to create an OpenMusic project. To do so, the user should open OpenMusic and select "create a new workspace", give it a name and press enter. This creates an empty workspace. Then click on file/new patch, and your workspace should look like figure 6.1. Double clicking on the patch will open it.

6.1.2 Load Melodizer and GiL

Now that you have created a patch, the next step is to load the Melodizer[1] library. Since it requires GiL[2] to work, it needs to be loaded too. To do so, click on windows/library. This will open the library window (figure 6.2a). Then click on file/add remote user library and navigate to the Melodizer and GiL folder on your desktop. The library window should now look like figure 6.2b. The libraries can be loaded by right-clicking

[1] https://github.com/sprockeelsd/Melodizer
[2] https://github.com/sprockeelsd/GiLv2.0

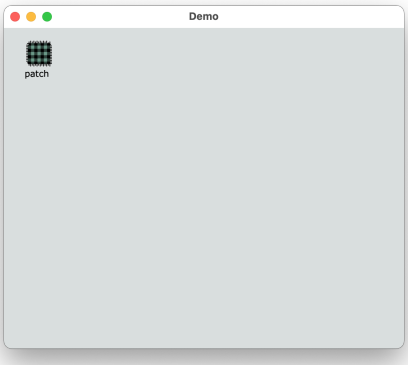
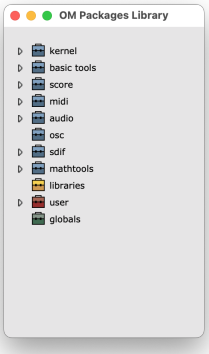
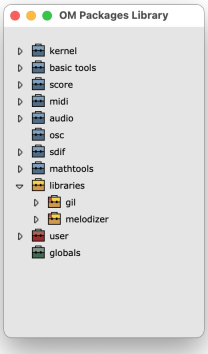


Figure 6.1: Workspace after adding a patch

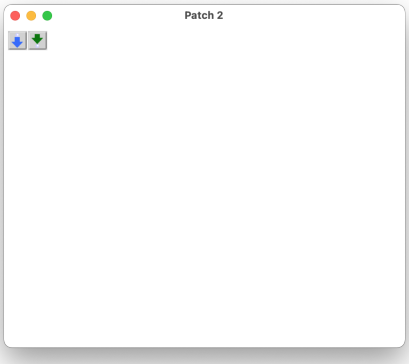


(a) Library window in OpenMusic

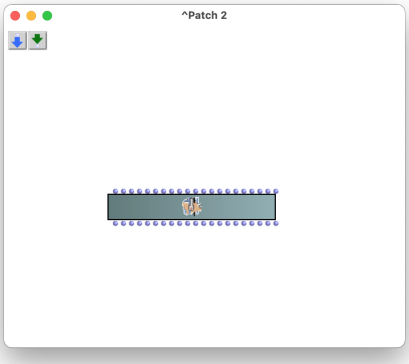


(b) Library window in OpenMusic when the libraries have been fetch

Figure 6.2: Library window before and after fetching the libraries



(a) Patch before creating a melodizer object



(b) Patch after creating a melodizer object

Figure 6.3: Patch before and after creating a melodizer object

the melodizer library and selecting “load library”. The boxes should turn blue, meaning that the libraries are loaded and ready to use.

6.1.3 Creating a melodizer object

Now that the libraries are loaded, they can be used in the patch that was just created. In the patch, select Classes/Libraries/Melodizer/ALL/MELODIZER, then click anywhere in the patch and a melodizer object will appear. Alternatively, you can double click in the patch and write “mldz::melodizer” to achieve the same result. Figure 6.3 shows a patch before and after creating a melodizer object. Once the object is created, the editor can be opened by double clicking on it. The melodizer object has 21 inlets and outlets, but only a few should be used for input and output. The inlets that can be used to give input to Melodizer are stated in section 5.3.1, and the ones that can be used to get output from Melodizer are stated in section 5.3.3.

6.2 Motif, phrase, period

In the following sections, motifs, phrases and periods will be mentioned multiple times so we feel it is important to give a proper definition of those terms [10].

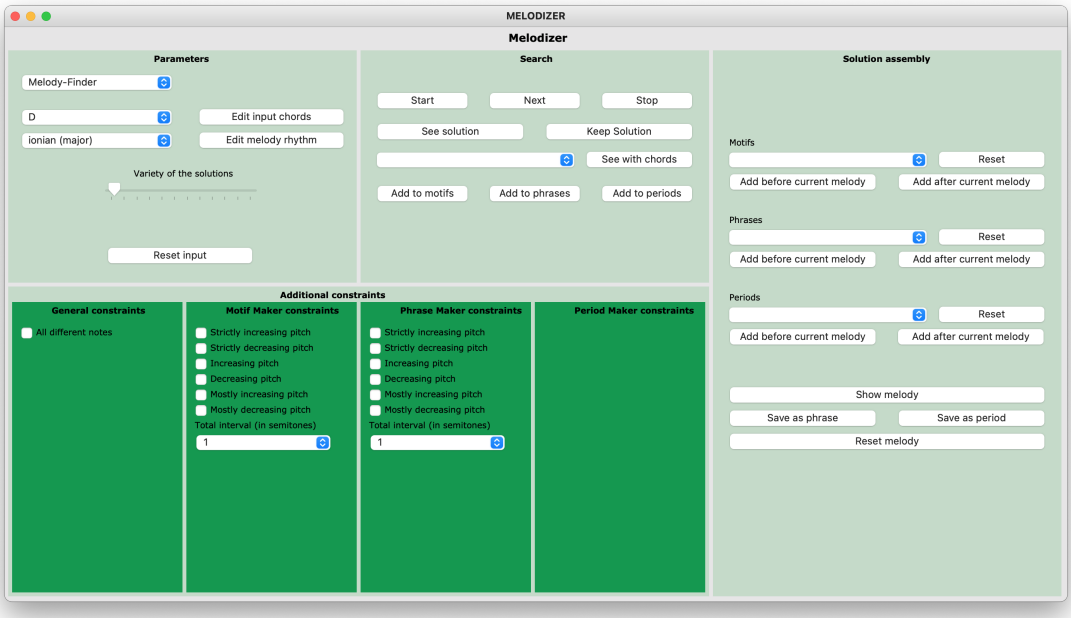


Figure 6.4: Editor window of the melodizer object

Motif A motif is a short melodic idea. It can be as simple as two notes, and can be more complex but is generally really short (up to one measure long).

Phrase A phrase is a little bit longer than a motif. It is generally a few bars long, and expresses a more complex melodic idea.

Period A period is a complete melodic idea. It can be composed of several phrases.

6.3 Giving input to the object

Now that you have created a melodizer object, the next step is to give it parameters to customize the problem to fit your needs. The input parameters are described in section 5.3.1. There are two ways to specify those parameters : through the editor window of the melodizer object or by connecting boxes to the corresponding inlets (the circles on top of the object) of the melodizer object.

6.3.1 Giving input from the editor window

This is the recommended way to set the parameters of your problem. To open the editor window, double click on the melodizer object. The window in figure 6.4 should appear. The editor window has four panels : the input panel on the top left, the search panel in the middle, the optional constraints panel at the bottom and the solution assembly panel on the right. For this section, we are interested in the input panel. The input panel is represented in figure 6.5. All the input parameters can be specified there. Here is how to enter parameters to the interface :

- To select the **mode of the tool**, the pop-up menu on the top left can be used. It currently has two modes : melody-finder, that allows you to find pitch for a melody that can be played on top of the input chords and that has the rhythm specified as input, and variation-maker, that allows you to generate variations of a melody that is given as an argument and that can be played on top of the chords given as an argument.
- To specify the **tonality** desired for the melody, there are two pop-up menus that allow you to select a key and a mode.

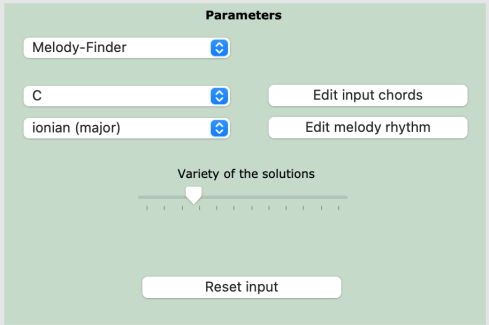


Figure 6.5: Input panel of the melodizer object editor window

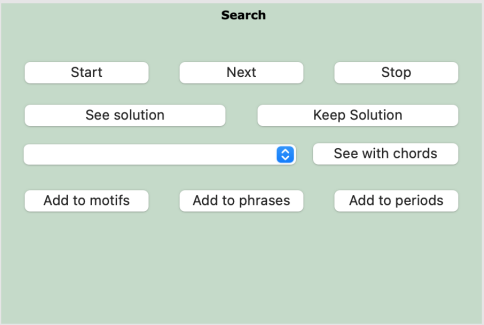


Figure 6.6: Search panel of the melodizer object editor window

- To enter a **chord progression** that serves as a base for the melody, you can press the *Edit input chords* button. This will open the editor window of a voice object and allow you to edit it like it is explained in section 5.2.
- To enter a **melody** or a **rhythm for a melody**, you can press the *Edit melody rhythm* button. That will open the editor window of a voice object and allow you to edit it.
- To **reset the input chords and melody** that you entered, you can press the *Reset input* button.

6.3.2 From the patch

Alternatively, you can give arguments to the object by creating data boxes and connecting them to the corresponding inlet of the melodizer object (the input-chords inlet for the input chords, the key inlet for the key, and so on). The name of an inlet will appear when trying to connect something to it, or can be found in the documentation of the melodizer object. Pay attention that giving inappropriate values to parameters can lead to errors (e.g. trying to set the input chords to a string). Additionally, connecting data boxes to the inlets of the melodizer object means that it must be evaluated to take these changes into account, and will therefore reset the object and everything that was saved in it. If you are more comfortable with creating voice objects from a rhythm-tree and a list of pitches, it is advised to create a voice object separately and then copy the staff and paste it inside the editor window after clicking on the *edit input chords* button.

6.4 Specifying the optional constraints

Now that the parameters to the problem are set, it is time to specify additional constraints if you want to add some to your problem. The optional constraints panel is displayed in figure 6.7. If the mode of the tool is melody-finder, depending on what you are looking for, there is a set of constraints that can be added to the problem. To do so, just check the corresponding box. Some constraints may require additional information (e.g. the mostly increasing/decreasing constraints). Be aware that currently, there is nothing preventing you from selecting incompatible constraints. Doing so will result in hazardous behaviour. Similarly, selecting a constraint that is not compatible with the mode of the tool will have hazardous behaviour.

6.5 Searching for a solution

Now that the parameters and the additional constraints have been set, you can start searching for solutions. Figure 6.6 shows the search panel of the editor window. The first thing to do is to press the *Start* button. At this point, the parameters and the optional constraints are locked for the search for a solution, and any changes will not be taken into account until the *Start* button is pressed again. The problem is now created, and you can **start searching for solutions** by clicking the *next button*. Depending on the problem and the constraints selected, the solver might take some time to find a solution. If you wish to **stop the search**, you can press the *Stop* button. A window will appear to inform you that the search has been stopped. When the

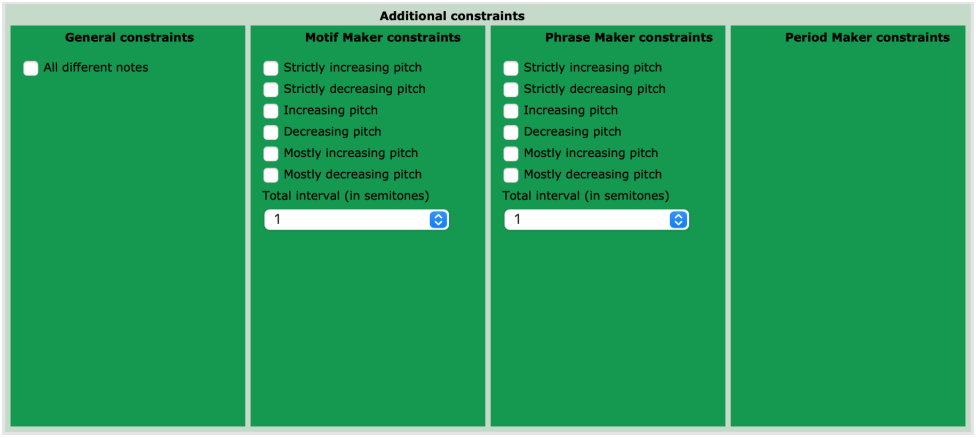


Figure 6.7: Optional constraints panel of the melodizer object editor window

solver finds a solution, a voice editor window containing the found melody will automatically open. You can modify it if you want, and then close the window.

If you want to **see the current solution** again, you can click the *See solution* button. If you want to **keep the solution for later** and keep searching for other solutions, you can press the *Keep solution* button. Doing so will save the solution in the pop-up menu below. Pressing the *Next* button again will erase the previous solution if it wasn't saved. If you want to **see a saved solution with the input chords**, you can select it in the pop-up menu and then press the *See with chords* button. This will open a window displaying the solution with the chords. Be careful that any modification in this window are not saved and will be lost.

If you want to **keep a saved solution for later**, you can add it to the list of motifs, phrases or periods by pressing the *Add to motifs*, *Add to phrases* or *Add to periods* button. You can modify the parameters and optional constraints for another problem, and click the *Start* button to start searching for solutions. Note that doing so will erase all solutions saved in the search panel, so make sure you save what you want by adding solutions to the motifs, phrases or periods accordingly.

6.6 Assembling a melody using generated solutions

Once you have kept solutions you liked, you might want to combine them together to create a musical piece. This can be done through the solution assembly panel, depicted in figure 6.8. The list of saved motifs, phrases and periods can be found in the pop-up menus. You can create a melody by adding different elements before or after the current melody using the corresponding buttons. The current melody can be opened and edited by pressing the *Show melody* button. You can add elements, then modify the melody or add notes/measures to the melody, then add more elements, etc. The melody can be saved as a phrase or as a period using the corresponding button, and can be erased to start a new melody by pressing the *Reset melody* button. The lists of motifs, phrases and periods can also be erased by pressing the corresponding *Reset* button.

6.7 Making Melodizer interact with other elements in OpenMusic

Other elements can interact with Melodizer by connecting to its inlets/outlets. Solutions can be exported from the melodizer object to be used in the patch using the "solutions-list", "motifs-list", "phrases-list", "periods-list", "output-solution", "output-motif", "output-phrase", "output-period" and "melody" outlets.

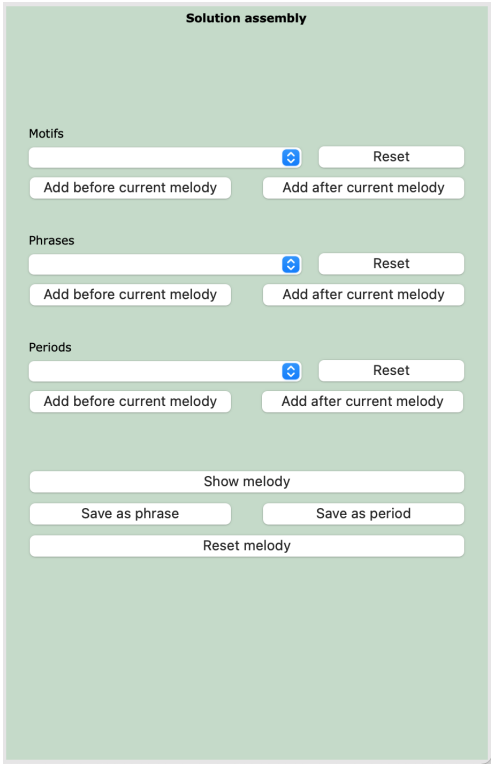


Figure 6.8: Solution assembly panel of the melodizer object editor window

Chapter 7

Composer use scenarios for Melodizer

This chapter aims to give a few examples of how Melodizer can be used in practice by composers to improve their expression and composition abilities. The different scenarios will be written as actual situations that composers might find themselves in, and should not be used as a tutorial. The reader should read chapter 6 for a tutorial on how to use Melodizer. Assuming you have installed Melodizer and have it open in front of you, you can follow these examples yourself. The following examples are different uses of Melodizer, including:

- Finding a melody to harmonize with a sequence of chords that the composer came up with
- Revisiting an existing piece by modifying the pitch of the melody
- Finding a melody to play on top of a sequence of chords using Rhythm-Box to generate rhythm
- Generating motifs, phrases and periods using optional constraints
- Generating variations of an existing melody while keeping its general feel
- Using Melodizer to generate and combine musical elements to make a musical piece

7.1 Reminder

This section gives a reminder about a few notions that are useful to understand this chapter.

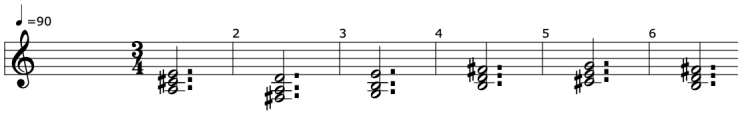
Creating a melodizer object To create a melodizer object, you can click on Classes/Libraries/Melodizer/ALL/MELODIZER and the click anywhere on the patch. Double clicking on the object will open its editor window.

Inlets/Outlets You can give input to Melodizer by connecting boxes to its inlets (the circles on top of the object box). Trying to connect them will show their name. You can get output from Melodizer similarly by connecting its outlets to other boxes.
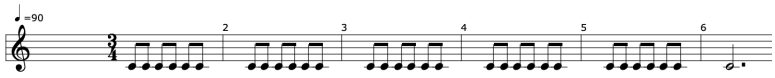
Motif A motif is a short melodic idea. It can be as simple as two notes, and can be more complex but is generally really short (up to one measure long).

Phrase A phrase is a little bit longer than a motif. It is generally a few bars long, and expresses a more complex melodic idea.
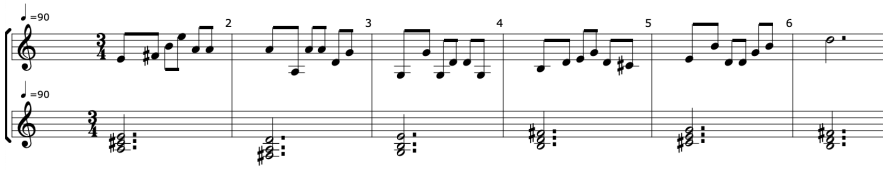
Period A period is a complete melodic idea. It can be composed of several phrases.



(a) Chord sequence given as input to the melodizer object



(b) Rhythm given as input to the melodizer object



(c) Solution suggested by Melodizer for the input presented above

Figure 7.1: Input and output of Melodizer when looking for a melody to fit a chord progression

7.2 Finding a melody to harmonize with a sequence of chords

A composer came up with a chord progression (figure 7.1a), and they would like to find a melody that can be played with this chord sequence. They also came up with a rhythm (figure 7.1b) for the melody they want, but didn't settle on the pitch of the melody. To do so, they first have to set the mode of the tool to "melody-finder". Then, they can either connect the voice objects for the chords and the rhythm to the corresponding inlets, or they can edit them using the corresponding buttons in the editor window of the melodizer object. Once that is done, they can select the desired tonality (here, B minor would be a good idea) using the pop-up menus to select a key and a mode. After that, they can press start and start navigating through the different solutions by pressing the next button and edit them or save them if they want. Figure 7.1c shows one of the suggested melodies, and figure 7.2 shows the complete patch for this scenario, with one of the solutions suggested by Melodizer. The input and output can be listened to by scanning the QR code or by clicking the following url : `https://www.youtube.com/watch?v=enxMEUmD6sM`.

7.3 Revisiting an existing piece

A composer has written a piece, with chords and a melody (figure 7.3a), and they wish to try using different melodies with the same rhythm to see what would work best with these chords. They can do that using Melodizer. First, they should set the mode to "melody-finder" and set the chords and the melody as input of the melodizer object. They can then select a tonality for the melody, that can be the same as the original melody idea they had, but can also be different. After that, they can specify additional constraints if they want, and then start searching for solutions. They can edit and save the solutions. Figure 7.4 shows the use of Melodizer to generate an alternative melody for the song "Puisque tu pars" by Jean-Jacques Goldman [9], and figure 7.3 shows the original song and the new song side-to-side. The original and new versions can be listened to by scanning the QR code or by clicking on the following link : `https://youtu.be/Rdy2XGoMiL4`.

7.4 Finding a melody to play on top of a sequence of chords using Rhythm-Box to generate the rhythm of the melody

A composer has an idea of a chord progression (figure 7.5a) and a rhythm (figure 7.5b), and they want to experiment with the rhythm and the pitch of the melody. To be able to use Rhythm-Box [11] with Melodizer,

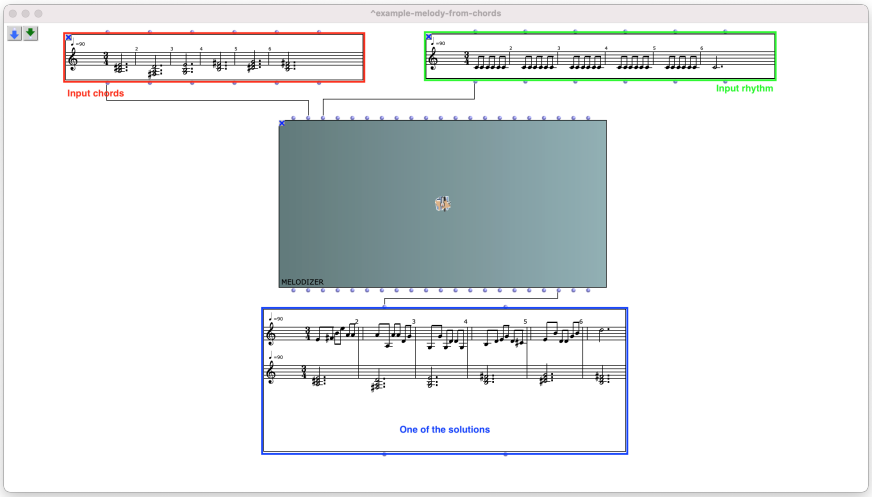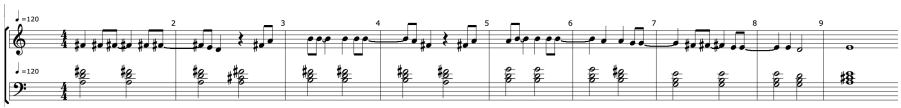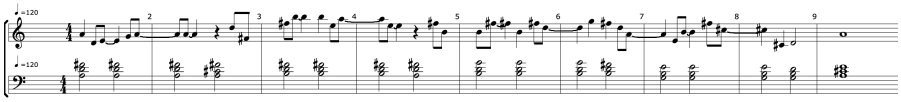


Figure 7.2: Patch for finding a melody harmonizing with a given chord sequence



(a) Simplified version of “Puisque tu pars” by Jean-Jacques Goldman



(b) Melody generated by Melodizer from the chords and melodic rhythm of “Puisque tu pars”

Figure 7.3: Example of the use of Melodizer to revisit “Puisque tu pars” by Jean-Jacques Goldman [9]

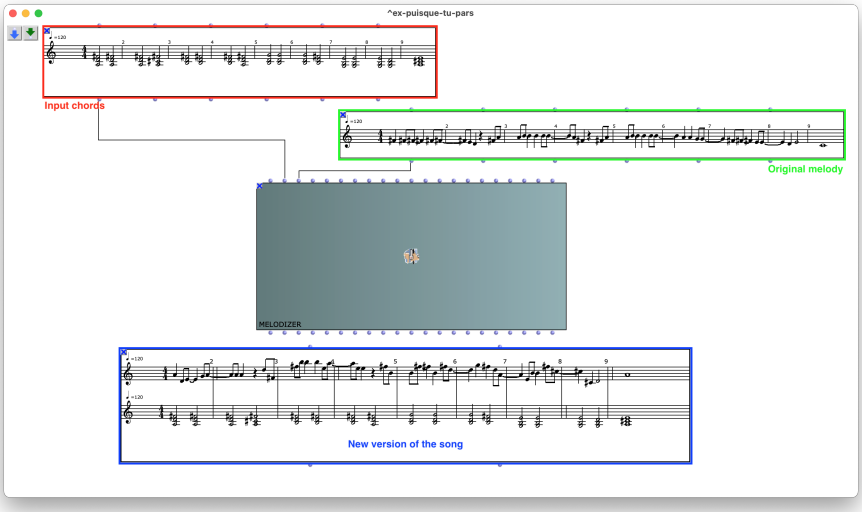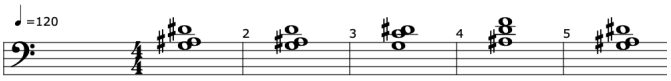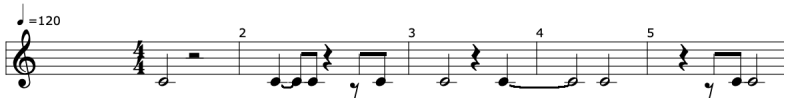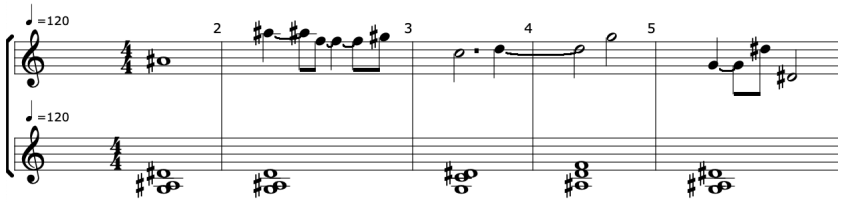


Figure 7.4: OpenMusic patch to generate an alternative melody for the song “Puisque tu pars” by Jean-Jacques Goldman



(a) Chords given as an input to the melodizer object (chords from "Fix You" by Coldplay)



(b) Rhythm used to generate other rhythms with Rhythm-Box



(c) Solution given by Melodizer for the input presented above, with the help of Rhythm-Box to generate a new rhythm

Figure 7.5: Input and output of Melodizer using Rhythm-Box to generate a melody to fit a chord progression

the composer has to give the input to the melodizer object through the inlets and not through the interface. A rhythm can be generated using Rhythm-Box[1] and given as an input to Melodizer, together with the chord sequence. After that, the composer can search for solutions like in section 7.2. Figure 7.6 shows the patch to do that in OpenMusic, and the input and output can be listened to by scanning the QR code or clicking on the following link: https://www.youtube.com/watch?v=d1hrL-Qs21I.

7.5 Generating motifs, phrases and periods using optional constraints

A composer has come up with a chord sequence (figure 7.7a) and a rhythm (figure 7.7b), and they want to find a melody to play on top of it. They want that melody to go in a certain direction, i.e up or down. They can use the "melody-finder" mode of Melodizer to achieve that result. They first have to specify the chord progression and the rhythm for the melody to the tool, and they can then select a tonality and additional constraints that the generated melody should follow. Figure 7.8 shows a patch generating a phrase for the input chords and rhythm at the top of the patch, with the constraint that the pitch must mostly be going up. The video on how to do it using Melodizer can be found by scanning the QR code or by clicking the following link: https://www.youtube.com/watch?v=QQwzvImpABI.

7.6 Generating variations of a melody while keeping the overall feel of the melody

A composer has written a melody with a chord progression (figure 7.9a), and they want to make variations of this melody while keeping the overall feel of the melody. Using Melodizer, they can select the "variation-maker" mode and give their melody and chords as input to the melodizer object. Melodizer will then suggest variations of that melody to the composer. Figure 7.9b shows one of the solutions suggested by Melodizer, and figure 7.10 shows a patch using Melodizer to generate variations of a melody. The original melody and the variation can be listened to by scanning the QR code or by clicking on the following link: https://www.youtube.com/watch?v=WyyXGCjOLXo.

[1] see https://github.com/blapiere/Rhythm-Box for a tutorial on how to use Rhythm-Box

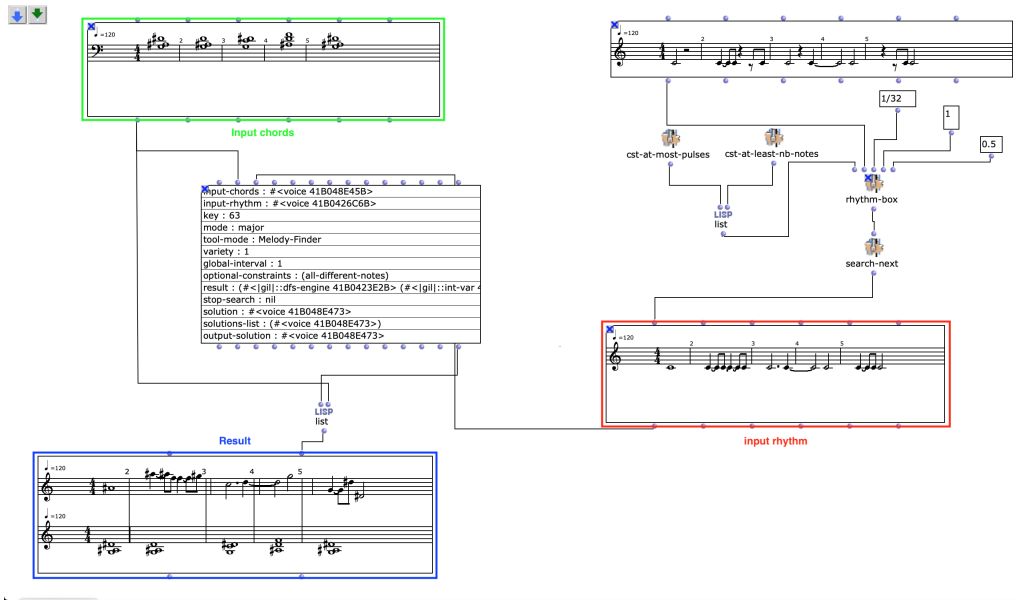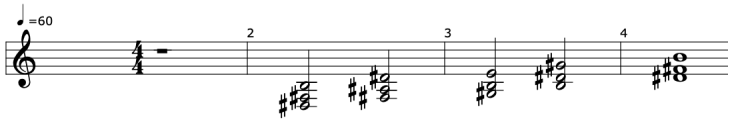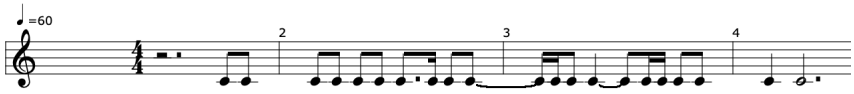


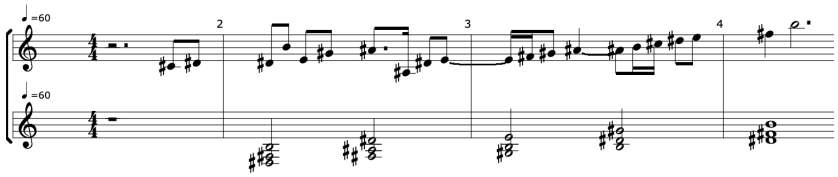
Figure 7.6: Example of the use of Melodizer with Rhythm-Box to find a melody harmonizing with a given chord sequence (chords from "Fix you" by Coldplay)



(a) Chord sequence given as an input to the melodizer object



(b) Rhythm given as an input to the melodizer object



(c) Solution suggested by Melodizer for the input presented above, using the mostly-increasing-constraint (see chapter 3) on the melody in the tonality of B major

Figure 7.7: Example of the use of Melodizer to generate a melody with a specific direction

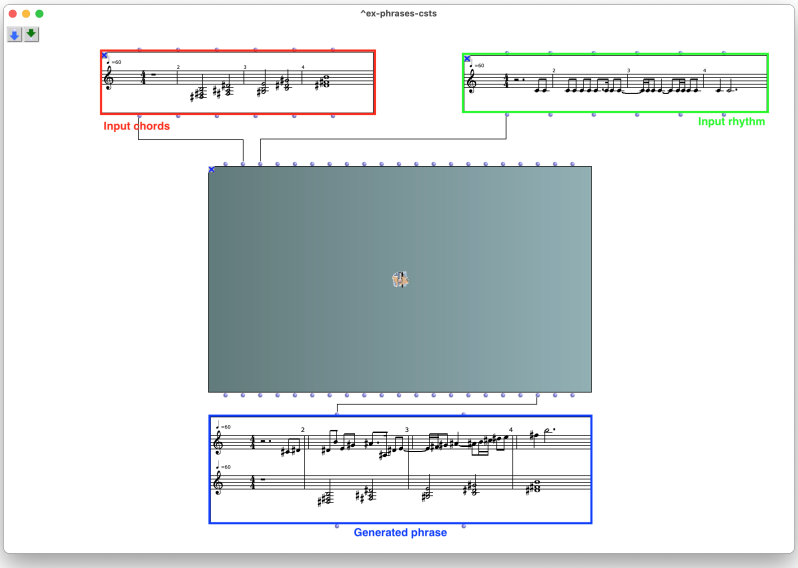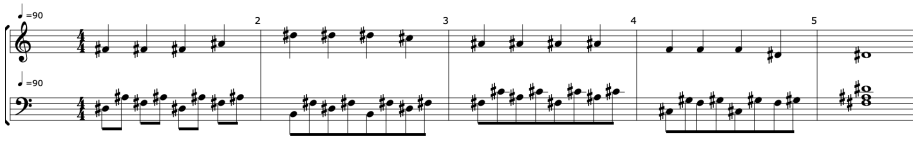
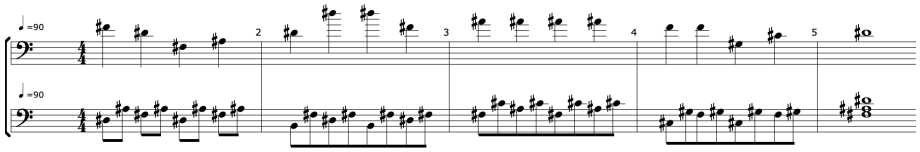


Figure 7.8: OpenMusic patch to generate a musical phrase based on the input chords and rhythm respecting a constraint



(a) Original melody given by the composer to Melodizer ("Faded" by Alan Walker)



(b) Variation of the melody generated by Melodizer

Figure 7.9: Original melody and variation of "Faded" by Alan Walker [18]

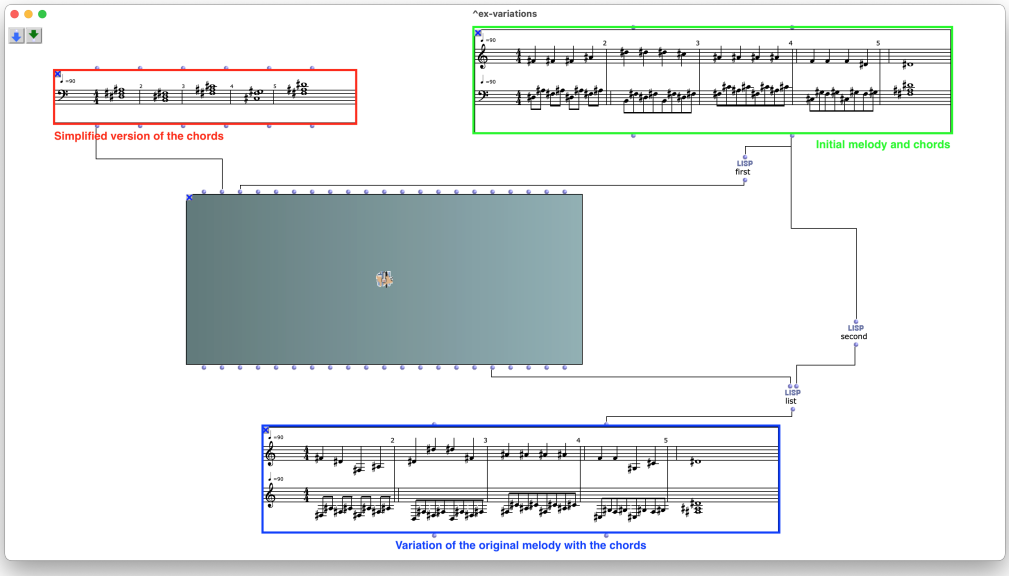
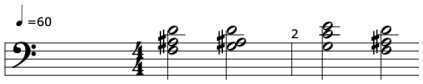


Figure 7.10: Patch using Melodizer to generate variations of a melody
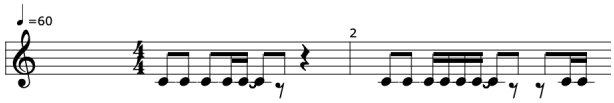
7.7 Using Melodizer to generate different musical phrases, creating variations of them and combining them together to create a musical piece

A composer has a couple chord progressions and rhythms in mind (figure 7.11), and wants to create a musical piece by combining them and creating variations of those melodies. One way to do that is to first generate melodies using the "melody-finder" mode of Melodizer, and then saving the phrases to be combined later with other melodies. Variations of a melody can then be obtained by using the "variaton-maker" mode of Melodizer, and musical periods can be created by combining phrases together. Figure 7.12 shows an example of a musical piece generated from the input of figure 7.11, and a video of the use of Melodizer to create this simple musical piece can be found by scanning the QR code or by clicking the following link : `https://www.youtube.com/watch?v=Cjk4vAdCMq0`. This example contains the following operations :

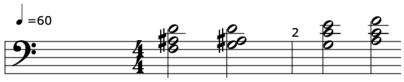
- Creating of a melodizer object in the patch
- Editing of the input chords and input rhythm objects with the chord sequence and rhythm of the problem through the interface
- Using Melodizer to generate melodies
- Navigating between the different solutions
- Editing of the solutions
- Saving solutions in the search panel to examine them later
- Saving solutions as phrases to be combined with other musical elements
- Extracting elements from Melodizer through the outlets
- Using Melodizer to generate variations of a melody
- Combining musical phrases to create a musical period.
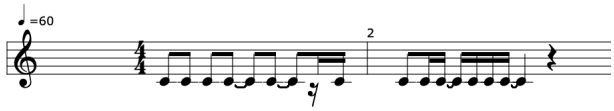


(a) First chord progression



(b) First rhythm idea



(c) Second chord progression



(d) Second rhythm idea

Figure 7.11: The composer's initial ideas
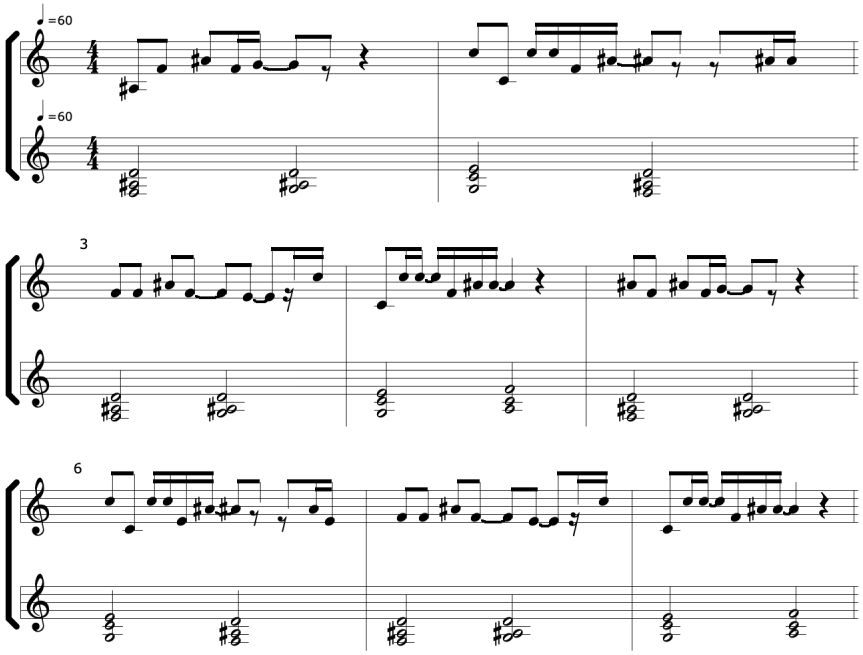


Figure 7.12: Musical piece created by generating different phrases and combining them using Melodizer

Chapter 8

GiL

The first step towards creating a constraint-based tool in OpenMusic was to bring the constraint programming aspect to Lisp. This has been done by Baptiste Lapière [11] with the creation of GiL[1], an interface between Gecode[2] and Lisp using CFFI[3]. GiL will therefore be used to handle the constraints part of Melodizer, and some functionalities will be added, which are described later.
This chapter does not aim to explain how GiL works in full detail, the curious reader can read [11] for more information on that. The first section explains the basic principles of GiL, the next section details the features that were added to make Melodizer work. This is followed by an example of how to use GiL to solve CSPs in Lisp. The last section will explain briefly how to contribute to GiL. The new version of GiL can be found at `https://github.com/sprockeelsd/GiLv2.0`.

8.1 Structure

GiL is composed of two main parts : a C wrapper and a Lisp wrapper. Since CFFI provides an interface between C and Lisp, but not C++, the first step is to create a C library that executes Gecode functions, that can then be called with Lisp code in OpenMusic.

8.1.1 C Wrapper

The C Wrapper is made of two main parts: the first part, called the space wrapper, contains all the calls to Gecode functions. It includes the definition of the WSpace class, including all its methods to create variables, post constraints and specify branching strategies, as well as the search engines classes. The second part, called the gecode wrapper, is the external C library calling the methods from the space wrapper. The functions from the gecode wrapper will then be called from the Lisp Wrapper, which will be detailed later.

In Gecode, variables, constraints and branching are modeled within a subclass of the "Space" class, which represents the problem. In GiL, this translates to a "WSpace" class with methods to add variables, post constraints, and specify a branching strategy. Upon creation, the WSpace instance is casted to a void pointer to be useable in CFFI. Everytime a function from the C external library that uses a method of the space wrapper is called, the pointer is casted back to a WSpace pointer to call the method. This is illustrated in figure 8.1.

Variables Variables in Gecode are instances of a class, so another way to reference them is necessary to be able to refer to them in C. The WSpace class has a vector as attribute for each type of variables (integer, boolean) and protected methods to retrieve a variable or variable array based on their indexes. Upon creation, a variable is pushed to the corresponding vector and its index in the vector is returned so it can be accessed later.

[1] https://github.com/blapiere/GiL
[2] https://www.gecode.org
[3] The Common Foreign Function Interface, see https://common-lisp.net/project/cffi/

```
// Wraps the WSpace constructor.
void* computation_space() {
    return (void*) new WSpace();
}

//Wraps the WSpace add_intVar method.
int add_intVar(void* sp, int min, int max) {
    return static_cast<WSpace*>(sp)->add_intVar(min, max);
}
```

Figure 8.1: Code from gecode wrapper, for creating a new space and adding an integer variable to it.

variable selection strategies	value selection strategies
variable with smallest domain	smallest value of the domain
random variable	random value
variable with the highest degree	values not greater than (min + max)/2
	values greater than (min + max)/2
	biggest value not greater than the median

Figure 8.2: Currently supported branching strategies

Constraints Constraints are wrapped in a similar way. For each constraint, one or more methods are created in the space wrapper and they take the indexes of variables as arguments. The corresponding WSpace method then translates indexes into variables to post the constraints.

Branching Branching is also wrapped in a similar manner. Before this master's thesis, there was no way to specify the branching strategy in GiL, and the default strategies were choosing the variable with the smallest domain, and choosing the smallest value of the domain first.

Search Engines Search engines are wrapped separately from the space. They are modeled as a class (e.g. WdfsEngine) with an attribute that is a reference to the corresponding Gecode search engine. Before this master's thesis they provided a method to call their next function, that returns a space that is a solution of the CSP if there is any.

8.1.2 Lisp Wrapper

Now that there is an external C library, it can be wrapped in Lisp with the help of CFFI. The Lisp wrapping is done in two stages in GiL. First, foreign functions are defined using CFFI, each calling a specific function from the Gecode Wrapper. Those functions are the link between Lisp and C, calling C functions and giving the returned value to Lisp. Then, the foreign functions are wrapped by Lisp functions to provide a more readable library, making using GiL as similar to using Gecode as possible.

8.2 Contribution

Since the goal of this master's thesis was not to develop GiL but rather to produce a useful tool for composers, only necessary features were added to GiL. However, these additions were designed in a way that allows to easily develop them or add other features without creating conflicts.

Branching Branching on integer variables was improved by adding the possibility to specify branching strategies, both for variable and value selection. This was done by passing integers as arguments, representing the branching strategies, using pre-defined parameters to make it more user friendly. These integers are then translated to actual Gecode strategies in the space wrapper. Figure 8.2 shows the currently supported strategies.

Search engines The search engine constructors have been modified to support search options (more explanations below). Additionally, they now have a method allowing to detect if the search engine has been stopped. The GiL Lisp function to create search engines has been modified to make it more modular and intuitive, by allowing to specify the type of search engine to create instead of setting an additional parameter to "t" or "nil" as it was before, allowing to support more than 2 types of search engines.

Search options In Gecode, it is possible to specify options for the search of solutions. It is done by using Options objects. In GiL, they are wrapped in a similar way to search engines, in a WSearchOptions class with one attribute that is an Options object from Gecode. The class has methods allowing to set the different options. Currently, only the "threads" and "stop"[4] options are supported, but others can easily be added in a similar way.

Stop Option The "stop" option in Gecode works by creating a stop object and specifying a limit depending on its type. GiL currently only supports TimeStop objects, and therefore only allows to specify a time limit and not for example a limit in terms of number of nodes visited. TimeStop objects are wrapped the same way search engines and options objects are, and they have a method that allows to reset the timer.

Value precedence constraint The Gecode "precede" constraint has been added to GiL. Given a variable array x and two integers s and t, the constraint ensures that there will always be a "s" before a "t". In other words,

$$x_0 \neq t$$

$$x_j = t \Rightarrow \exists i \in [0, j[\mid x_i = s$$

Count constraint One variation of the "count" constraint has been added to GiL. Given an array of variables x, a set of integers c, a relation rel_type and an integer z, the constraint ensures that the number of variables in x that take a value in c has relation rel_type to z. In other words,

$$\#\{i \in [0, ...|x| - 1] \mid x_i \in c\}\ rel_type\ z$$

8.3 Example of use

This section will show how the different steps of creating a simple CSP compare in Gecode and in GiL. The problem chosen here is a very simple problem with 3 variables with a domain ranging from 1 to 4, where the only constraint posted is that all variables must be different. The branching strategies are selecting the variable with the smallest domain first and selecting the smallest value of the domain first. A depth first search engine is used. Figure 8.3a show the code for this simple problem in Gecode, and figure 8.3b shows the exact same problem in Lisp using GiL.

8.4 How to add to GiL

Adding a feature to GiL can be done in four steps. First, it must be wrapped in the space wrapper. If you wish to add a constraint, a simple function calling the Gecode function is enough. If you want to add something more complex like a search engine type, you will probably need to wrap it into a class. Then, one or more functions must be created in the gecode wrapper (external C library) to call the functions created in the space wrapper. Note that pointer arguments for functions in the gecode wrapper must be of type void*, and casted into the appropriate type when calling space wrapper functions. The third step is to create Lisp functions that call the C functions using CFFI. Finally, the last step is to add more user-friendly functions to call the foreign functions created with CFFI, using for example the "int-var" class.

[4] for a comprehensive list of options, see [14], Chapter 9.3

```
class Simple_problem : public Space {
  protected:
    IntVarArray vars;
  public:
    Simple_problem(): vars(*this,3, 1, 4){ // initialize the variables
      // post the constraints
      distinct(*this, vars);
      //specify the branching
      branch(*this, vars, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
    }
    ... //code that is not relevant to show how Gecode and GiL compare
};

int main(int argc, char* argv[]) {
  Simple_problem* m = new Simple_problem(); //create the space
  Gecode::Search::Options opts;// create and initialize the search options
  Gecode::Search::TimeStop maxTime(500);//create and initialize the time stop object
  opts.stop = &maxTime; //specify the time limit in the search options
  DFS<Simple_problem> e(m);// create the search engine
  delete m;

  // search and print all solutions
  while (Simple_problem* s = e.next()) {
    s->print(); delete s; maxTime.reset();// reset timer
  }
  return 0;
}

```

(a) Simple CSP in Gecode

```
(defun dummy-problem ()
    (let ((sp (gil::new-space))     ; create the space
    vars se tstop sopts max id-list)
        (setq vars (gil::add-int-var-array sp 3 1 4)); initialize the variables
        ; post constraints
        (gil::g-distinct sp vars)
        ; specify branching
        (gil::g-branch sp vars gil::INT_VAR_SIZE_MIN gil::INT_VAL_MIN)

        ; search options
        (setq sopts (gil::search-opts)); create the search options object
        (gil::init-search-opts sopts); initialize it
        ;time stop
        (setq tstop (gil::t-stop)); create the time stop object
        (gil::time-stop-init tstop 500); initialize it (in ms)
        (gil::set-time-stop sopts tstop); specify the time limit in the search options

        ; create the search engine
        (setq se (gil::search-engine sp (gil::opts sopts) gil::DFS))
        ; return for the ``next" function
        (list se vars tstop sopts)))

(defun search-next (l)
    (let ((se (first l)) (pitch* (second l)) (tstop (third l)) (sopts (fourth l)) sol
    pitches)
        (gil::time-stop-reset tstop);reset the timer before launching the search
        (setq sol (gil::search-next se)); search for the next solution
        (if (null sol) (error ``No more solutions"))
        (setq pitches (gil::g-values sol pitch*)); store the values of the solution
        (print pitches)))

```

(b) Simple CSP in lisp using GiL

Figure 8.3: Simple CSP in Gecode and in GiL

Chapter 9

Conclusion

This master's thesis had one main objective, namely to provide composers with a useful tool to assist them in musical composition and to improve their expression abilities. This objective can be split into three parts : developing GiL further, creating a musical constraint satisfaction problem and creating a tool and an interactive interface for composers.

9.1 GiL

GiL [11] was an already existing, though incomplete, interface between OpenMusic and Gecode. It had general features allowing to create CSPs in OpenMusic, but was lacking several features allowing for customization of problems. The following features were added to GiL:

- Multiple branching strategies : instead of having only one default variable selection and value selection strategy, a GiL user now has a choice between a few strategies for both variable selection and value selection. Other strategies can easily be added to GiL due to the design of the branching function.
- Search engines : Several features have been added to search engines in GiL, such as the possibility to know if the solver has been stopped or not, and the ability to specify options for the search. For now, the available options are setting the number of threads to use, and setting a time limit to the search. Other options can easily be added due to the design of the search engine wrapping.
- Constraints : a few constraints have been added to GiL, making it more versatile, including several versions of the count constraint or the value precedence constraint from Gecode.

9.2 Two musical CSPs

Two CSPs have been created during this master's thesis. The first one can be used to generate melodies based on a chord sequence and a rhythm, and the second one can be used to generate variations of a given melody. Along with these CSPs were provided a set of musical constraints allowing the composer to customize the problem to better fit their needs with optional constraints, and allowing to guarantee a relatively good-sounding melody with some mandatory constraints. A variety of branching strategies were also provided, depending on the problem. Here are some of the provided constraints :

- A constraint to ensure that the notes of the melody will be in a given tonality
- A constraint restraining the possible values of a note played at the same time as a chord
- A constraint restraining the possible values of the interval between two consecutive notes
- A set of constraints allowing to give a general direction to the melody

An algorithm for the search for a solution has also been provided, with a way of guaranteeing that consecutive solutions would be relatively different, making it more interesting for the user, with the use of a branch-and-bound search engine. The composer has full control on the variety of solutions. This strategy was not implemented in the final tool due to a problem implementing it using GiL. A technique to stop the search if it takes too long has also been implemented using threads.

9.3 Melodizer, an OpenMusic tool to assist composers in their creation process

The tool that was the objective of this master's thesis is an OpenMusic object called Melodizer. It has an intuitive interface allowing composers to easily interact with it, and allows to use the CSPs developed to solve a variety of musical problems. It can easily interact with other OpenMusic components and with Rhythm-Box [11]. Here are some of the things you can do with Melodizer:

- Create a melody to harmonize with a sequence of chords
- Revisit an existing piece
- Generate motifs, phrases, and periods and combine them to create musical pieces
- Generate variations of a melody while keeping its overall feel

9.4 Future work

Though this master's thesis has provided a tool able to help composers in a variety of ways, it is only scratching the surface of what can be done using constraints programming to generate melodies. This section gives a few ideas of how Melodizer could be improved and how to take it a step further.

9.4.1 Developing GiL

An obvious improvement would be to keep developing GiL even further. The more complete it is, the more complex models can be created and solved. The first obvious step would be to make the branch-and-bound search engine work, as this would also improve Melodizer quite a bit. Other constraints, branching heuristics, and types of variables would be a great addition as well. There are a lot of features in Gecode that are designed to make the search more efficient, like symmetry breaking, that could be added as well. Finally, having a visual interface for GiL in OpenMusic allowing to create CSPs directly in OpenMusic would be a great tool for composers as well.

9.4.2 Adding other CSPs to model a variety of problems

For now, there are only two CSPs that can model a variety of problems, but more could be added to make the tool even more versatile. This section discusses a few ideas of how to improve the constraint oriented part of Melodizer.

CounterPoint CSP A CSP modelling a problem where the composer would give a series of voices played simultaneously, that could generate an additional voice that can be played on top of the input voices while respecting a set of constraints.

Harmonizing CSP A CSP that can generate chords to harmonize with a melody given as an input by the composer. There could be different styles of harmonization.

Genre CSP A CSP that takes a musical piece and that can transform it in another genre (e.g from reggaeton to rock,...).

Additional constraints Additional constraints could be added to further improve the customizability of the problem by composers. They could for example select an instrument to play the melody, that would restrain the domain of the notes based on the ambitus of the instrument. Some constraints could be added to the mandatory constraints as well to make the proposed melodies even better, but one has to be careful not to constraint the initial problem too much because that might hide some interesting solutions to the user.

9.4.3 Improving Melodizer

Melodizer offers an interface that is easy to use for composers, however it could still be greatly improved. This sections offers a few ideas of improvements.

Selecting notes to post constraints One addition would be to allow the composer to select a note or group of notes from a proposed solution, and only launch the search for that group of notes. This would allow the composer to further customize the problem to best suit their needs.

Add modes Currently, Melodizer only has two modes but adding more CSPs would require to add more modes to the Melodizer tool. Its interface could also dynamically adapt depending on the selected mode, only allowing the selection of relevant constraints for example.

Rhythm-Box Rhythm-Box could also be added to the interface of Melodizer, allowing to generate rhythms instead of forcing the composer to come up with rhythms themselves. Rhythm-Box could also be improved, see [11] for ideas on how to improve it.

Input Currently, the input for the supported mode is restricting in the sense that it has to be flat three note chords. However, typically composers like to create more complex patterns and use chords that have more than three notes as well. The input could be generalized to more closely represent the composer's idea.

Bigger picture Melodizer currently offers a simple way of combining musical ideas and solutions created by the CSP, but it could also be improved to make it easier for composers to combine their ideas. One possible approach would be based on PowerPoint. Each instance of Melodizer would be like a PowerPoint slide, with all its memorized musical elements. It could be possible to copy motifs, phrases, periods from one instance to the other, and combine them into a musical network. The way of exporting melodies from Melodizer to be used with other OpenMusic components could also be improved.

Bibliography

[1] Torsten Anders. "Composing Music by Composing Rules: Design and Usage of a Generic Music Constraint System". PhD thesis. School of Music & Sonic Arts, Queen's University Belfast, 2007.

[2] Torsten Anders and Eduardo Miranda. "Constraint Programming Systems for Modeling Music Theories and Composition". In: *ACM Computing Surveys* 43 (Oct. 2011), 30:1–30:38. DOI: 10.1145/1978802.1978809.

[3] Jean Bresson. "Notions de programmation par objets en Common Lisp et dans OM". In: (2010).

[4] Coldplay. *Fix you*. Musical work (song). 2005.

[5] *Constraint programming*. In: *Wikipedia*. Page Version ID: 1057052808. Nov. 25, 2021. URL: https://en.wikipedia.org/w/index.php?title=Constraint_programming&oldid=1057052808 (visited on 12/20/2021).

[6] A Danhauser. *Théorie de la musique*. Lemoine éditions, 1994.

[7] Imagine Dragons. *West Coast*. Musical work (song). 2018.

[8] Jean-Louis Foucart. *Vingt leçons d'harmonie pour comprendre et composer la musique*. 2004. URL: www.foucart.net.

[9] Jean-Jacques Goldman. *Puisque tu pars*. Musical work (song). 1988.

[10] Scott Jarrett and Holly Day. *Music Composition for dummies*. Second edition. John Wiley & Sons, Inc, 2021.

[11] Baptiste Lapière. "Computer-Aided Musical Composition". Master's thesis. Université catholique de Louvain, 2020.

[12] *OpenMusic Documentation - OM 6.6 User Manual*. URL: https://support.ircam.fr/docs/om/om6-manual/co/OM-User-Manual.html (visited on 11/26/2021).

[13] Pierre Schaus. *LINGI2365 : Constraint Programming*. Academic course. Université catholique de Louvain. 2020.

[14] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode*. 2019. URL: https://www.gecode.org/doc/6.2.0/MPG.pdf.

[15] *The Common Lisp Cookbook - Threads*. URL: http://cl-cookbook.sourceforge.net/process.html (visited on 01/02/2022).

[16] Charlotte Truchet. "Contraintes, Recherche Locale et Composition Assistée par Ordinateur". PhD thesis. Université Pierre et Marie Curie, 2004.

[17] Charlotte Truchet, Carlos Agon, and Philippe Codognet. "A Constraint Programming System for Music Composition". In: *Musical Constraints Workshop, 7th International Conference on Principles and Practice of Constraint Programming* (Nov. 2001).

[18] Alan Walker. *Faded*. Musical work (song). 2015.

Appendix A

How to install Melodizer

This appendix explains in detail how to install Melodizer and set it up. Melodizer should work on all platforms, however it requires GiL to work and currently GiL only works on MacOS. It does not work on Windows because the Lisp license used by OpenMusic is a 32bit version, and the Gecode Windows version is 64 bits by default. You could theoretically download Gecode in a 32bit version for Windows, but it is not recommended to compile Gecode yourself.

A.1 Download

In order to use Melodizer, you will need to follow these steps :

- Download and install Gecode : `https://www.gecode.org/download.html`
- Download and install OpenMusic : `https://openmusic-project.github.io/openmusic/`
- Download GiL : `https://github.com/sprockeelsd/GiLv2.0`
- Download Melodizer : `https://github.com/sprockeelsd/Melodizer`
- You might want to download Rhythm-Box as well, if you want to use it. It is not necessary to use Melodizer : `https://github.com/blapiere/Rhythm-Box`

For GiL, depending on what MacOS version you are using, there might be a problem with relative paths and the dynamic library might not find Gecode. This can be fixed with the script that can be found in the C++ folder of GiL. You should edit it so that the full path to Gecode corresponds to the actual path to Gecode on your computer, and then execute the script.

A.2 Load the libraries in OpenMusic

Once you have downloaded everything, the next step is to launch OpenMusic. The window in figure A.1 will appear. Create a new workspace and create a patch in your workspace by clicking on file/new patch. Your workspace should look like figure A.2. Now, it is time to load the libraries. To do so, click on window/library, and a window will appear. Then, click on file/Add remote user library, and navigate to your GiL and Melodizer folders. They shoud appear in the library window, and look like figure A.3. Then, all you have to do is right-click on the melodizer library and click on Load Library. Melodizer can now be used in a patch. Chapter 6 explains how to use Melodizer.

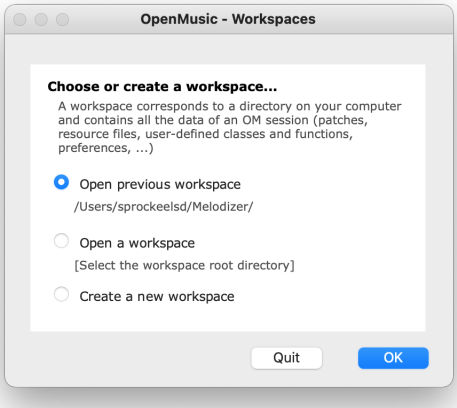


Figure A.1: Workspace selecting window in OpenMusic

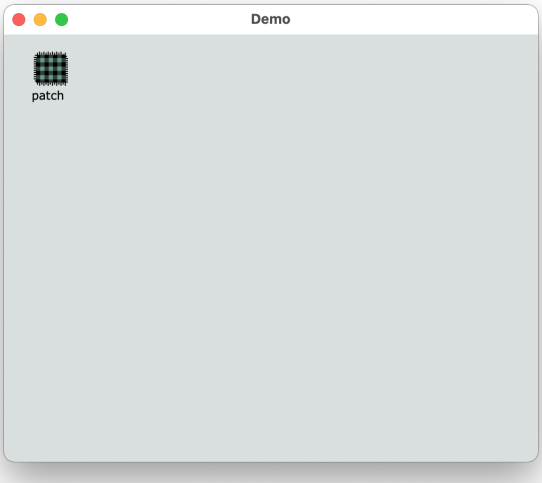


Figure A.2: Workspace containing a patch

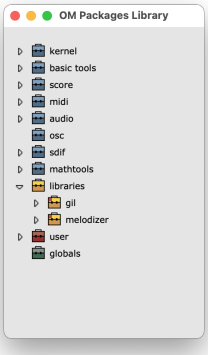


Figure A.3: Library window in OpenMusic when the libraries have been fetch

Appendix B

GiL source code

This appendix gives the code for the interface between Gecode and Lisp, GiL. GiL has been developed by Baptiste Lapière [11] and has been further developed during this master's thesis. It is composed of two main parts : the C wrapper, wrapping the C++ Gecode calls into C functions, and the Lisp wrapper, wrapping the C functions into Lisp. The additions and changes that were made during this master's thesis are highlighted in blue. The current version of GiL can be found at `https://github.com/sprockeelsd/GiLv2.0`

B.1 C wrapper

The C wrapper wraps the Gecode function calls into C functions. It is composed of four files :

- **space_wrapper.hpp** declares the classes that will wrap the Gecode calls. WSpace wraps methods to create a search space, create variables, post constraints and branching. WTimeStop wraps methods to specify a time limit to the search. WSearchOptions wraps methods to create and set a variety of options for the search. WbabEngine and WdfsEngine wrap methods to create search engines, and methods to search for the next solution and to know if the search has been stopped or not.
- **space_wrapper.cpp** implements the classes and methods declared in space_wrapper.hpp
- **gecode_wrapper.hpp** declares the external C library that will be called by the Lisp wrapper.
- **gecode_wrapper.cpp** implements the methods of the external C library declared in gecode_wrapper.hpp.

The following sections show the code of these files.

B.1.1 space_wrapper.hpp

```
1  #ifndef space_wrapper_hpp
2  #define space_wrapper_hpp
3
4  #include <vector>
5  #include <iostream>
6  #include <stdlib.h>
7  #include <exception>
8  #include "gecode/kernel.hh"
9  #include "gecode/int.hh"
10 #include "gecode/search.hh"
11 #include "gecode/minimodel.hh"
12 #include "gecode/set.hh"
13
14 using namespace Gecode;
15 using namespace Gecode::Int;
16 using namespace std;
17 using namespace Gecode::Search;
18
19 class WSpace: public IntMinimizeSpace {
20 protected:
21     vector<IntVar> int_vars;
```

```
22      vector<BoolVar> bool_vars;
23      int i_size;
24      int b_size;
25      int cost_id;
26
27      int* solution_variable_indexes;// to know what variables will hold the solution, useful for bab
28      int var_sol_size;
29
30      //======================
31      //= Variables from idx =
32      //======================
33
34      /**
35       Return the IntVar contained in int_vars at index vid.
36       */
37      IntVar get_int_var(int vid);
38
39      /**
40       Return the BoolVar contained in bool_vars at index vid.
41       */
42      BoolVar get_bool_var(int vid);
43
44
45      //====================
46      //= Args for methods =
47      //====================
48
49      /**
50       Return an IntVarArgs of size n, containing the n IntVars contained in
51       int_vars at indices vids.
52       */
53      IntVarArgs int_var_args(int n, int* vids);
54
55      /**
56       Return a BoolVarArgs of size n, containing the n BoolVars contained in
57       bool_vars at indices vids.
58       */
59      BoolVarArgs bool_var_args(int n, int* vids);
60
61      /**
62       Return an IntArgs of size n, containing the n values in vals
63      */
64      IntArgs int_args(int n, int* vals);
65
66      /**
67       Return the expression int_rel(vid, val)
68      */
69      BoolVar bool_expr_val(int vid, int int_rel, int val);
70
71      /**
72       Return the expression int_rel(vid1, vid2)
73      */
74      BoolVar bool_expr_var(int vid1, int int_rel, int vid2);
75
76  public:
77      /**
78       Default constructor
79       */
80      WSpace();
81
82      //=========================
83      //= Variables and domains =
84      //=========================
85
86      /**
87       Add an IntVar to the WSpace ranging from min to max.
88       In practice, push a new IntVar at the end of the vector int_vars.
89       Return the index of the IntVar in int_vars
90       */
91      int add_intVar(int min, int max);
92
```

```
93      /**
94       Add an IntVar to the WSpace with domain dom of size s.
95       In practice, push a new IntVar at the end of the vector int_vars.
96       Return the index of the IntVar in int_vars
97       */
98      int add_intVarWithDom(int s, int* dom);
99
100     /**
101      Add n IntVars to the WSpace ranging from min to max.
102      In practice, push n new IntVars at the end of the vector int_vars.
103      Return the indices of the IntVars in int_vars.
104      */
105     int* add_intVarArray(int n, int min, int max);
106
107     /**
108      Add n IntVars to the WSpace with domain dom of size s.
109      In practice, push n new IntVars at the end of the vector int_vars.
110      Return the indices of the IntVars in int_vars.
111      */
112     int* add_intVarArrayWithDom(int n, int s, int* dom);
113
114     /**
115     Define which variables are to be the solution so they can be accessed
116     to add a constraint with bab
117     */
118     int* set_as_solution_variables(int n, int* vids);
119
120     /**
121      Return the number of IntVars in the space.
122      */
123     int nvars();
124
125     enum {
126         //Relations for BoolExpr
127         B_EQ,
128         B_NQ,
129         B_LE,
130         B_LQ,
131         B_GQ,
132         B_GR
133     };
134
135     /**
136      Add a BoolVar to the WSpace ranging from min to max.
137      In practice, push a new BoolVar at the end of the vector bool_vars.
138      Return the index of the BoolVar in bool_vars
139      */
140     int add_boolVar(int min, int max);
141
142     /**
143      Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid, val).
144      In practice, push a new BoolVar at the end of the vector bool_vars.
145      Return the index of the BoolVar in bool_vars
146      */
147     int add_boolVar_expr_val(int vid, int int_rel, int val);
148
149     /**
150      Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid1, vid2).
151      In practice, push a new BoolVar at the end of the vector bool_vars.
152      Return the index of the BoolVar in bool_vars
153      */
154     int add_boolVar_expr_var(int vid1, int int_rel, int vid2);
155
156     //=======================
157     //= Posting constraints =
158     //=======================
159
160     //=== INTVARS ===
161
162     /**
163      Post a relation constraint between the IntVar denoted by vid and the val.
```

```
164      */
165     void cst_val_rel(int vid, int rel_type, int val);
166
167     /**
168      Post a relation constraint between the IntVars denoted by vid1 and vid2.
169     */
170     void cst_var_rel(int vid1, int rel_type, int vid2);
171
172     /**
173      Post a relation constraint between the n IntVars denoted by vids and the val.
174     */
175     void cst_arr_val_rel(int n, int* vids, int rel_type, int val);
176
177     /**
178      Post a relation constraint between the n IntVars denoted by vids and
179      the the IntVar vid.
180     */
181     void cst_arr_var_rel(int n, int* vids, int rel_type, int vid);
182
183     /**
184      Post a relation constraint between the n IntVars denoted by vids.
185     */
186     void cst_arr_rel(int n, int* vids, int rel_type);
187
188     /**
189      Post a lexicographic relation constraint between the n1 IntVars denoted by vids1 and
190      the n2 IntVars denoted by vids2.
191     */
192     void cst_arr_arr_rel(int n1, int* vids1, int rel_type, int n2, int* vids2);
193
194     /**
195      Post the constraint that the n IntVars denoted by vids are distinct
196      */
197     void cst_distinct(int n, int* vids);
198
199     /**
200      Post the linear constraint [c]*[vids] rel val.
201      */
202     void cst_val_linear(int n, int* c, int* vids, int rel_type, int val);
203
204     /**
205      Post the linear constraint [c]*[vids] rel_type vid.
206     */
207     void cst_var_linear(int n, int* c, int* vids, int rel_type, int vid);
208
209     /**
210      Post the constraint that |vid1| = vid2.
211      */
212     void cst_abs(int vid1, int vid2);
213
214     /**
215      Post the constraint that dom(vid) = d, where d is a set of size n.
216      */
217     void cst_dom(int vid, int n, int* d);
218
219     /**
220      Post the constraint that vid is included in {vids[0], ..., vids[n-1]}
221      */
222     void cst_member(int n, int* vids, int vid);
223
224     /**
225      Post the constraint that vid1 / vid2 = vid3.
226      */
227     void cst_div(int vid1, int vid2, int vid3);
228
229     /**
230      Post the constraint that vid1 % vid2 = vid3.
231      */
232     void cst_mod(int vid1, int vid2, int vid3);
233
234     /**
```

```
235      Post the constraint that vid1 / vid2 = vid3
236      and vid1 % vid2 = div4
237      */
238     void cst_divmod(int vid1, int vid2, int vid3, int vid4);
239
240     /**
241      Post the constraint that min(vid1, vid2) = vid3.
242      */
243     void cst_min(int vid1, int vid2, int vid3);
244
245     /**
246      Post the constraint that vid = min(vids).
247      */
248     void cst_arr_min(int n, int* vids, int vid);
249
250     /**
251      Post the constraint that vid = argmin(vids).
252      */
253     void cst_argmin(int n, int* vids, int vid);
254
255     /**
256      Post the constraint that max(vid1, vid2) = vid3.
257      */
258     void cst_max(int vid1, int vid2, int vid3);
259
260     /**
261      Post the constraint that vid = max(vids).
262      */
263     void cst_arr_max(int n, int* vids, int vid);
264
265     /**
266      Post the constraint that vid = argmax(vids).
267      */
268     void cst_argmax(int n, int* vids, int vid);
269
270     /**
271      Post the constraint that vid1 * vid2 = vid3.
272      */
273     void cst_mult(int vid1, int vid2, int vid3);
274
275     /**
276      Post the constraint that sqr(vid1) = vid2.
277      */
278     void cst_sqr(int vid1, int vid2);
279
280     /**
281      Post the constraint that sqrt(vid1) = vid2.
282      */
283     void cst_sqrt(int vid1, int vid2);
284
285     /**
286      Post the constraint that pow(vid1, n) = vid2.
287      */
288     void cst_pow(int vid1, int n, int vid2);
289
290     /**
291      Post the constraint that nroot(vid1, n) = vid2.
292      */
293     void cst_nroot(int vid1, int n, int vid2);
294
295     /**
296      Post the constraint that vid = sum(vids).
297      */
298     void cst_sum(int vid, int n, int* vids);
299
300     /**
301      Post the constraint that the number of variables in vids equal to val1
302      has relation rel_type with val2.
303      */
304     void cst_count_val_val(int n, int* vids, int val1, int rel_type, int val2);
305
```

```
    /**
     Post the constraint that the number of variables in vids equal to val
     has relation rel_type with vid.
     */
    void cst_count_val_var(int n, int* vids, int val, int rel_type, int vid);

    /**
     Post the constraint that the number of variables in vids equal to vid
     has relation rel_type with val.
     */
    void cst_count_var_val(int n, int* vids, int vid, int rel_type, int val);

    /**
     Post the constraint that the number of variables in vids equal to vid1
     has relation rel_type with vid2.
     */
    void cst_count_var_var(int n, int* vids, int vid1, int rel_type, int vid2);

    /**
    Post the constraint that the number of variables in vids in the set set
    has relation rel_type with vid2
    */
    void cst_count_var_set_val(int n, int*vids, int s, int* set, int rel_type, int val);

    /**
    Post the constraint that the number of variables in vids where
    vars[i] = c[i] and c is an array of integers has rel_type to val
    */
    void cst_count_array_val(int n, int*vids, int* c, int rel_type, int val);

    /**
     Post the constraint the number of distinct values in the n variables denoted by vids
     has the given rel_type relation with the variable vid.
     */
    void cst_nvalues(int n, int* vids, int rel_type, int vid);

    /**
     Post the constraint that values of vids1 are the edges of an hamiltonian circuit in
     the graph formed by the n variables in vids1, vids2 are the costs of these edges
     described by c, and vid is the total cost of the circuit, i.e. sum(vids2).
     */
    void cst_circuit(int n, int* c, int* vids1, int* vids2, int vid);

    /**
    Post the constraint that if there exists j (0 <= j < |x|) such that x[j] = t,
    then there must exist i with i < j such that x[i] = s
    */
    void cst_precede(int n, int* vids, int t, int u);

    //=== BOOLVARS ===

    /**
     Post the constraint that vid1 bool_op vid2 = val.
     */
    void cst_boolop_val(int vid1, int bool_op, int vid2, int val);

    /**
     Post the constraint that vid1 bool_op vid2 = vid3.
     */
    void cst_boolop_var(int vid1, int bool_op, int vid2, int vid3);

    /**
     Post a relation constraint between vid and val.
     */
    void cst_boolrel_val(int vid, int rel_type, int val);

    /**
     Post a relation constraint between vid1 and vid2.
     */
    void cst_boolrel_var(int vid1, int rel_type, int vid2);

```

```
377     //====================================
378     //Branch and bound constraint function =
379     //====================================
380 
381     /**
382     Constrain method for BAB search
383     This is called everytime the solver finds a solution
384     */
385     virtual void constrain(const Space& _b);
386 
387     //==========================
388     //= Exploration strategies =
389     //==========================
390 
391     /**
392      Post a branching strategy on the n IntVars in vids, with strategies denoted by
393      var_strategy and val_strategy.
394      */
395     void branch(int n, int* vids, int var_strategy, int val_strategy);
396 
397     /**
398      Post a branching strategy on the n BoolVars in vids, with strategies denoted by
399      var_strategy and val_strategy.
400      */
401     void branch_b(int n, int* vids, int var_strategy, int val_strategy);
402 
403     //==================
404     //= Search support =
405     //==================
406 
407     void cost(int vid);
408 
409     virtual IntVar cost(void) const;
410 
411     WSpace(WSpace& s);
412 
413     virtual Space* copy(void);
414 
415     //======================
416     //= Getting solutions =
417     //======================
418 
419     /**
420      Return the current values of the variable denoted by vid.
421      */
422     int value(int vid);
423 
424     /**
425      Return the current values of the n variables denoted by vids.
426      */
427     int* values(int n, int* vids);
428 
429     //======================
430     //= Printing solutions =
431     //======================
432 
433     /**
434      Print the n variables denoted by vids.
435      */
436     void print(int n, int* vids);
437 };
438 
439 //==================
440 //= Search options =
441 //==================
442 
443 class WTimeStop {
444 
445     protected:
446         Gecode::Search::TimeStop stop;
447         Gecode::Search::TimeStop* stop_ptr;
```

```
448
449     public:
450          WTimeStop(int maxTime);
451          ~WTimeStop();
452
453          void reset();
454          TimeStop getStop();
455          TimeStop* getStopPtr();
456 };
457
458 class WSearchOptions {
459
460 protected:
461     Gecode::Search::Options opts;
462
463 public:‘
464     WSearchOptions();
465     ~WSearchOptions();
466
467
468     /**
469      getter for the opts field
470     */
471     Options getOpts();
472
473     /**
474      set the number of threads to use for parallel search
475     */
476     int setNbThreads(int nThreads);
477
478     /**
479      Set the time stopping mechanism that is to be used during the search to a certain
480      duration in ms
481     */
482     void setTimeStop(WTimeStop* timestop);
483 ;}
484
485
486
487 //==================
488 //= Search engine =
489 //==================
490
491 class WbabEngine { // new version
492 protected:
493     BAB<WSpace>* bab;
494 public:
495     WbabEngine(WSpace* sp, Options opts);
496     ~WbabEngine();
497
498     /**
499      Search the next solution for this search engine.
500      */
501     WSpace* next();
502
503     int stopped();
504 };
505
506 class WdfsEngine { // new version
507 protected:
508     DFS<WSpace>* dfs;
509 public:
510     WdfsEngine(WSpace* sp, Options opts);
511     ~WdfsEngine();
512
513     /**
514      Search the next solution for this search engine.
515      */
516     WSpace* next();
517
518     int stopped();
```

```
};

#endif
```

B.1.2 space_wrapper.cpp

```
#include "headers/space_wrapper.hpp"
#include <iostream>

using namespace Gecode;
using namespace Gecode::Int;
using namespace std;

/**
 Default constructor
 */
WSpace::WSpace() {
    i_size = 0;
    b_size = 0;
}

//=======================
//= Variables from idx =
//=======================

/**
 Return the IntVar contained in int_vars at index vid
 */
IntVar WSpace::get_int_var(int vid) {
    return int_vars.at(vid);
}

/**
 Return the BoolVar contained in bool_vars at index vid
 */
BoolVar WSpace::get_bool_var(int vid) {
    return bool_vars.at(vid);
}

//====================
//= Args for methods =
//====================

/**
 Return an IntVarArgs of size n, containing the n IntVars contained in
 int_vars at indices vids.
 */
IntVarArgs WSpace::int_var_args(int n, int* vids) {
    IntVarArgs x(n);
    for(int i = 0; i < n; i++)
        x[i] = get_int_var(vids[i]);
    return x;
}

/**
 Return an BoolVarArgs of size n, containing the n BoolVars contained in
 bool_vars at indices vids.
 */
BoolVarArgs WSpace::bool_var_args(int n, int* vids) {
    BoolVarArgs x(n);
    for(int i = 0; i < n; i++)
        x[i] = get_bool_var(vids[i]);
    return x;
}

/**
 Return an IntArgs of size n, containing the n values in vals
*/
IntArgs WSpace::int_args(int n, int* vals) {
    IntArgs c(n);
```

```
65      for(int i = 0; i < n; i++)
66          c[i] = vals[i];
67      return c;
68  }
69
70  /**
71   Return the expression int_rel(vid, val)
72  */
73  BoolVar WSpace::bool_expr_val(int vid, int int_rel, int val) {
74      switch(int_rel) {
75          case B_EQ: return expr(*this, get_int_var(vid) == val);
76          case B_NQ: return expr(*this, get_int_var(vid) != val);
77          case B_LE: return expr(*this, get_int_var(vid) < val);
78          case B_LQ: return expr(*this, get_int_var(vid) <= val);
79          case B_GQ: return expr(*this, get_int_var(vid) >= val);
80          case B_GR: return expr(*this, get_int_var(vid) > val);
81          default:
82              cout << "Wrong expression type in BoolVar creation." << endl;
83              return BoolVar();
84      }
85  }
86
87  /**
88   Return the expression int_rel(vid1, vid2)
89  */
90  BoolVar WSpace::bool_expr_var(int vid1, int int_rel, int vid2) {
91      switch(int_rel) {
92          case B_EQ: return expr(*this, get_int_var(vid1) == get_int_var(vid2));
93          case B_NQ: return expr(*this, get_int_var(vid1) != get_int_var(vid2));
94          case B_LE: return expr(*this, get_int_var(vid1) < get_int_var(vid2));
95          case B_LQ: return expr(*this, get_int_var(vid1) <= get_int_var(vid2));
96          case B_GQ: return expr(*this, get_int_var(vid1) >= get_int_var(vid2));
97          case B_GR: return expr(*this, get_int_var(vid1) > get_int_var(vid2));
98          default:
99              cout << "Wrong expression type in BoolVar creation." << endl;
100             return BoolVar();
101     }
102 }
103
104
105 //=========================
106 //= Variables and domains =
107 //=========================
108
109 /**
110  Add an IntVar to the WSpace ranging from min to max.
111  In practice, push a new IntVar at the end of the vector int_vars.
112  Return the index of the IntVar in int_vars
113  */
114 int WSpace::add_intVar(int min, int max) {
115     int_vars.push_back(IntVar(*this, min, max));
116     return i_size++;
117 }
118
119 /**
120  Add an IntVar to the WSpace with domain dom of size s.
121  In practice, push a new IntVar at the end of the vector int_vars.
122  Return the index of the IntVar in int_vars
123  */
124 int WSpace::add_intVarWithDom(int s, int* dom) {
125     int_vars.push_back(IntVar(*this, IntSet(dom, s)));
126     return i_size++;
127 }
128
129 /**
130  Add n IntVars to the WSpace ranging from min to max.
131  In practice, push n new IntVars at the end of the vector int_vars.
132  Return the indices of the IntVars in int_vars.
133  */
134 int* WSpace::add_intVarArray(int n, int min, int max) {
135     int* vids = new int[n];
```

```
136     for(int i = 0; i < n; i++)
137         vids[i] = this->add_intVar(min, max);
138     return vids;
139 }
140 
141 /**
142  Add n IntVars to the WSpace with domain dom of size s.
143  In practice, push n new IntVars at the end of the vector int_vars.
144  Return the indices of the IntVars in int_vars.
145  */
146 int* WSpace::add_intVarArrayWithDom(int n, int s, int* dom) {
147     int* vids = new int[n];
148     for(int i = 0; i < n; i++)
149         vids[i] = this->add_intVarWithDom(s, dom);
150     return vids;
151 }
152 
153 /**
154  Define which variables are to be the solution so they can be accessed to
155  add a constraint with bab
156 */
157 int* WSpace::set_as_solution_variables(int n, int* vids){
158     solution_variable_indexes = vids;
159     return solution_variable_indexes;
160 }
161 
162 /**
163  Return the number of IntVars in the space.
164  */
165 int WSpace::nvars() {
166     return i_size;
167 }
168 
169 /**
170  Add a BoolVar to the WSpace ranging from min to max.
171  In practice, push a new BoolVar at the end of the vector bool_vars.
172  Return the index of the BoolVar in bool_vars
173  */
174 int WSpace::add_boolVar(int min, int max) {
175     bool_vars.push_back(BoolVar(*this, min, max));
176     return b_size++;
177 }
178 
179 /**
180  Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid, val).
181  In practice, push a new BoolVar at the end of the vector bool_vars.
182  Return the index of the BoolVar in bool_vars
183  */
184 int WSpace::add_boolVar_expr_val(int vid, int int_rel, int val) {
185     bool_vars.push_back(bool_expr_val(vid, int_rel, val));
186     return b_size++;
187 }
188 
189 /**
190  Add a BoolVar to the WSpace corresponding to the evaluation of int_rel(vid1, vid2).
191  In practice, push a new BoolVar at the end of the vector bool_vars.
192  Return the index of the BoolVar in bool_vars
193  */
194 int WSpace::add_boolVar_expr_var(int vid1, int int_rel, int vid2) {
195     bool_vars.push_back(bool_expr_var(vid1, int_rel, vid2));
196     return b_size++;
197 }
198 
199 
200 //=======================
201 //= Posting constraints =
202 //=======================
203 
204 //=== INTVAR ===
205 
206 /**
```

```
207  Post a relation constraint between the IntVar denoted by vid and the val.
208  */
209 void WSpace::cst_val_rel(int vid, int rel_type, int val) {
210     rel(*this, get_int_var(vid), (IntRelType) rel_type, val);
211 }
212
213 /**
214  Post a relation constraint between the IntVars denoted by vid1 and vid2.
215 */
216 void WSpace::cst_var_rel(int vid1, int rel_type, int vid2) {
217     rel(*this, get_int_var(vid1), (IntRelType) rel_type, get_int_var(vid2));
218 }
219
220 /**
221  Post a relation constraint between the n IntVars denoted by vids and the val.
222 */
223 void WSpace::cst_arr_val_rel(int n, int* vids, int rel_type, int val) {
224     rel(*this, int_var_args(n, vids), (IntRelType) rel_type, val);
225 }
226
227 /**
228  Post a relation constraint between the n IntVars denoted by vids and the the IntVar vid.
229 */
230 void WSpace::cst_arr_var_rel(int n, int* vids, int rel_type, int vid) {
231     rel(*this, int_var_args(n, vids), (IntRelType) rel_type, get_int_var(vid));
232 }
233
234 /**
235  Post a relation constraint between the n IntVars denoted by vids.
236 */
237 void WSpace::cst_arr_rel(int n, int* vids, int rel_type) {
238     rel(*this, int_var_args(n, vids), (IntRelType) rel_type);
239 }
240
241 /**
242  Post a lexicographic relation constraint between the n1 IntVars denoted by vids1 and
243  the n2 IntVars denoted by vids2.
244 */
245 void WSpace::cst_arr_arr_rel(int n1, int* vids1, int rel_type, int n2, int* vids2) {
246     rel(*this, int_var_args(n1, vids1), (IntRelType) rel_type, int_var_args(n2, vids2));
247 }
248
249 /**
250  Post the constraint that all IntVars denoted by vids are distinct
251  */
252 void WSpace::cst_distinct(int n, int* vids) {
253     distinct(*this, int_var_args(n, vids));
254 }
255
256 /**
257  Post the linear constraint [c]*[vids] rel_type val.
258  */
259 void WSpace::cst_val_linear(int n, int* c, int* vids, int rel_type, int val) {
260     linear(*this, int_args(n, c), int_var_args(n, vids), (IntRelType) rel_type, val);
261 }
262
263 /**
264  Post the linear constraint [c]*[vids] rel_type vid.
265 */
266 void WSpace::cst_var_linear(int n, int* c, int* vids, int rel_type, int vid) {
267     linear(*this, int_args(n, c), int_var_args(n, vids), (IntRelType) rel_type,
268         get_int_var(vid));
269 }
270
271 /**
272  Post the constraint that |vid1| = vid2.
273  */
274 void WSpace::cst_abs(int vid1, int vid2) {
275     abs(*this, get_int_var(vid1), get_int_var(vid2));
276 }
277
```

```
278 /**
279  Post the constaraint that dom(vid) = d.
280  */
281 void WSpace::cst_dom(int vid, int n, int* d) {
282     dom(*this, get_int_var(vid), IntSet(d, n));
283 }
284
285 /**
286  Post the constraint that vid is included in {vids[0], ..., vids[n-1]}
287 */
288 void WSpace::cst_member(int n, int* vids, int vid) {
289     member(*this, int_var_args(n, vids), get_int_var(vid));
290 }
291
292 /**
293  Post the constraint that vid1 / vid2 = vid3.
294  */
295 void WSpace::cst_div(int vid1, int vid2, int vid3) {
296     div(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
297 }
298
299 /**
300  Post the constraint that vid1 % vid2 = vid3.
301  */
302 void WSpace::cst_mod(int vid1, int vid2, int vid3) {
303     mod(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
304 }
305
306 /**
307  Post the constraint that vid1 / vid2 = vid3
308  and vid1 % vid2 = div4
309  */
310 void WSpace::cst_divmod(int vid1, int vid2, int vid3, int vid4) {
311     divmod(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3),
312     get_int_var(vid4));
313 }
314
315 /**
316  Post the constraint that min(vid1, vid2) = vid3.
317  */
318 void WSpace::cst_min(int vid1, int vid2, int vid3) {
319     Gecode::min(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
320 }
321
322 /**
323  Post the constraint that vid = min(vids).
324  */
325 void WSpace::cst_arr_min(int n, int* vids, int vid) {
326     Gecode::min(*this, int_var_args(n, vids), get_int_var(vid));
327 }
328
329 /**
330  Post the constraint that vid = argmin(vids).
331  */
332 void WSpace::cst_argmin(int n, int* vids, int vid) {
333     Gecode::argmin(*this, int_var_args(n, vids), get_int_var(vid));
334 }
335
336 /**
337  Post the constraint that max(vid1, vid2) = vid3.
338  */
339 void WSpace::cst_max(int vid1, int vid2, int vid3) {
340     Gecode::max(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
341 }
342
343 /**
344  Post the constraint that vid = max(vids).
345  */
346 void WSpace::cst_arr_max(int n, int* vids, int vid) {
347     Gecode::max(*this, int_var_args(n, vids), get_int_var(vid));
348 }
```

```
349
350 /**
351  Post the constraint that vid = argmax(vids).
352  */
353 void WSpace::cst_argmax(int n, int* vids, int vid) {
354     Gecode::argmax(*this, int_var_args(n, vids), get_int_var(vid));
355 }
356
357 /**
358  Post the constraint that vid1 * vid2 = vid3.
359  */
360 void WSpace::cst_mult(int vid1, int vid2, int vid3) {
361     mult(*this, get_int_var(vid1), get_int_var(vid2), get_int_var(vid3));
362 }
363
364 /**
365  Post the constraint that sqr(vid1) = vid2.
366  */
367 void WSpace::cst_sqr(int vid1, int vid2) {
368     sqr(*this, get_int_var(vid1), get_int_var(vid2));
369 }
370
371 /**
372  Post the constraint that sqrt(vid1) = vid2.
373  */
374 void WSpace::cst_sqrt(int vid1, int vid2) {
375     Gecode::sqrt(*this, get_int_var(vid1), get_int_var(vid2));
376 }
377
378 /**
379  Post the constraint that pow(vid1, n) = vid2.
380  */
381 void WSpace::cst_pow(int vid1, int n, int vid2) {
382     Gecode::pow(*this, get_int_var(vid1), n, get_int_var(vid2));
383 }
384
385 /**
386  Post the constraint that nroot(vid1, n) = vid2.
387  */
388 void WSpace::cst_nroot(int vid1, int n, int vid2) {
389     nroot(*this, get_int_var(vid1), n, get_int_var(vid2));
390 }
391
392 /**
393  Post the constraint that vid = sum(vids).
394  */
395 void WSpace::cst_sum(int vid, int n, int* vids) {
396     rel(*this, get_int_var(vid), IRT_EQ, expr(*this, sum(int_var_args(n, vids))));
397 }
398
399 /**
400  Post the constraint that the number of variables in vids equal to val1 has
401  relation rel_type with val2.
402  */
403 void WSpace::cst_count_val_val(int n, int* vids, int val1, int rel_type, int val2) {
404     count(*this, int_var_args(n, vids), val1, (IntRelType) rel_type, val2);
405 }
406
407 /**
408  Post the constraint that the number of variables in vids equal to val has relation rel_type
409  with vid.
410  */
411 void WSpace::cst_count_val_var(int n, int* vids, int val, int rel_type, int vid) {
412     count(*this, int_var_args(n, vids), val, (IntRelType) rel_type, get_int_var(vid));
413 }
414
415 /**
416  Post the constraint that the number of variables in vids equal to vid has relation rel_type
417  with val.
418  */
419 void WSpace::cst_count_var_val(int n, int* vids, int vid, int rel_type, int val) {
```

```
420     count(*this, int_var_args(n, vids), get_int_var(vid), (IntRelType) rel_type, val);
421 }
422
423 /**
424  Post the constraint that the number of variables in vids equal to vid1 has
425  relation rel_type with vid2.
426  */
427 void WSpace::cst_count_var_var(int n, int* vids, int vid1, int rel_type, int vid2) {
428     count(*this, int_var_args(n, vids), vid1, (IntRelType) rel_type, get_int_var(vid2));
429 }
430
431 /**
432  Post the constraint that the number of variables in vids in the set set has
433  relation rel_type with val
434  */
435 void WSpace::cst_count_var_set_val(int n, int*vids, int s, int* set, int rel_type, int val){
436     count(*this, int_var_args(n, vids), IntSet(set, s), (IntRelType) rel_type, val);
437 }
438
439 /**
440  Post the constraint that the number of variables in vids where vars[i] = c[i] and c is an array of integers
441  has rel_type to val
442  */
443 void WSpace::cst_count_array_val(int n, int*vids, int* c, int rel_type, int val){
444     count(*this, int_var_args(n, vids), int_args(n, c), (IntRelType) rel_type, val);
445 }
446
447 /**
448  Post the constraint that the number of distinct values in the n variables denoted by vids
449  has the given rel_type relation with the variable vid.
450  */
451 void WSpace::cst_nvalues(int n, int* vids, int rel_type, int vid) {
452     nvalues(*this, int_var_args(n, vids), (IntRelType) rel_type, get_int_var(vid));
453 }
454
455 /**
456  Post the constraint that values of vids1 are the edges of an hamiltonian circuit in
457  the graph formed by the n variables in vids1, vids2 are the costs of these edges described
458  by c, and vid is the total cost of the circuit, i.e. sum(vids2).
459  */
460 void WSpace::cst_circuit(int n, int* c, int* vids1, int* vids2, int vid) {
461     circuit(*this, int_args(n*n, c), int_var_args(n, vids1), int_var_args(n, vids2),
462         get_int_var(vid));
463 }
464
465 /**
466  Post the constraint that if there exists j (0 <= j < |x|) such that x[j] = u,
467  then there must exist i with i < j such that x[i] = s
468 */
469 void WSpace::cst_precede(int n, int* vids, int s, int u){
470     precede(*this, int_var_args(n, vids), s, u);
471 }
472
473
474 //=== BOOLVAR ===
475
476 /**
477  Post the constraint that vid1 bool_op vid2 = val.
478  */
479 void WSpace::cst_boolop_val(int vid1, int bool_op, int vid2, int val) {
480     rel(*this, get_bool_var(vid1), (BoolOpType) bool_op, get_bool_var(vid2), val);
481 }
482
483 /**
484  Post the constraint that vid1 bool_op vid2 = vid3.
485  */
486 void WSpace::cst_boolop_var(int vid1, int bool_op, int vid2, int vid3) {
487     rel(*this, get_bool_var(vid1), (BoolOpType) bool_op, get_bool_var(vid2),
488     get_bool_var(vid3));
489 }
490
```

```
491 /**
492  Post a relation constraint between vid and val.
493  */
494 void WSpace::cst_boolrel_val(int vid, int rel_type, int val) {
495     rel(*this, get_bool_var(vid), (IntRelType) rel_type, val);
496 }
497
498 /**
499  Post a relation constraint between vid1 and vid2.
500  */
501 void WSpace::cst_boolrel_var(int vid1, int rel_type, int vid2) {
502     rel(*this, get_bool_var(vid1), (IntRelType) rel_type, get_bool_var(vid2));
503 }
504
505 //====================================
506 //Branch and bound constraint function =
507 //====================================
508
509 /**
510  Constrain method for BAB search
511  This is called everytime the solver finds a solution
512  This is a virtual method as declared in space_wrapper.h
513 */
514 void WSpace::constrain(const Space& _b) {
515     // right now, do nothing
516 }
517
518 //============================
519 //= Exploration strategies =
520 //============================
521
522 /**
523  Post a branching strategy on the variables in vids, with strategies denoted
524  by var_strategy and val_strategy.
525  var_strategy:
526     - 0 :  INT_VAR_SIZE_MIN()
527     - 1 :  INT_VAR_SIZE_RND(r)
528     - 2 :  INT_VAR_DEGREE_MAX()
529  val_strategy:
530     - 0 :  INT_VAL_MIN()
531     - 1 :  INT_VAL_RND(r)
532     - 2 :  INT_VAL_SPLIT_MIN()
533     - 3 :  INT_VAL_SPLIT_MAX()
534     - 4 :  INT_VAL_MED()
535  */
536 void WSpace::branch(int n, int* vids, int var_strategy, int val_strategy) {
537     IntVarBranch var_strat;
538     IntValBranch val_strat;
539
540     Rnd r1(1U);
541     Rnd r2(3U);
542
543     //determine the variable strategy
544     if(var_strategy == 0){
545         var_strat = INT_VAR_SIZE_MIN();
546     }
547     else if(var_strategy == 1){
548         var_strat = INT_VAR_RND(r1);
549     }
550     else if(var_strategy == 2){
551         var_strat = INT_VAR_DEGREE_MAX();
552     }
553
554     //determine the value strategy
555     if(val_strategy == 0){
556         val_strat = INT_VAL_MIN();
557     }
558     if(val_strategy == 1){
559         val_strat = INT_VAL_RND(r2);
560     }
561     if(val_strategy == 2){
```

```
562         val_strat = INT_VAL_SPLIT_MIN();
563     }
564     if(val_strategy == 3){
565         val_strat = INT_VAL_SPLIT_MAX();
566     }
567     if(val_strategy == 4){
568         val_strat = INT_VAL_MED();
569     }
570 
571     Gecode::branch(*this, int_var_args(n, vids), var_strat, val_strat);
572 }
573 
574 
575 /**
576  Post a branching strategy on the n BoolVars in vids, with strategies
577  denoted by var_strategy and val_strategy.
578  */
579 void WSpace::branch_b(int n, int* vids, int var_strategy, int val_strategy) {
580     Gecode::branch(*this, bool_var_args(n, vids), BOOL_VAR_NONE(),
581         BOOL_VAL_MIN()); //default for now
582 }
583 
584 //==================
585 //= Search support =
586 //==================
587 
588 /**
589  Define which variable, denoted by vid, will be considered as the cost.
590  */
591 void WSpace::cost(int vid) {
592     cost_id = vid;
593 }
594 
595 IntVar WSpace::cost(void) const {
596     return int_vars.at(cost_id);
597 }
598 
599 WSpace::WSpace(WSpace& s): IntMinimizeSpace(s), int_vars(s.i_size), bool_vars(s.b_size),
600     i_size(s.i_size), b_size(s.b_size), cost_id(s.cost_id) {
601     //IntVars update
602     vector<IntVar>::iterator itd, its;
603     for(itd = int_vars.begin(), its = s.int_vars.begin(); itd != int_vars.end(); ++itd, ++
    its)
604         itd->update(*this, *its);
605 
606     //BoolVars update
607     vector<BoolVar>::iterator btd, bts;
608     for(btd = bool_vars.begin(), bts = s.bool_vars.begin(); btd != bool_vars.end(); ++btd,
    ++bts)
609         btd->update(*this, *bts);
610 }
611 
612 Space* WSpace::copy(void) {
613     return new WSpace(*this);
614 }
615 
616 //======================
617 //= Getting solutions =
618 //======================
619 
620 /**
621  Return the current values of the variable denoted by vid.
622  */
623 int WSpace::value(int vid) {
624     return get_int_var(vid).val();
625 }
626 
627 /**
628  Return the current values of the n variables denoted by vids.
629  */
630 int* WSpace::values(int n, int* vids) {
```

```
631     int* vals = new int[n];
632     for(int i = 0; i < n; i++)
633         vals[i] = get_int_var(vids[i]).val();
634     return vals;
635 }
636 
637 //=======================
638 //= Printing solutions =
639 //=======================
640 
641 void WSpace::print(int n, int* vids) {
642     std::cout << "{";
643     for(int i = 0; i < n; i++) {
644         std::cout << get_int_var(vids[i]);
645         if(i < n - 1) std::cout << ", ";
646     }
647     std::cout << "}" << std::endl;
648 }
649 
650 //============================
651 //= Search options managment =
652 //============================
653 
654 // === TIME STOP OBJECT ===
655 
656 /**
657  Default constructor
658  */
659 WTimeStop::WTimeStop(int maxTime) :  stop(Gecode::Search::TimeStop(maxTime)) {
660     stop_ptr = &stop;
661 }
662 
663 WTimeStop::~WTimeStop(){
664 
665 }
666 
667 TimeStop WTimeStop::getStop(){
668     return stop;
669 }
670 
671 TimeStop* WTimeStop::getStopPtr(){
672     return stop_ptr;
673 }
674 
675 /**
676  Reset the time value of the time stop object
677  */
678 void WTimeStop::reset(){
679     stop.reset();
680 }
681 
682 // === OPTIONS OBJECT ===
683 
684 /**
685  Default constructor
686  */
687 WSearchOptions::WSearchOptions(){
688 
689 }
690 
691 WSearchOptions::~WSearchOptions(){
692 
693 }
694 
695 Options WSearchOptions::getOpts(){
696     return opts;
697 }
698 
699 /**
700  Set the number of threads to use for parallel search
701  */
```

```
int WSearchOptions::setNbThreads(int nThreads){
    opts.threads = nThreads;
    return opts.threads;
}

/**
 Set the time stopping mechanism that is to be used during the search to a
 certain duration in ms
 Takes a WTimeStop object as argument, and sets the WSearchOptions object's
 opts.stop field to the TimeStop pointer of the WTimeStop object
 */
void WSearchOptions::setTimeStop(WTimeStop* timestop){
    opts.stop = timestop->getStopPtr();
}

//=================
//= Search engine =
//=================

/*
 Branch and bound
 */
WbabEngine::WbabEngine(WSpace* sp, Options opts) {
    bab = new BAB<WSpace>(sp, opts);
}

WbabEngine::~WbabEngine() {
    delete bab;
}

/**
 Search the next solution for this search engine.
 */
WSpace* WbabEngine::next() {
    return bab->next();
}

/**
 Returns true if the search has been stopped by a search object
 */
int WbabEngine::stopped(){
    return bab->stopped();
}

/*
 Depth-first search
*/
WdfsEngine::WdfsEngine(WSpace* sp, Options opts) {
    dfs = new DFS<WSpace>(sp, opts);
}

WdfsEngine::~WdfsEngine() {
    delete dfs;
}

/**
 Search the next solution for this search engine.
 */
WSpace* WdfsEngine::next() {
    return dfs->next();
}

/**
 Returns true if the search has been stopped by a search object
 */
int WdfsEngine::stopped(){
    return dfs->stopped();
}
```

B.1.3 gecode_wrapper.hpp

```
#ifndef gecode_wrapper_hpp
#define gecode_wrapper_hpp

#include <stdlib.h>

#ifdef __cplusplus
extern "C" {
#endif

enum {
    IRT_EQ,
    IRT_NQ,
    IRT_LQ,
    IRT_LE,
    IRT_GQ,
    IRT_GR
};

enum {
    BOT_AND,
    BOT_OR,
    BOT_IMP,
    BOT_EQV,
    BOT_XOR
};

/**
 Wraps the WSpace constructor.
 */
void* computation_space();

/**
 Wraps the WSpace add_intVar method.
 */
int add_intVar(void* sp, int min, int max);

/**
 Wraps the WSpace add_intVarWithDom method.
 */
int add_intVarWithDom(void* sp, int s, int* dom);

/**
 Wraps the WSpace add_intVarArray method.
 */
int* add_intVarArray(void* sp, int n, int min, int max);

/**
 Wraps the WSpace add_intVarArrayWithDom method.
 */
int* add_intVarArrayWithDom(void* sp, int n, int s, int* dom);

/**
 Wraps the WSpace set_as_solution_variables method.
 */
int* set_solution_vars(void* sp, int n, int* vids);

/**
 Wraps the WSpace nvars method.
 */
int nvars(void* sp);

/**
 Wraps the WSpace add_boolVar method.
 */
int add_boolVar(void* sp, int min, int max);

/**
 Wraps the WSpace add_boolVar_expr_val method.
 */
```

```
70 int add_boolVar_expr_val(void* sp, int vid, int rel_type, int val);
71
72 /**
73  Wraps the WSpace add_boolVar_expr_var method.
74  */
75 int add_boolVar_expr_var(void* sp, int vid1, int rel_type, int vid2);
76
77 /**
78  Wraps the WSpace cst_val_rel method.
79  */
80 void val_rel(void* sp, int vid, int rel_type, int val);
81
82 /**
83  Wraps the WSpace cst_var_relr method.
84  */
85 void var_rel(void* sp, int vid1, int rel_type, int vid2);
86
87 /**
88  Wraps the WSpace cst_arr_val_rel method.
89  */
90 void arr_val_rel(void* sp, int n, int* vids, int rel_type, int val);
91
92 /**
93  Wraps the WSpace cst_arr_var_rel method.
94  */
95 void arr_var_rel(void* sp, int n, int* vids, int rel_type, int vid);
96
97 /**
98  Wraps the WSpace cst_arr_rel method.
99  */
100 void arr_rel(void* sp, int n, int* vids, int rel_type);
101
102 /**
103  Wraps the WSpace cst_arr_arr_rel method.
104  */
105 void arr_arr_rel(void* sp, int n1, int* vids1, int rel_type, int n2, int* vids2);
106
107 /**
108  Wraps the WSpace cst_distinct method.
109  */
110 void distinct(void* sp, int n, int* vids);
111
112 /**
113  Wraps the WSpace cst_val_linear method.
114  */
115 void val_linear(void* sp, int n, int* c, int* vids, int rel_type, int value);
116
117 /**
118  Wraps the WSpace cst_var_linear method.
119  */
120 void var_linear(void* sp, int n, int* c, int* vids, int rel_type, int vid);
121
122 /**
123  Wraps the WSpace cst_abs method.
124  */
125 void arithmetics_abs(void* sp, int vid1, int vid2);
126
127 /**
128  Wraps the WSpace acst_div method.
129  */
130 void arithmetics_div(void* sp, int vid1, int vid2, int vid3);
131
132 /**
133  Wraps the WSpace cst_var_mod method.
134  */
135 void arithmetics_mod(void* sp, int vid1, int vid2, int vid3);
136
137 /**
138  Wraps the WSpace cst_divmod method.
139  */
140 void arithmetics_divmod(void* sp, int vid1, int vid2, int vid3, int vid4);
```

```
141
142 /**
143  Wraps the WSpace cst_min method.
144  */
145 void arithmetics_min(void* sp, int vid1, int vid2, int vid3);
146
147 /**
148  Wraps the WSpace cst_arr_min method.
149  */
150 void arithmetics_arr_min(void* sp, int n, int* vids, int vid);
151
152 /**
153  Wraps the WSpace cst_argmin method.
154  */
155 void arithmetics_argmin(void* sp, int n, int* vids, int vid);
156
157 /**
158  Wraps the WSpace cst_max method.
159  */
160 void arithmetics_max(void* sp, int vid1, int vid2, int vid3);
161
162 /**
163  Wraps the WSpace cst_arr_max method.
164  */
165 void arithmetics_arr_max(void* sp, int n, int* vids, int vid);
166
167 /**
168  Wraps the WSpace cst_argmax method.
169  */
170 void arithmetics_argmax(void* sp, int n, int* vids, int vid);
171
172 /**
173  Wraps the WSpace cst_mult method.
174  */
175 void arithmetics_mult(void* sp, int vid1, int vid2, int vid3);
176
177 /**
178  Wraps the WSpace cst_sqr method.
179  */
180 void arithmetics_sqr(void* sp, int vid1, int vid2);
181
182 /**
183  Wraps the WSpace cst_sqrt method.
184  */
185 void arithmetics_sqrt(void* sp, int vid1, int vid2);
186
187 /**
188  Wraps the WSpace cst_pow method.
189  */
190 void arithmetics_pow(void* sp, int vid1, int n, int vid2);
191
192 /**
193  Wraps the WSpace cst_nroot method.
194  */
195 void arithmetics_nroot(void* sp, int vid1, int n, int vid2);
196
197 /**
198  Wraps the WSpace cst_dom method.
199  */
200 void set_dom(void* sp, int vid, int n, int* d);
201
202 /**
203  Wraps the WSpace cst_member method.
204  */
205 void set_member(void* sp, int n, int* vids, int vid);
206
207 /**
208  Wraps the WSpace cst_sum method.
209  */
210 void rel_sum(void* sp, int vid, int n, int* vids);
211
```

```
212 /**
213  Wraps the WSpace cst_count_val_val method.
214  */
215 void count_val_val(void* sp, int n, int* vids, int val1, int rel_type, int val2);
216
217 /**
218  Wraps the WSpace cst_count_val_var method.
219  */
220 void count_val_var(void* sp, int n, int* vids, int val, int rel_type, int vid);
221
222 /**
223  Wraps the WSpace cst_count_var_val method.
224  */
225 void count_var_val(void* sp, int n, int* vids, int vid, int rel_type, int val);
226
227 /**
228  Wraps the WSpace cst_count_var_var method.
229  */
230 void count_var_var(void* sp, int n, int* vids, int vid1, int rel_type, int vid2);
231
232 /**
233  Wraps the WSpace cst_count_var_set_val method.
234  */
235 void count_var_set_val(void*sp, int n, int* vids, int s, int* set, int rel_type, int val);
236
237 /**
238  Wraps the WSpace cst_count_array_val method.
239  */
240 void count_array_val(void* sp, int n, int* vids, int* c, int rel_type, int val);
241
242 /**
243  Wraps the WSpace cst_nvalues method.
244  */
245 void nvalues(void* sp, int n, int* vids, int rel_type, int vid);
246
247 /**
248  Wraps the WSpace cst_circuit method.
249  */
250 void circuit(void* sp, int n, int* c, int* vids1, int* vids2, int vid);
251
252 /**
253  Wraps the WSpace cst_precede method.
254  */
255 void precede(void* sp, int n, int* vids, int s, int u);
256
257 /**
258  Wraps the WSpace cst_boolop_val method.
259  */
260 void val_boolop(void* sp, int vid1, int bool_op, int vid2, int val);
261
262 /**
263  Wraps the WSpace cst_boolop_var method.
264  */
265 void var_boolop(void* sp, int vid1, int bool_op, int vid2, int vid3);
266
267 /**
268  Wraps the WSpace cst_boolrel_val method.
269  */
270 void val_boolrel(void* sp, int vid, int rel_type, int val);
271
272 /**
273  Wraps the WSpace cst_boolrel_var method.
274  */
275 void var_boolrel(void* sp, int vid1, int rel_type, int vid2);
276
277 /**
278  Wraps the WSpace branch method.
279  */
280 void branch(void* sp, int n, int* vids, int var_strategy, int val_strategy);
281
282 /**
```

```
283  Wraps the WSpace branch_b method.
284  */
285 void branch_b(void* sp, int n, int* vids, int var_strategy, int val_strategy);
286
287 /**
288  Wraps the WSpace cost method.
289  */
290 void cost(void* sp, int vid);
291
292 /**
293  Wraps the WTimeStop constructor.
294  */
295 void* new_time_stop(int maxTime);
296
297 /**
298  Wraps the WTimeStop reset method.
299  */
300 void reset_time_stop(void* tStop);
301
302 /**
303  Wraps the WSearchOptions constructor.
304  */
305 void* new_search_options();
306
307 /**
308  Wraps the WSearchOptions setNbThreads method.
309  */
310 int set_nb_threads(void* sOpts, int nThreads);
311
312 /**
313  Wraps the WSearchOptions setTimeStop method.
314  */
315 void* set_time_stop(void* sOpts, void* tStop);
316
317
318 /**
319  Wraps the WbabEngine constructor.
320  */
321 void* new_bab_engine(void* sp, void* opts);
322
323 /**
324  Wraps the WbabEngine next method.
325  */
326 void* bab_next(void* se);
327
328 /**
329  Wraps the WbabEngine stopped method.
330  */
331 int bab_stopped(void* se);
332
333 /**
334  Wraps the WdfsEngine constructor.
335  */
336 void* new_dfs_engine(void* sp, void* opts);
337
338 /**
339  Wraps the WdfsEngine next method.
340  */
341 void* dfs_next(void* se);
342
343 /**
344  Wraps the WdfsEngine stopped method.
345  */
346 int dfs_stopped(void* se);
347
348 /**
349  Wraps the WSpace destructor.
350  */
351 void release(void* sp);
352
353 /**
```

```
 Wraps the WSpace value method.
 */
int get_value(void* sp, int vid);

/**
 Wraps the WSpace values method.
 */
int* get_values(void* sp, int n, int* vids);

/**
 Wraps the WSpace print method.
 */
void print_vars(void* sp, int n, int* vids);

#ifdef __cplusplus
};
#endif
#endif
```

B.1.4 gecode_wrapper.cpp

```
#include "headers/gecode_wrapper.hpp"
#include "headers/space_wrapper.hpp"

/**
 Wraps the WSpace constructor.
 */
void* computation_space() {
    return (void*) new WSpace();
}

/**
 Wraps the WSpace add_intVar method.
 */
int add_intVar(void* sp, int min, int max) {
    return static_cast<WSpace*>(sp)->add_intVar(min, max);
}

/**
 Wraps the WSpace add_intVarWithDom method.
 */
int add_intVarWithDom(void* sp, int s, int* dom) {
    return static_cast<WSpace*>(sp)->add_intVarWithDom(s, dom);
}

/**
 Wraps the WSpace add_intVarArray method.
 */
int* add_intVarArray(void* sp, int n, int min, int max) {
    return static_cast<WSpace*>(sp)->add_intVarArray(n, min, max);
}

/**
 Wraps the WSpace add_intVarArrayWithDom method.
 */
int* add_intVarArrayWithDom(void* sp, int n, int s, int* dom) {
    return static_cast<WSpace*>(sp)->add_intVarArrayWithDom(n, s, dom);
}

/**
 Wraps the WSpace set_as_solution_variables method.
 */
int* set_solution_vars(void* sp, int n, int* vids){
    return static_cast<WSpace*>(sp)->set_as_solution_variables(n, vids);
}

/**
 Wraps the WSpace nvars method.
 */
int nvars(void* sp) {
```

```
50      return static_cast<WSpace*>(sp)->nvars();
51  }
52
53  /**
54   Wraps the WSpace add_boolVar method.
55   */
56  int add_boolVar(void* sp, int min, int max) {
57      return static_cast<WSpace*>(sp)->add_boolVar(min, max);
58  }
59
60  /**
61   Wraps the WSpace add_boolVar_expr_val method.
62   */
63  int add_boolVar_expr_val(void* sp, int vid, int rel_type, int val) {
64      return static_cast<WSpace*>(sp)->add_boolVar_expr_val(vid, rel_type, val);
65  }
66
67  /**
68   Wraps the WSpace add_boolVar_expr_var method.
69   */
70  int add_boolVar_expr_var(void* sp, int vid1, int rel_type, int vid2) {
71      return static_cast<WSpace*>(sp)->add_boolVar_expr_var(vid1, rel_type, vid2);
72  }
73
74  /**
75   Wraps the WSpace cst_val_rel method.
76   */
77  void val_rel(void* sp, int vid, int rel_type, int val) {
78      return static_cast<WSpace*>(sp)->cst_val_rel(vid, rel_type, val);
79  }
80
81  /**
82   Wraps the WSpace cst_var_relr method.
83   */
84  void var_rel(void* sp, int vid1, int rel_type, int vid2) {
85      return static_cast<WSpace*>(sp)->cst_var_rel(vid1, rel_type, vid2);
86  }
87
88  /**
89   Wraps the WSpace cst_arr_val_rel method.
90   */
91  void arr_val_rel(void* sp, int n, int* vids, int rel_type, int val) {
92      return static_cast<WSpace*>(sp)->cst_arr_val_rel(n, vids, rel_type, val);
93  }
94
95  /**
96   Wraps the WSpace cst_arr_var_rel method.
97   */
98  void arr_var_rel(void* sp, int n, int* vids, int rel_type, int vid) {
99      return static_cast<WSpace*>(sp)->cst_arr_var_rel(n, vids, rel_type, vid);
100 }
101
102 /**
103  Wraps the WSpace cst_arr_rel method.
104  */
105 void arr_rel(void* sp, int n, int* vids, int rel_type) {
106     return static_cast<WSpace*>(sp)->cst_arr_rel(n, vids, rel_type);
107 }
108
109 /**
110  Wraps the WSpace cst_arr_arr_rel method.
111  */
112 void arr_arr_rel(void* sp, int n1, int* vids1, int rel_type, int n2, int* vids2) {
113     return static_cast<WSpace*>(sp)->cst_arr_arr_rel(n1, vids1, rel_type, n2, vids2);
114 }
115
116 /**
117  Wraps the WSpace cst_distinct method.
118  */
119 void distinct(void* sp, int n, int* vids) {
120     return static_cast<WSpace*>(sp)->cst_distinct(n, vids);
```

```
}

/**
 Wraps the WSpace cst_val_linear method.
 */
void val_linear(void* sp, int n, int* c, int* vids, int rel_type, int value) {
    return static_cast<WSpace*>(sp)->cst_val_linear(n, c, vids, rel_type, value);
}

/**
 Wraps the WSpace cst_var_linear method.
 */
void var_linear(void* sp, int n, int* c, int* vids, int rel_type, int vid) {
    return static_cast<WSpace*>(sp)->cst_var_linear(n, c, vids, rel_type, vid);
}

/**
 Wraps the WSpace cst_abs method.
 */
void arithmetics_abs(void* sp, int vid1, int vid2) {
    return static_cast<WSpace*>(sp)->cst_abs(vid1, vid2);
}

/**
 Wraps the WSpace acst_div method.
 */
void arithmetics_div(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_div(vid1, vid2, vid3);
}

/**
 Wraps the WSpace cst_mod method.
 */
void arithmetics_mod(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_mod(vid1, vid2, vid3);
}

/**
 Wraps the WSpace cst_divmod method.
 */
void arithmetics_divmod(void* sp, int vid1, int vid2, int vid3, int vid4) {
    return static_cast<WSpace*>(sp)->cst_divmod(vid1, vid2, vid3, vid4);
}

/**
 Wraps the WSpace cst_min method.
 */
void arithmetics_min(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_min(vid1, vid2, vid3);
}

/**
 Wraps the WSpace cst_arr_min method.
 */
void arithmetics_arr_min(void* sp, int n, int* vids, int vid) {
    return static_cast<WSpace*>(sp)->cst_arr_min(n, vids, vid);
}

/**
 Wraps the WSpace cst_argmin method.
 */
void arithmetics_argmin(void* sp, int n, int* vids, int vid) {
    return static_cast<WSpace*>(sp)->cst_argmin(n, vids, vid);
}

/**
 Wraps the WSpace cst_max method.
 */
void arithmetics_max(void* sp, int vid1, int vid2, int vid3) {
    return static_cast<WSpace*>(sp)->cst_max(vid1, vid2, vid3);
}
```

```
192
193 /**
194  Wraps the WSpace cst_arr_max method.
195  */
196 void arithmetics_arr_max(void* sp, int n, int* vids, int vid) {
197     return static_cast<WSpace*>(sp)->cst_arr_max(n, vids, vid);
198 }
199
200 /**
201  Wraps the WSpace cst_argmax method.
202  */
203 void arithmetics_argmax(void* sp, int n, int* vids, int vid) {
204     return static_cast<WSpace*>(sp)->cst_argmax(n, vids, vid);
205 }
206
207 /**
208  Wraps the WSpace cst_mult method.
209  */
210 void arithmetics_mult(void* sp, int vid1, int vid2, int vid3) {
211     return static_cast<WSpace*>(sp)->cst_mult(vid1, vid2, vid3);
212 }
213
214 /**
215  Wraps the WSpace cst_sqr method.
216  */
217 void arithmetics_sqr(void* sp, int vid1, int vid2) {
218     return static_cast<WSpace*>(sp)->cst_sqr(vid1, vid2);
219 }
220
221 /**
222  Wraps the WSpace cst_sqrt method.
223  */
224 void arithmetics_sqrt(void* sp, int vid1, int vid2) {
225     return static_cast<WSpace*>(sp)->cst_sqrt(vid1, vid2);
226 }
227
228 /**
229  Wraps the WSpace cst_pow method.
230  */
231 void arithmetics_pow(void* sp, int vid1, int n, int vid2) {
232     return static_cast<WSpace*>(sp)->cst_pow(vid1, n, vid2);
233 }
234
235 /**
236  Wraps the WSpace cst_nroot method.
237  */
238 void arithmetics_nroot(void* sp, int vid1, int n, int vid2) {
239     return static_cast<WSpace*>(sp)->cst_nroot(vid1, n, vid2);
240 }
241
242 /**
243  Wraps the WSpace cst_dom method.
244  */
245 void set_dom(void* sp, int vid, int n, int* d) {
246     return static_cast<WSpace*>(sp)->cst_dom(vid, n, d);
247 }
248
249 /**
250  Wraps the WSpace cst_member method.
251  */
252 void set_member(void* sp, int n, int* vids, int vid) {
253     return static_cast<WSpace*>(sp)->cst_member(n, vids, vid);
254 }
255
256 /**
257  Wraps the WSpace cst_sum method.
258  */
259 void rel_sum(void* sp, int vid, int n, int* vids) {
260     return static_cast<WSpace*>(sp)->cst_sum(vid, n, vids);
261 }
262
```

```
263 /**
264  Wraps the WSpace cst_count_val_val method.
265  */
266 void count_val_val(void* sp, int n, int* vids, int val1, int rel_type, int val2) {
267     return static_cast<WSpace*>(sp)->cst_count_val_val(n, vids, val1, rel_type, val2);
268 }
269
270 /**
271  Wraps the WSpace cst_count_val_var method.
272  */
273 void count_val_var(void* sp, int n, int* vids, int val, int rel_type, int vid) {
274     return static_cast<WSpace*>(sp)->cst_count_val_var(n, vids, val, rel_type, vid);
275 }
276
277 /**
278  Wraps the WSpace cst_count_var_val method.
279  */
280 void count_var_val(void* sp, int n, int* vids, int vid, int rel_type, int val) {
281     return static_cast<WSpace*>(sp)->cst_count_var_val(n, vids, vid, rel_type, val);
282 }
283
284 /**
285  Wraps the WSpace cst_count_var_var method.
286  */
287 void count_var_var(void* sp, int n, int* vids, int vid1, int rel_type, int vid2) {
288     return static_cast<WSpace*>(sp)->cst_count_var_var(n, vids, vid1, rel_type, vid2);
289 }
290
291 /**
292  Wraps the WSpace cst_count_var_set_val method.
293  */
294 void count_var_set_val(void*sp, int n, int* vids, int s, int* set, int rel_type, int val){
295     return static_cast<WSpace*>(sp)->cst_count_var_set_val(n, vids, s, set, rel_type, val);
296 }
297
298 /**
299  Wraps the WSpace cst_count_array_val method.
300  */
301 void count_array_val(void* sp, int n, int* vids, int* c, int rel_type, int val){
302     return static_cast<WSpace*>(sp)->cst_count_array_val(n, vids, c, rel_type, val);
303 }
304
305 /**
306  Wraps the WSpace cst_nvalues method.
307  */
308 void nvalues(void* sp, int n, int* vids, int rel_type, int vid) {
309     return static_cast<WSpace*>(sp)->cst_nvalues(n, vids, rel_type, vid);
310 }
311
312 /**
313  Wraps the WSpace cst_circuit method.
314  */
315 void circuit(void* sp, int n, int* c, int* vids1, int* vids2, int vid) {
316     return static_cast<WSpace*>(sp)->cst_circuit(n, c, vids1, vids2, vid);
317 }
318
319 /**
320  Wraps the WSpace cst_precede method
321  */
322 void precede(void* sp, int n, int* vids, int s, int u){
323     return static_cast<WSpace*>(sp)->cst_precede(n, vids, s, u);
324 }
325
326 /**
327  Wraps the WSpace cst_boolop_val method.
328  */
329 void val_boolop(void* sp, int vid1, int bool_op, int vid2, int val) {
330     return static_cast<WSpace*>(sp)->cst_boolop_val(vid1, bool_op, vid2, val);
331 }
332
333 /**
```

```
334  Wraps the WSpace cst_boolop_var method.
335  */
336 void var_boolop(void* sp, int vid1, int bool_op, int vid2, int vid3) {
337     return static_cast<WSpace*>(sp)->cst_boolop_var(vid1, bool_op, vid2, vid3);
338 }
339
340 /**
341  Wraps the WSpace cst_boolrel_val method.
342  */
343 void val_boolrel(void* sp, int vid, int rel_type, int val) {
344     return static_cast<WSpace*>(sp)->cst_boolrel_val(vid, rel_type, val);
345 }
346
347 /**
348  Wraps the WSpace cst_boolrel_var method.
349  */
350 void var_boolrel(void* sp, int vid1, int rel_type, int vid2) {
351     return static_cast<WSpace*>(sp)->cst_boolrel_var(vid1, rel_type, vid2);
352 }
353
354 /**
355  Wraps the WSpace branch method.
356  */
357 void branch(void* sp, int n, int* vids, int var_strategy, int val_strategy) {
358     return static_cast<WSpace*>(sp)->branch(n, vids, var_strategy, val_strategy);
359 }
360
361 /**
362  Wraps the WSpace branch_b method.
363  */
364 void branch_b(void* sp, int n, int* vids, int var_strategy, int val_strategy) {
365     return static_cast<WSpace*>(sp)->branch_b(n, vids, var_strategy, val_strategy);
366 }
367
368 /**
369  Wraps the WSpace cost method.
370  */
371 void cost(void* sp, int vid) {
372     return static_cast<WSpace*>(sp)->cost(vid);
373 }
374
375 /**
376  Wraps the WTimeStop constructor.
377  */
378 void* new_time_stop(int maxTime){
379     return (void*) new WTimeStop(maxTime);
380 }
381
382 /**
383  Wraps the WTimeStop reset method.
384  */
385 void reset_time_stop(void* tStop){
386     WTimeStop* _tStop = static_cast<WTimeStop*>(tStop);
387     _tStop->reset();
388 }
389
390 /**
391  Wraps the WSearchOptions constructor.
392  */
393 void* new_search_options(){
394     return (void*) new WSearchOptions();
395 }
396
397 /**
398  Wraps the WSearchOptions setNbThreads method.
399  */
400 int set_nb_threads(void* sOpts, int nThreads){
401     return static_cast<WSearchOptions*>(sOpts)->setNbThreads(nThreads);
402 }
403
404 /**
```

```
 Wraps the WSearchOptions setTimeStop method.
 Returns the options object passed as an argument as a void pointer
 */
void* set_time_stop(void* sOpts, void* tStop){
    WTimeStop* _tStop = static_cast<WTimeStop*>(tStop);
    WSearchOptions* _sOpts = static_cast<WSearchOptions*>(sOpts);
    _sOpts->setTimeStop(_tStop);
    return (void*) _sOpts;
}

/**
 Wraps the WbabEngine constructor.
 */
void* new_bab_engine(void* sp, void* opts) {
    WSpace* _sp = static_cast<WSpace*>(sp);
    WSearchOptions* _opts = static_cast<WSearchOptions*>(opts);
    return (void*) new WbabEngine(_sp, _opts->getOpts());
}

/**
 Wraps the WbabEngine next method.
 */
void* bab_next(void* se) {
    return (void*) static_cast<WbabEngine*>(se)->next();
}

/**
 Wraps the WbabEngine stopped method.
 */
int bab_stopped(void* se){
    return static_cast<WbabEngine*>(se)->stopped();
}

/**
 Wraps the WdfsEngine constructor.
 */
void* new_dfs_engine(void* sp, void* opts) {
    WSpace* _sp = static_cast<WSpace*>(sp);
    WSearchOptions* _opts = static_cast<WSearchOptions*>(opts);
    return (void*) new WdfsEngine(_sp, _opts->getOpts());
}

/**
 Wraps the WdfsEngine next method.
 */
void* dfs_next(void* se) {
    return (void*) static_cast<WdfsEngine*>(se)->next();
}

/**
 Wraps the WdfsEngine stopped method.
 */
int dfs_stopped(void* se){
    return static_cast<WdfsEngine*>(se)->stopped();
}

/**
 Wraps the WSpace destructor.
 */
void release(void* sp) {
    delete static_cast<WSpace*>(sp);
}

/**
 Wraps the WSpace value method.
 */
int get_value(void* sp, int vid) {
    return static_cast<WSpace*>(sp)->value(vid);
}

/**
```

```
476  Wraps the WSpace values method.
477  */
478 int* get_values(void* sp, int n, int* vids) {
479     return static_cast<WSpace*>(sp)->values(n, vids);
480 }
481
482 /**
483  Wraps the WSpace print method.
484  */
485 void print_vars(void* sp, int n, int* vids) {
486     return static_cast<WSpace*>(sp)->print(n, vids);
487 }
```

B.2 Lisp Wrapper

The Lisp wrapper is made of two files :

- **gecode-wrapper.lisp** that calls the functions from the external library via CFFI. It is the lower level of the gecode-wrapper.
- **gecode-wrapper-ui.lisp** that wraps the CFFI function calls to make the function calls more user-friendly

The following sections show the content of these files.

B.2.1 gecode-wrapper.lisp

```
(cl:defpackage "gil"
  (:nicknames "GIL")
   (:use common-lisp :cl-user :cl :cffi))

(in-package :gil)

(cffi::defcfun ("computation_space" new-space) :pointer
    "Create a new computation space."
)

(cffi::defcfun ("add_intVar" add-int-var-low) :int
    "Add an IntVar ranging from min to max to the specified space. Return the reference of
    this variable for this space."
    (sp :pointer)
    (min :int)
    (max :int)
)

(cffi::defcfun ("add_intVarWithDom" add-int-var-dom-aux) :int
    "Add an IntVar with domain dom of size s to the specified space. Return the reference of
     this variable for this space."
    (sp :pointer)
    (s :int)
    (dom :pointer)
)

(defun add-int-var-dom-low (sp dom)
    "Add an IntVar with domain dom to the specified space. Return the reference of this
    variable for this space."
    (let ((x (cffi::foreign-alloc :int :initial-contents dom)))
        (add-int-var-dom-aux sp (length dom) x))
)

(cffi::defcfun ("add_intVarArray" add-int-var-array-aux) :pointer
    "Add n IntVar ranging from min to max to the specified space."
    (sp :pointer)
    (n :int)
    (min :int)
    (max :int)
)

(defun add-int-var-array-low (sp n min max)
    "Add n IntVar ranging from min to max to the specified space. Return the references of
    those variables for this space"
    (let ((p (add-int-var-array-aux sp n min max)))
        (loop for i from 0 below n
            collect (cffi::mem-aref p :int i)))
)

(cffi::defcfun ("add_intVarArrayWithDom" add-int-var-array-dom-aux) :pointer
    "Add n IntVar with domain dom of size s to the specified space."
    (sp :pointer)
    (n :int)
    (s :int)
    (dom :pointer)
)

(defun add-int-var-array-dom-low (sp n dom)
    "Add n IntVar with domain dom to the specified space. Return the references of those
    variables for this space"
    (let ((x (cffi::foreign-alloc :int :initial-contents dom))
          p)
        (setq p (add-int-var-array-dom-aux sp n (length dom) x))
        (loop for i from 0 below n
            collect (cffi::mem-aref p :int i)))
)

(cffi::defcfun ("set_solution_vars" set-solution-vars-aux) :pointer
    (sp :pointer)
```

```
65      (n :int)
66      (vids :pointer)
67  )
68
69  (defun set-solution-vars (sp vids)
70      (let ((x (cffi::foreign-alloc :int :initial-contents vids))
71            p)
72          (setq p (set-solution-vars-aux sp (length vids) x))
73          (loop for i from 0 below (length vids)
74              collect (cffi::mem-aref p :int i))
75      )
76  )
77
78  (cffi::defcfun ("nvars" nvars) :int
79      "Return the number of variables in the space."
80      (sp :pointer)
81  )
82
83  ;IntVar relation flags
84  (defparameter gil::IRT_EQ 0)     ; equality relation
85  (defparameter gil::IRT_NQ 1)     ; inequality
86  (defparameter gil::IRT_LQ 2)     ; Less or equal
87  (defparameter gil::IRT_LE 3)     ; Strictly lower
88  (defparameter gil::IRT_GQ 4)     ; Greater or equal
89  (defparameter gil::IRT_GR 5)     ; Strictly greater
90
91  (cffi::defcfun ("add_boolVar" add-bool-var-range) :int
92      "Add a BoolVar ranging from l to h. Return the index to this BoolVar."
93      (sp :pointer)
94      (l :int)
95      (h :int)
96  )
97
98  (cffi::defcfun ("add_boolVar_expr_val" add-bool-var-expr-val) :int
99      "Add a BoolVar corresponding to the evalueation of rel-type(vid, val)."
100     (sp :pointer)
101     (vid :int)
102     (rel-type :int)
103     (val :int)
104 )
105
106 (cffi::defcfun ("add_boolVar_expr_var" add-bool-var-expr-var) :int
107     "Add a BoolVar corresponding to the evalueation of rel-type(vid1, vid2)."
108     (sp :pointer)
109     (vid1 :int)
110     (rel-type :int)
111     (vid2 :int)
112 )
113
114 (cffi::defcfun ("val_rel" val-rel) :void
115     "Post a variable/value rel constraint."
116     (sp :pointer)
117     (vid :int)
118     (rel-type :int)
119     (val :int)
120 )
121
122 (cffi::defcfun ("var_rel" var-rel) :void
123     "Post a variable/variable rel constraint."
124     (sp :pointer)
125     (vid1 :int)
126     (rel-type :int)
127     (vid2 :int)
128 )
129
130 (cffi::defcfun ("arr_val_rel" arr-val-rel-aux) :void
131     "Post a variable-array/value rel constraint."
132     (sp :pointer)
133     (n :int)
134     (vids :pointer)
135     (rel-type :int)
```

```
136     (val :int)
137 )
138
139 (defun arr-val-rel (sp vids rel-type val)
140     "Post a variable-array/value rel constraint."
141     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
142         (arr-val-rel-aux sp (length vids) x rel-type val))
143 )
144
145 (cffi::defcfun ("arr_var_rel" arr-var-rel-aux) :void
146     "Post a variable-array/variable rel constraint."
147     (sp :pointer)
148     (n :int)
149     (vids :pointer)
150     (rel-type :int)
151     (vid :int)
152 )
153
154 (defun arr-var-rel (sp vids rel-type vid)
155     "Post a variable-array/variable rel constraint."
156     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
157         (arr-var-rel-aux sp (length vids) x rel-type vid))
158 )
159
160 (cffi::defcfun ("arr_rel" arr-rel-aux) :void
161     "Post a variable-array rel constraint."
162     (sp :pointer)
163     (n :int)
164     (vids :pointer)
165     (rel-type :int)
166 )
167
168 (defun arr-rel (sp vids rel-type)
169     "Post a variable-array rel constraint."
170     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
171         (arr-rel-aux sp (length vids) x rel-type))
172 )
173
174 (cffi::defcfun ("arr_arr_rel" arr-arr-rel-aux) :void
175     "Post a variable-array/variable-array rel constraint."
176     (sp :pointer)
177     (n1 :int)
178     (vids1 :pointer)
179     (rel-type :int)
180     (n2 :int)
181     (vids2 :pointer)
182 )
183
184 (defun arr-arr-rel (sp vids1 rel-type vids2)
185     "Post a variable-array/variable-array rel constraint."
186     (let ((x (cffi::foreign-alloc :int :initial-contents vids1))
187           (y (cffi::foreign-alloc :int :initial-contents vids2)))
188         (arr-arr-rel-aux sp (length vids1) x rel-type (length vids2) y))
189 )
190
191 (cffi::defcfun ("distinct" distinct-aux) :void
192     "Post a distinct constraint on the n variables denoted in vids."
193     (sp :pointer)
194     (n :int)
195     (vids :pointer)
196 )
197
198 (defun distinct (sp vids)
199     "Post a distinct constraint on the variables denoted in vids."
200     (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
201         (distinct-aux sp (length vids) x))
202 )
203
204 (cffi::defcfun ("val_linear" val-linear-aux) :void
205     "Post a linear equation constraint."
206     (sp :pointer)
```

```
207     (n :int)
208     (c :pointer)
209     (vids :pointer)
210     (rel-type :int)
211     (val :int)
212 )
213
214 (defun val-linear (sp coeffs vars rel-type value)
215     "Post a linear equation constraint. coeffs and vars must have the same number of
        elements."
216     (let ((c (cffi::foreign-alloc :int :initial-contents coeffs))
217           (x (cffi::foreign-alloc :int :initial-contents vars)))
218         (val-linear-aux sp (length coeffs) c x rel-type value))
219 )
220
221 (cffi::defcfun ("var_linear" var-linear-aux) :void
222     "Post a linear equation constraint."
223     (sp :pointer)
224     (n :int)
225     (c :pointer)
226     (vids :pointer)
227     (rel-type :int)
228     (vid :int)
229 )
230
231 (defun var-linear (sp coeffs vars rel-type vid)
232     "Post a linear equation constraint. coeffs and vars must have the same number of
        elements."
233     (let ((c (cffi::foreign-alloc :int :initial-contents coeffs))
234           (x (cffi::foreign-alloc :int :initial-contents vars)))
235         (var-linear-aux sp (length coeffs) c x rel-type vid))
236 )
237
238 (cffi::defcfun ("arithmetics_abs" ge-abs) :void
239     "Post the constraint that |vid1| = vid2."
240     (sp :pointer)
241     (vid1 :int)
242     (vid2 :int)
243 )
244
245 (cffi::defcfun ("arithmetics_div" ge-div) :void
246     "Post the constraint that vid3 = vid1/vid2."
247     (sp :pointer)
248     (vid1 :int)
249     (vid2 :int)
250     (vid3 :int)
251 )
252
253 (cffi::defcfun ("arithmetics_mod" var-mod) :void
254     "Post the constraint that vid1 % vid2 = vid3."
255     (sp :pointer)
256     (vid1 :int)
257     (vid2 :int)
258     (vid3 :int)
259 )
260
261 (cffi::defcfun ("arithmetics_divmod" ge-divmod) :void
262     "Post the constraint that vid3 = vid1/vid2 and vid4 = vid1 % vid2."
263     (sp :pointer)
264     (vid1 :int)
265     (vid2 :int)
266     (vid3 :int)
267     (vid4 :int)
268 )
269
270 (cffi:defcfun ("arithmetics_min" ge-min) :void
271     "Post the constraint that vid3 = min(vid1, vid2)."
272     (sp :pointer)
273     (vid1 :int)
274     (vid2 :int)
275     (vid3 :int)
```

```
)

(cffi:defcfun ("arithmetics_arr_min" ge-arr-min-aux) :void
    "Post the constraint that vid = min(vids)."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun ge-arr-min (sp vid vids)
    "Post the constraint that vid = min(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (ge-arr-min-aux sp (length vids) x vid))
)

(cffi:defcfun ("arithmetics_argmin" ge-argmin-aux) :void
    "Post the constraint that vid = argmin(vids)."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun ge-argmin (sp vids vid)
    "Post the constraint that vid = argmin(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (ge-argmin-aux sp (length vids) x vid))
)

(cffi:defcfun ("arithmetics_max" ge-max) :void
    "Post the constraint that vid3 = max(vid1, vid2)."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi:defcfun ("arithmetics_arr_max" ge-arr-max-aux) :void
    "Post the constraint that vid = max(vids)."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun ge-arr-max (sp vid vids)
    "Post the constraint that vid = max(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (ge-arr-max-aux sp (length vids) x vid))
)

(cffi:defcfun ("arithmetics_argmax" ge-argmax-aux) :void
    "Post the constraint that vid = argmax(vids)."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun ge-argmax (sp vids vid)
    "Post the constraint that vid = argmax(vids)."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (ge-argmax-aux sp (length vids) x vid))
)

(cffi:defcfun ("arithmetics_mult" ge-mult) :void
    "Post the constraint that vid3 = vid1 * vid2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
```

```
    (vid3 :int)
)

(cffi:defcfun ("arithmetics_sqr" ge-sqr) :void
    "Post the constraint that vid2 = vid1^2."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)

(cffi:defcfun ("arithmetics_sqrt" ge-sqrt) :void
    "Post the constraint that vid2 = vid1^(1/2)."
    (sp :pointer)
    (vid1 :int)
    (vid2 :int)
)

(cffi:defcfun ("arithmetics_pow" ge-pow) :void
    "Post the constraint that vid2 = vid1^n."
    (sp :pointer)
    (vid1 :int)
    (n :int)
    (vid2 :int)
)

(cffi:defcfun ("arithmetics_nroot" ge-nroot) :void
    "Post the constraint that vid2 = vid1^(1/n)."
    (sp :pointer)
    (vid1 :int)
    (n :int)
    (vid2 :int)
)

(cffi::defcfun ("set_dom" set-dom-aux) :void
    "Post the constraint that dom(vid) = domain of size n."
    (sp :pointer)
    (vid :int)
    (n :int)
    (domain :pointer)
)

(defun set-dom (sp vid domain)
    "Post the constraint that dom(vid) = domain."
    (let ((x (cffi::foreign-alloc :int :initial-contents domain)))
        (set-dom-aux sp vid (length domain) x))
)

(cffi::defcfun ("set_member" set-member-aux) :void
    "Post the constraint that vid is a member vids."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
)

(defun set-member (sp vids vid)
    "Post the constraint that vid is a member vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (set-member-aux sp (length vids) x vid))
)

(cffi::defcfun ("rel_sum" rel-sum-aux) :void
    "Post the constraint that vid = sum(vids). n is the number of vars in vids."
    (sp :pointer)
    (vid :int)
    (n :int)
    (vids :pointer)
)

(defun rel-sum (sp vid vids)
    "Post the constraint that vid = sum(vids)."
```

```
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (rel-sum-aux sp vid (length vids) x))
)

(cffi::defcfun ("count_val_val" count-val-val-aux) :void
    "Post the constraint that the number of variables in vids equal to val1 has relation
    rel-type with val2."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (val1 :int)
    (rel-type :int)
    (val2 :int)
)

(defun count-val-val (sp vids val1 rel-type val2)
    "Post the constraint that the number of variables in vids equal to val1 has relation
    rel-type with val2."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-val-val-aux sp (length vids) x val1 rel-type val2))
)

(cffi::defcfun ("count_val_var" count-val-var-aux) :void
    "Post the constraint that the number of variables in vids equal to val1 has relation
    rel-type with vid."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (val :int)
    (rel-type :int)
    (vid :int)
)

(defun count-val-var (sp vids val rel-type vid)
    "Post the constraint that the number of variables in vids equal to val has relation
    rel-type with vid."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-val-var-aux sp (length vids) x val rel-type vid))
)

(cffi::defcfun ("count_var_val" count-var-val-aux) :void
    "Post the constraint that the number of variables in vids equal to vid has relation
    rel-type with val."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid :int)
    (rel-type :int)
    (val :int)
)

(defun count-var-val (sp vids vid rel-type val)
    "Post the constraint that the number of variables in vids equal to vid has relation
    rel-type with val."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-var-val-aux sp (length vids) x vid rel-type val))
)

(cffi::defcfun ("count_var_var" count-var-var-aux) :void
    "Post the constraint that the number of variables in vids equal to vid1 has relation
    rel-type with vid2."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(defun count-var-var (sp vids vid1 rel-type vid2)
    "Post the constraint that the number of variables in vids equal to vid1 has relation
```

```
    rel-type with vid2."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (count-var-var-aux sp (length vids) x vid1 rel-type vid2))
)

(cffi::defcfun ("count_var_set_val" count-var-set-val-aux) :void
"Post the constraint that the number of variables in vids belonging to the set set has relation rel-type
    with val."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (s :int)
    (s-set :pointer)
    (rel-type :int)
    (val :int)
)

(defun count-var-set-val (sp vids s-set rel-type val)
"Post the constraint that the number of variables in vids belonging to the set set has relation rel-type
    with val."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids))
             (y (cffi::foreign-alloc :int :initial-contents s-set)))
        (count-var-set-val-aux sp (length vids) x (length s-set) y rel-type val))
)

(cffi::defcfun ("count_array_val" count-array-val-aux) :void
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (c :pointer)
    (rel-type :int)
    (val :int)
)

(defun count-array-val (sp vids c rel-type val)
"Post the constraint that the number of times that vars[i] = c[i] is equal to val"
        (let ((x (cffi::foreign-alloc :int :initial-contents vids))
                 (y (cffi::foreign-alloc :int :initial-contents c)))
            (count-array-val-aux sp (length vids) x y rel-type val))
)

(cffi::defcfun ("nvalues" nvalues-aux) :void
    "Post the constraint the number of distinct values in the n variables denoted by vids
     has the given rel-type relation with the variable vid."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (rel-type :int)
    (vid :int)
)

(defun nvalues (sp vids rel-type vid)
    "Post the constraint the number of distinct values in the n variables denoted by vids
    has the given rel-type relation with the variable vid."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (nvalues-aux sp (length vids) x rel-type vid))
)

(cffi::defcfun ("circuit" hcircuit-aux) :void
    "Post the constraint that values of vids1 are the edges of an hamiltonian circuit in
    the graph formed by the n variables in vids1, vids2 are the costs of these edges
    described
    by c, and vid is the total cost of the circuit, i.e. sum(vids2)."
    (sp :pointer)
    (n :int)
    (c :pointer)
    (vids1 :pointer)
    (vids2 :pointer)
    (vid :int)
)
```

```
(defun hcircuit (sp c vids1 vids2 vid)
    "Post the constraint that values of vids1 are the edges of an hamiltonian circuit in
    the graph formed by the variables in vids1, vids2 are the costs of these edges described
    by c, and vid is the total cost of the circuit, i.e. sum(vids2)."
    (let ((costs (cffi::foreign-alloc :int :initial-contents c))
          (x (cffi::foreign-alloc :int :initial-contents vids1))
          (y (cffi::foreign-alloc :int :initial-contents vids2)))
        (hcircuit-aux sp (length vids1) costs x y vid))
)

(cffi::defcfun ("precede" precede-aux) :void
    "Post the constraint that if there exists j (0 <= j < |x|) such that x[j] = u,
    then there must exist i with i < j such that x[i] = s"
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (s :int)
    (u :int)
)

(defun precede (sp vids s u)
    "Post the constraint that if there exists j (0 <= j < |x|) such that x[j] = u,
    then there must exist i with i < j such that x[i] = s"
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (precede-aux sp (length vids) x s u)
    )
)

;BoolVar operation flags
(defparameter gil::BOT_AND 0)    ; logical and
(defparameter gil::BOT_OR 1)     ; logical or
(defparameter gil::BOT_IMP 2)    ; logical implication
(defparameter gil::BOT_EQV 3)    ; logical equivalence
(defparameter gil::BOT_XOR 4)    ; logical exclusive or

(cffi::defcfun ("val_boolop" val-bool-op) :void
    "Post the constraint that val = bool-op(vid1, vid2)."
    (sp :pointer)
    (vid1 :int)
    (bool-op :int)
    (vid2 :int)
    (val :int)
)

(cffi::defcfun ("var_boolop" var-bool-op) :void
    "Post the constraint that vid3 = bool-op(vid1, vid2)."
    (sp :pointer)
    (vid1 :int)
    (bool-op :int)
    (vid2 :int)
    (vid3 :int)
)

(cffi::defcfun ("val_boolrel" val-bool-rel) :void
    "Post boolean rel constraint."
    (sp :pointer)
    (vid :int)
    (rel-type :int)
    (val :int)
)

(cffi::defcfun ("var_boolrel" var-bool-rel) :void
    "Post boolean rel constraint."
    (sp :pointer)
    (vid1 :int)
    (rel-type :int)
    (vid2 :int)
)

(cffi::defcfun ("branch" branch-aux) :void
    "Post branching on the n IntVars denoted by vids."
```

```
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (var-strat :int)
    (val-strat :int)
)

(defun branch (sp vids var-strat val-strat)
    "Post branching on the IntVars denoted by vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (branch-aux sp (length vids) x var-strat val-strat))
)

(cffi::defcfun ("branch_b" branch-b-aux) :void
    "Post branching on the n BoolVars denoted by vids."
    (sp :pointer)
    (n :int)
    (vids :pointer)
    (var-strat :int)
    (val-strat :int)
)

(defun branch-b (sp vids var-strat val-strat)
    "Post branching on the BoolVars denoted by vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (branch-b-aux sp (length vids) x var-strat val-strat))
)

(cffi::defcfun ("cost" set-cost) :void
    "Define which variable is to be the cost."
    (sp :pointer)
    (vid :int)
)

(cffi::defcfun ("new_time_stop" new-time-stop) :pointer
    "Create a new TimeStop object to specify the time after which the search should stop"
    (max-time :int)
)

(cffi::defcfun ("reset_time_stop" reset-time-stop) :void
    "Reset the timer of the timeStop object"
    (t-stop :pointer)
)

(cffi::defcfun ("new_search_options" new-search-options) :pointer
    "Create a new options object to specify the search options"
)

(cffi::defcfun ("set_nb_threads" set-nb-threads) :int
    "Sets the number of threads to use during the search"
    (s-opts :pointer)
    (n-threads :int)
)

(cffi::defcfun ("set_time_stop" set-t-stop) :pointer
    "Sets the stop field of the Options object to the timeStop object"
    (s-opts :pointer)
    (t-stop :pointer)
)

(cffi::defcfun ("new_bab_engine" bab-engine-low) :pointer
    "Create a new branch and bound search-engine."
    (sp :pointer)
    (opts :pointer)
)

(cffi::defcfun ("bab_next" bab-next) :pointer
    "Find the next solution for the search-engine se."
    (se :pointer)
)

```

```
(cffi::defcfun ("bab_stopped" bab-stopped) :int
    "returns t if the search engine has been stopped, nil otherwise"
    (se :pointer)
)

(cffi::defcfun ("new_dfs_engine" dfs-engine-low) :pointer
    "Create a new depth-first search search-engine."
    (sp :pointer)
    (opts :pointer)
)

(cffi::defcfun ("dfs_next" dfs-next) :pointer
    "Find the next solution for the search-engine se."
    (se :pointer)
)

(cffi::defcfun ("dfs_stopped" dfs-stopped) :int
    "returns t if the search engine has been stopped, nil otherwise"
    (se :pointer)
)

(cffi::defcfun ("get_value" get-value) :int
    "Get the value of the variable denoted by vid."
    (sp :pointer)
    (vid :int)
)

(cffi::defcfun ("get_values" get-values-aux) :pointer
    "Get the values of the n variables denoted by vids."
    (sp :pointer)
    (n :int)
    (vids :pointer)
)

(defun get-values (sp vids)
    "Print the values of the variables denoted by vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids))
          p)
        (setq p (get-values-aux sp (length vids) x))
        (loop for i from 0 below (length vids)
            collect (cffi::mem-aref p :int i)))
)

(cffi::defcfun ("print_vars" print-vars-aux) :void
    "Print the values of the n variables denoted by vids."
    (sp :pointer)
    (n :int)
    (vids :pointer)
)

(defun print-vars (sp vids)
    "Print the values of the variables denoted by vids."
    (let ((x (cffi::foreign-alloc :int :initial-contents vids)))
        (print-vars-aux sp (length vids) x))
)
```

B.2.2 gecode-wrapper-ui.lisp

```
(cl:defpackage "gil"
  (:nicknames "GIL")
   (:use common-lisp :cl-user :cl :cffi))

(in-package :gil)
;;;;;;;;;;;;;;;;;;;;;;;;
; Creating int variables ;
;;;;;;;;;;;;;;;;;;;;;;;;

(defclass int-var ()
    ((id :initarg :id :accessor id))
)

(defmethod add-int-var (sp l h)
    "Adds a integer variable with domain [l,h] to sp"
    (make-instance 'int-var :id (add-int-var-low sp l h)))

(defmethod add-int-var-dom (sp dom)
    "Adds a integer variable with domain dom to sp"
    (make-instance 'int-var :id (add-int-var-dom-low sp dom)))

(defmethod add-int-var-array (sp n l h)
    "Adds an array of n integer variables with domain [l,h] to sp"
    (loop for v in (add-int-var-array-low sp n l h) collect
        (make-instance 'int-var :id v)))

(defmethod add-int-var-array-dom (sp n dom)
    "Adds an array of n integer variables with domain dom to sp"
    (loop for v in (add-int-var-array-dom-low sp n dom) collect
        (make-instance 'int-var :id v)))

(defmethod g-specify-sol-variables (sp vids)
    "Specifies the variables that will contain the solution"
    (set-solution-vars sp vids)
)

;id getter
(defmethod vid ((self int-var))
    "Gets the vid of the variable self"
    (id self))

(defmethod vid ((self list))
    "Gets the vids of the variables in self"
    (loop for v in self collect (vid v)))

;;;;;;;;;;;;;;;;;;;;;;;;
; Creating bool variables ;
;;;;;;;;;;;;;;;;;;;;;;;;

(defclass bool-var ()
    ((id :initarg :id :accessor id))
)

(defmethod add-bool-var (sp l h)
    "Adds a boolean variable with domain [l,h] to sp"
    (make-instance 'bool-var :id (add-bool-var-range sp l h)))

(defmethod add-bool-var-expr (sp (v1 int-var) rel-type (v2 fixnum))
    "Adds a boolean variable representing the expression
    v1 rel-type v2 to sp"
    (make-instance 'bool-var
        :id (add-bool-var-expr-val sp (vid v1) rel-type v2)))

(defmethod add-bool-var-expr (sp (v1 int-var) rel-type (v2 int-var))
    (make-instance 'bool-var
        :id (add-bool-var-expr-var sp (vid v1) rel-type (vid v2))))

;id getter
(defmethod vid ((self bool-var)) (id self))
```

```
70
71 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
72 ; Methods for int constraints ;
73 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
74
75 ;REL
76 (defmethod g-rel (sp (v1 int-var) rel-type (v2 fixnum))
77     "Post the constraint that v1 rel-type v2."
78     (val-rel sp (vid v1) rel-type v2))
79
80 (defmethod g-rel (sp (v1 int-var) rel-type (v2 int-var))
81     (var-rel sp (vid v1) rel-type (vid v2)))
82
83 (defmethod g-rel (sp (v1 list) rel-type (v2 null))
84     (arr-rel sp (vid v1) rel-type))
85
86 (defmethod g-rel (sp (v1 list) rel-type (v2 fixnum))
87     (arr-val-rel sp (vid v1) rel-type v2))
88
89 (defmethod g-rel (sp (v1 list) rel-type (v2 int-var))
90     (arr-var-rel sp (vid v1) rel-type (vid v2)))
91
92 (defmethod g-rel (sp (v1 list) rel-type (v2 list))
93     (arr-arr-rel sp (vid v1) rel-type (vid v2)))
94
95 ;DISTINCT
96 (defmethod g-distinct (sp vars)
97     "Post the constraint that the given vars are distinct."
98     (distinct sp (vid vars)))
99
100 ;LINEAR
101 (defmethod g-linear (sp coeffs vars rel-type (v fixnum))
102     "Post the linear relation coeffs*vars rel-type v."
103     (val-linear sp coeffs (vid vars) rel-type v))
104
105 (defmethod g-linear (sp coeffs vars rel-type (v int-var))
106     (var-linear sp coeffs (vid vars) rel-type (vid v)))
107
108 ;ARITHMETICS
109 (defmethod g-abs (sp (v1 int-var) (v2 int-var))
110     "Post the constraints that v2 = |v1|."
111     (ge-abs sp (vid v1) (vid v2)))
112
113 (defmethod g-div (sp (v1 int-var) (v2 int-var) (v3 int-var))
114     "Post the constraints that v3 = v1/v2."
115     (ge-div sp (vid v1) (vid v2) (vid v3)))
116
117 (defmethod g-mod (sp (v1 int-var) (v2 int-var) (v3 int-var))
118     "Post the constraints that v3 = v1%v2."
119     (var-mod sp (vid v1) (vid v2) (vid v3)))
120
121 (defmethod g-divmod (sp (v1 int-var) (v2 int-var) (v3 int-var) (v4 int-var))
122     "Post the constraints that v3 = v1/v2 and v4 = v1%v2."
123     (ge-divmod sp (vid v1) (vid v2) (vid v3) (vid v4)))
124
125 (defmethod g-min (sp (v1 int-var) (v2 int-var) (v3 int-var) &rest vars)
126     "Post the constraints that v1 = min(v2, v3, ...)."
127     (cond
128         ((null vars)
129             (ge-min sp (vid v2) (vid v3) (vid v1)))
130         (t (ge-arr-min sp (vid v1)
131             (append (list (vid v2) (vid v3)) (vid vars))))))
132
133 (defmethod g-lmin (sp (v int-var) vars)
134     "Post the constraints that v = min(vars)."
135     (ge-arr-min sp  (vid v) (vid vars)))
136
137 (defmethod g-argmin (sp vars (v int-var))
138     "Post the constraints that v = argmin(vars)."
139     (ge-argmin sp (vid vars) (vid v2)))
140
```

```
(defmethod g-max (sp (v1 int-var) (v2 int-var) (v3 int-var) &rest vars)
    "Post the constraints that v1 = max(v2, v3, ...)."
    (cond ((null vars) (ge-max sp (vid v2) (vid v3) (vid v1)))
          (t (ge-arr-max sp (vid v1) (append (list (vid v2) (vid v3)) (vid vars))))))

(defmethod g-lmax (sp (v int-var) vars)
    "Post the constraints that v = max(vars)."
    (ge-arr-max sp (vid v) (vid vars)))

(defmethod g-argmax (sp vars (v int-var))
    "Post the constraints that v2 = argmax(vars)."
    (ge-argmax sp (vid vars) (vid v)))

(defmethod g-mult (sp (v1 int-var) (v2 int-var) (v3 int-var))
    "Post the constraints that v3 = v1*v2."
    (ge-mult sp (vid v1) (vid v2) (vid v3)))

(defmethod g-sqr (sp (v1 int-var) (v2 int-var))
    "Post the constraints that v2 is the square of v1."
    (ge-sqr sp (vid v1) (vid v2)))

(defmethod g-sqrt (sp (v1 int-var) (v2 int-var))
    "Post the constraints that v2 square root of v1."
    (ge-sqrt sp (vid v1) (vid v2)))

(defmethod g-pow (sp (v1 int-var) n (v2 int-var))
    "Post the constraints that v2 nth power of v1."
    (ge-pow sp (vid v1) n (vid v2)))

(defmethod g-nroot (sp (v1 int-var) n (v2 int-var))
    "Post the constraints that v2 is the nth root of v1."
    (ge-nroot sp (vid v1) n (vid v2)))

(defmethod g-sum (sp (v int-var) vars)
    "Post the constraints that v = sum(vars)."
    (rel-sum sp (vid v) (vid vars)))

;DOM
(defmethod g-dom (sp (v int-var) dom)
    "Post the constraints that dom(v) = dom."
    (set-dom sp (vid v) dom))

(defmethod g-member (sp vars (v int-var))
    "Post the constraints that v is in vars."
    (set-member sp (vid vars) (vid v)))


;COUNT
(defmethod g-count (sp vars (v1 fixnum) rel-type (v2 fixnum))
    "Post the constraints that v2 is the number of times v1 occurs in vars."
    (count-val-val sp (vid vars) v1 rel-type v2))

(defmethod g-count (sp vars (v1 fixnum) rel-type (v2 int-var))
    (count-val-var sp (vid vars) v1 rel-type (vid v2)))

(defmethod g-count (sp vars (v1 int-var) rel-type (v2 fixnum))
    (count-var-val sp (vid vars) (vid v1) rel-type v2))

(defmethod g-count (sp vars (v1 int-var) rel-type (v2 int-var))
    (count-var-var sp (vid vars) (vid v1) rel-type (vid v2)))

(defmethod g-count (sp vars (s-set list) rel-type (val fixnum))
    (count-var-set-val sp (vid vars) s-set rel-type val)
)

(defmethod g-count-array (sp vars (c list) rel-type (val fixnum))
    (count-array-val sp (vid vars) c rel-type val)
)

;NUMBER OF VALUES
(defmethod g-nvalues (sp vars rel-type (v int-var))
```

```
212     "Post the constraints that v is the number of distinct values in vars."
213     (nvalues sp (vid vars) rel-type (vid v)))
214
215 ;HAMILTONIAN PATH/CIRCUIT
216 (defmethod g-circuit (sp costs vars1 vars2 v)
217     "Post the constraint that values of vars1 are the edges of an hamiltonian circuit in
218     the graph formed by the n variables in vars1, vars2 are the costs of these edges
        described
219     by costs, and v is the total cost of the circuit, i.e. sum(vars2)."
220     (hcircuit sp costs (vid vars1) (vid vars2) (vid v)))
221
222 ;VALUE PRECEDENCE
223 (defmethod g-precede (sp vars s u)
224     "Post the constraint that if there exists j (0 <= j < |x|) such that x[j] = u,
225     then there must exist i with i < j such that x[i] = s"
226     (precede sp (vid vars) s u)
227 )
228
229 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
230 ; Methods for bool constraints ;
231 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
232
233 ;OP
234 (defmethod g-op (sp (v1 bool-var) bool-op (v2 bool-var) (v3 fixnum))
235     "Post the constraints that v1 bool-op v2 = v3."
236     (val-bool-op sp (vid v1) bool-op (vid v2) v3))
237
238 (defmethod g-op (sp (v1 bool-var) bool-op (v2 bool-var) (v3 bool-var))
239     (var-bool-op sp (vid v1) bool-op (vid v2) (vid v3)))
240
241 ;REL
242 (defmethod g-rel (sp (v1 bool-var) rel-type (v2 fixnum))
243     "Post the constraints that v1 rel-type v2."
244     (val-bool-rel sp (vid v1) rel-type v2))
245
246 (defmethod g-rel (sp (v1 bool-var) rel-type (v2 bool-var))
247     (var-bool-rel sp (vid v1) rel-type (vid v2)))
248
249 ;;;;;;;;;;;;;;;;;;;;;;;;;;
250 ; Methods for exploration ;
251 ;;;;;;;;;;;;;;;;;;;;;;;;;;
252
253 ;Variable branching strategies
254 (defparameter gil::INT_VAR_SIZE_MIN 0) ; select first the variable with the smallest domain
255 (defparameter gil::INT_VAR_RND 1) ; select first a random variable
256 (defparameter gil::INT_VAR_DEGREE_MAX 2); select the variable with the highest degree
257
258 ;Value branching strategies
259 (defparameter gil::INT_VAL_MIN 0) ; select first the smallest value of the domain
260 (defparameter gil::INT_VAL_RND 1) ; select first a random value
261 (defparameter gil::INT_VAL_SPLIT_MIN 2) ; select the values not greater than the (min+max)/2
262 (defparameter gil::INT_VAL_SPLIT_MAX 3) ; select the values greater than (min+max)/2
263 (defparameter gil::INT_VAL_MED 4) ; selects the greatest value not bigger than the median
264
265 (defmethod g-branch (sp (v int-var) var-strat val-strat)
266     "Post a branching on v with strategies var-strat and val-strat."
267     (branch sp (list (vid v)) var-strat val-strat))
268
269 (defmethod g-branch (sp (v bool-var) var-strat val-strat)
270     (branch-b sp (list (vid v)) var-strat val-strat))
271
272 (defmethod g-branch (sp (v list) var-strat val-strat)
273     (if (typep (car v) 'int-var)
274         (branch sp (vid v) var-strat val-strat)
275         (branch-b sp (vid v) var-strat val-strat)))
276
277 ;cost
278 (defmethod g-cost (sp (v int-var))
279     "Defines that v is the cost of sp."
280     (set-cost sp (vid v)))
281
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Methods for search engines ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Methods for search engine options

(defclass time-stop ()
    ((ts :initform nil :initarg ts :accessor ts)); ts is a void pointer to a WTimeStop object in Gecode
)

(defmethod t-stop ()
    (make-instance 'time-stop)
)

(defmethod time-stop-init (tstop max-time)
    (setf (ts tstop) (new-time-stop max-time))
)

(defmethod time-stop-reset (tstop)
    (reset-time-stop (ts tstop))
)

(defclass search-options ()
    ((opts :initform nil :initarg opts :accessor opts)); opts is a void* to a WSearchOptions object in Gecode
)

(defmethod search-opts ()
    (make-instance 'search-options)
)

(defmethod init-search-opts (sopts)
    (setf (opts sopts) (new-search-options))
)

(defmethod set-n-threads (s-opts nthreads)
    (set-nb-threads (opts s-opts) nthreads)
)

(defmethod set-time-stop (s-opts t-stop)
    (set-t-stop (opts s-opts) (ts t-stop))
)

;Search-engine types
(defparameter gil::DFS "dfs")
(defparameter gil::BAB "bab")

(defclass BAB-engine ()
    ((bab :initform nil :initarg :bab :accessor bab))
)

(defclass DFS-engine ()
    ((dfs :initform nil :initarg :dfs :accessor dfs))
)

(defmethod search-engine (sp opts se-type)
    "Creates a new search engine (dfs or bab)."
    (cond
        ((string-equal se-type gil::DFS) (make-instance 'DFS-engine :dfs (dfs-engine-low sp opts)))
        ((string-equal se-type gil::BAB) (make-instance 'BAB-engine :bab (bab-engine-low sp opts)))
    )
)

;solution exist?
(defun sol? (sol)
    "Existence predicate for a solution"
    (and (not (cffi::null-pointer-p sol)) sol))

;next solution
(defmethod search-next ((se BAB-engine))
    "Search the next solution of se."
    (sol? (bab-next (bab se))))
```

```
353
354 (defmethod search-next ((se DFS-engine))
355     (sol? (dfs-next (dfs se))))
356
357 (defmethod search-next ((se null))
358     nil)
359
360 ;stopped
361 ; returns 1 if stopped, 0 if not
362 (defmethod stopped ((se BAB-engine))
363     (bab-stopped (bab se))
364 )
365
366 (defmethod stopped ((se DFS-engine))
367     (dfs-stopped (dfs se))
368 )
369
370 ;;;;;;;;;;;;;;;;;;;;;;;;
371 ; Methods for solutions ;
372 ;;;;;;;;;;;;;;;;;;;;;;;;
373
374 ;values
375 (defmethod g-values (sp (v int-var))
376     "Get the values assigned to v."
377     (get-value sp (vid v)))
378
379 (defmethod g-values (sp (v list))
380     (get-values sp (vid v)))
381
382 (defmethod g-values ((sp null) v)
383     nil)
384
385 ;print
386 (defmethod g-print (sp (v int-var))
387     "Print v."
388     (print-vars sp (list (vid v))))
389
390 (defmethod g-print (sp (v list))
391     (print-vars sp (vid v)))
392
393 (defmethod g-print ((sp null) v)
394     nil)
```

Appendix C

Melodizer source code

This appendix gives the code for Melodizer. It is composed of four files :

- **melodizer-csts.lisp** that contains all the musical constraints developed in the context of this master's thesis.
- **melodizer-csp.lisp** that contains the musical CSPs as well as the method to search for a solution.
- **melodizer-utils.lisp** that contains a variety of utility functions.
- **melodizer.lisp** that contains the code for the melodizer object as well as for the interface that is the editor window of the melodizer object.

The current version of Melodizer can be found at `https://github.com/sprockeelsd/Melodizer`.

C.1 melodizer-csts.lisp

```
1  (in-package :mldz)
2
3  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
4  ; ALL-DIFFERENT-NOTES constraint ;
5  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
6
7  ; <sp> is the space
8  ; <notes> is a list of IntVars representing the pitch of the notes
9  ; ensures that all the notes are different in terms of strict value, not in terms of notes
10 ; (e.g. 60 and 72 can be values taken by two variables simultaneously even
11 ; though they both represent a C)
12 (defun all-different-notes (sp notes)
13     (gil::g-distinct sp notes)
14 )
15
16 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
17 ; MELODY DIRECTION constraints ;
18 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
19
20 ; <sp> is the space
21 ; <notes> is a list of IntVars representing the pitch of the notes
22 ; posts the constraint that notes[i] < notes[i+1]
23 (defun strictly-increasing-pitch (sp notes)
24     (gil::g-rel sp notes gil::IRT_LE nil) ; nil = v2
25 )
26
27 ; <sp> is the space
28 ; <notes> is a list of IntVars representing the pitch of the notes
29 ; posts the constraint that notes[i] <= notes[i+1]
30 (defun increasing-pitch (sp notes)
31     (gil::g-rel sp notes gil::IRT_LQ nil) ; nil = v2
32 )
33
34 ; <sp> is the space
```

```
; <notes> is a list of IntVars representing the pitch of the notes
; <intervals> is a list if IntVars representing the intervals between consecutive notes
; <global-interval> is a list if IntVars representing the global interval that the melody spans
; ensures that the melodic direction is mostly upwards
(defun mostly-increasing-pitch (sp notes intervals global-interval)
    (let (sum interval-domain-greater-than-zero)
        ;  variable to hold the result of the sum of all intervals (<=127)
        (setq sum (gil::add-int-var sp -127 127))
        (setq interval-domain-greater-than-zero
            (loop :for n :from 1 :below 13 :by 1 collect n)); [1..12]

        (gil::g-sum sp sum intervals); sum = sum(intervals)
        ; sum >= global-interval
        (gil::g-rel sp sum gil::IRT_GQ (parse-integer global-interval))
        ; (nb of intervals > 0) >= 0.8*size(intervals)
        (gil::g-count sp intervals interval-domain-greater-than-zero gil::IRT_GQ
            (ceiling (* 80 (length intervals)) 100))
    )
)

; <sp> is the space
; <notes> is a list of IntVars representing the pitch of the notes
; posts the constraint that notes[i] > notes[i+1]
(defun strictly-decreasing-pitch (sp notes)
    (gil::g-rel sp notes gil::IRT_GR nil) ; nil = v2
)

; <sp> is the space
; <notes> is a list of IntVars representing the pitch of the notes
; posts the constraint that notes[i] >= notes[i+1]
(defun decreasing-pitch (sp notes)
    (gil::g-rel sp notes gil::IRT_GQ nil) ; nil = v2
)

; <sp> is the space
; <notes> is a list of IntVars representing the pitch of the notes
; <intervals> is a list if IntVars representing the intervals between consecutive notes
; <global-interval> is the global interval that the melody spans
; ensures that the melodic direction is mostly downwards
(defun mostly-decreasing-pitch (sp notes intervals global-interval)
    (let (sum interval-domain-smaller-than-zero)
        ; variable to hold the result of the sum of the intervals
        (setq sum (gil::add-int-var sp (- (* 127 (length intervals))) -1))
        (setq interval-domain-smaller-than-zero
            (loop :for n :from 1 :to 13 :by 1 collect (- n))); [-12...-1]

        (gil::g-sum sp sum intervals); sum = sum(intervals)
        ; sum <= -global-interval
        (gil::g-rel sp sum gil::IRT_LQ (- (parse-integer global-interval)))
        ; (nb of intervals < 0) >= 0.8*size(intervals)
        (gil::g-count sp intervals interval-domain-smaller-than-zero gil::IRT_GQ
            (ceiling (* 80 (length intervals)) 100))
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;
; AT LEAST N constraint ;
;;;;;;;;;;;;;;;;;;;;;;;;

;<sp> is the space
;notes> is the array of variables representing the pitch
;<values> is the array of values
;<n> is the number of variables such that notes[i] = values[i]
; Ensures that the number of times that notes[i] = values[i] is at least n
(defun at-least-n (sp notes values n)
    (gil::g-count-array sp notes values gil::IRT_EQ n)
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; RANGE RESTRICTION constraint ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```
106
107 ;<sp> is the space
108 ;<notes> is the array of variables representing the pitch
109 ;<input-chords> is the input chords in the form of a voice object
110 ; Ensures that the melody is always higher than the lowest note from the chords and lower
111 ; than the highest note of the chord + 2 octaves
112 (defun range-restriction (sp notes input-chords)
113     ; get the list of pitches for all the chords in the voice object
114     (let ((chords (om::chords input-chords))
115         vals min-note max-note)
116         ; collect all the notes and put them in one big list
117         (setf vals
118                 (to-midi (om::flat
119                     (loop :for n :from 0 :below (length chords) :by 1 collect
120                         (om::lmidic (nth n chords)))
121                 ))
122         )
123         (setf min-note (min-list vals)); get the lowest note
124         (setf max-note (max-list vals)); get the highest note
125
126         (dolist (note notes);restrain the interval domains to acceptable values
127             (gil::g-rel sp note gil::IRT_GQ min-note) ; note >= min-note
128             ; note <= max-note + 2 octaves (24 semitones)
129             (gil::g-rel sp note gil::IRT_LQ (+ max-note 24))
130         )
131     )
132 )
133
134 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
135 ; INTERVAL-BETWEEN-ADJACENT-NOTES constraint ;
136 ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
137
138 ; <sp> is the space
139 ; <notes> is a list of IntVars representing the pitch of the notes
140 ; <intervals> is a list of IntVars representing the intervals between successive notes
141 ; Ensures that the interval between two adjacent notes is valid
142 ; The interval must be everything up to a minor sixth (it can be an octave) except for a
143 ; tritone, a major seven or a minor seven
144 ; (-1 -4 -5 -7 -8 -9 -12 0 1 2 3 4 5 7 8 12) admissible intervals
145 ; IMPROVEMENT IDEA : PRIORITIZE SMALLER INTERVALS
146 (defun interval-between-adjacent-notes (sp notes intervals)
147     (let ((valid-intervals '(-1 -4 -5 -7 -8 -9 -12 0 1 2 3 4 5 7 8 12)))
148         (dolist (interval intervals);restrain the interval domains to acceptable values
149             (gil::g-dom sp interval valid-intervals)
150         )
151     )
152 )
153
154 ;;;;;;;;;;;;;;;;;;;;;;;;
155 ; IN TONALITY constraint ;
156 ;;;;;;;;;;;;;;;;;;;;;;;;
157
158 ; <sp> is the space
159 ; <notes> is the variable array on which the constraint is executed
160 ; <key> is the key
161 ; <mode> is the mode
162 ; Ensures that the notes are in the tonality specified by the user(e.g. C major)
163 (defun in-tonality (sp notes key mode)
164     (let (admissible-notes)
165         ; get the list of notes in that tonality
166         (setf admissible-notes (notes-from-tonality key mode))
167         ; set the domain of each variable to the set of notes from the tonality
168         (loop :for j :from 0 :below (length notes) :do
169             (gil::g-dom sp (nth j notes) admissible-notes)
170         )
171     )
172 )
173
174 ; returns the set of notes that are in the tonality specified by the arguments
175 (defun notes-from-tonality (key mode)
176     (let (admissible-notes note scale i)
```

```
        ; get the scale in semi-tones
        (setq scale (get-scale mode))
        ; then, create a list and add the notes in it
        (setq note key)
        (setq i 0)
        (setq admissible-notes (list))
        ; add all notes above the key, then add all notes below the key
        (om::while (<= note 127) :do
            (setq admissible-notes (cons note admissible-notes)); add it to the list
            (if (>= i (length scale))
                (setq i 0)
            )
            (incf note (nth i scale)); note = note + scale[i mod 6]
            (incf i 1); i++
        )
        (setq note key)
        (decf note (nth (- (- (length scale) 1) 0) scale)); note = note - scale[6-i mod 6]
        (setq i 1)

        (om::while (>= note 0) :do
            (setq admissible-notes (cons note admissible-notes)); add it to the list
            (if (>= i (length scale))
                (setq i 0)
            )
            (decf note (nth (- (- (length scale) 1) i) scale)); note = note-scale[6-i mod 6]
            (incf i 1); i++
        )
        admissible-notes
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; HARMONIC-INTERVAL-CHORD constraint ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; <sp> is the space,<notes> is the variable array on which the constraint is executed
; <input-rhythm> is the voice object of the rhythm for the melody we want to find
; <chords> is the voice object of the chords
(defun harmonic-interval-chord (sp notes input-rhythm chords)
    ; get the starting time of each of the notes of the melody
    (let ((melody-starting-times (voice-onsets input-rhythm))
        ; get the starting time of each of the chords
        (chords-starting-times (voice-onsets chords))
        (chord-counter 0); counter to know which chord we are currently looking at
        (variable-counter 0); counter to keep track of the note
        ; counter to keep track of which variable is the start of the melody over the given
    chord
        (local-starting-note-melody 0)
        (starting-time-next-chord 0))
        (dolist (c chords-starting-times); go through the chords starting times
            (setf starting-time-next-chord (nth (+ 1 chord-counter)
                chords-starting-times)); get the starting time of the next chord
            ; keep a track of the first note in the context of this chord
            (setf local-starting-note-melody variable-counter)
            (cond
                ((>= (+ 1 variable-counter) (length melody-starting-times)); last note
                        (return )))
            ; go through all the notes that we haven't seen yet
            (dolist (m (subseq melody-starting-times variable-counter))
                (cond
                    ; last note
                    ((= variable-counter (- (length melody-starting-times) 1))
                        (hic-constraint sp notes local-starting-note-melody
                            variable-counter))
                    ((typep starting-time-next-chord 'null); last chord
                        ; post the constraint on all remaining notes
                        (hic-constraint sp notes local-starting-note-melody
                            (- (length melody-starting-times) 1))
                        (return ))
                    ; if the note is in the context of this chord
                    ((< m starting-time-next-chord)
```

```
                        (incf variable-counter 1))
                    ; the note is not in the context of this chord
                    ((>= m starting-time-next-chord)
                        ; post the constraint on the corresponding notes
                        (if (>= (- variable-counter 1) local-starting-note-melody)
                            (hic-constraint sp notes local-starting-note-melody
                                (- variable-counter 1)))
                        (return ))))
            (setf chord-counter (+ 1 chord-counter)))))

; post the constraint that max(pitch[starting-note],..., pitch[ending-note]) -
min(pitch[starting-note],..., pitch[ending-note]) <= 12 (an octave)
(defun hic-constraint (sp notes starting-note ending-note)
    (let (notes-array max-min-array)
        ; create an array to represent all the notes played with that chord
        (setf notes-array
            (gil::add-int-var-array sp (+ (- ending-note starting-note) 1) 0 127))
        ; create a table where the first value is the max and the second is the min
        (setf max-min-array (gil::add-int-var-array sp 2 0 127))
        ;for each element of note-array, make it equal to the note it represents
        (loop :for j :from 0 :below (+ (- ending-note starting-note) 1) :do
            ; notes-array[j] = notes[j + starting-notes]
            (gil::g-rel sp (nth j notes-array) gil::IRT_EQ
                (nth (+ starting-note j) notes))
        )
        (gil::g-lmax sp (first max-min-array) notes-array); max-val = max(notes-array)
        (gil::g-lmin sp (second max-min-array) notes-array); min-val = min(notes-array)
        ; max(notes-array) - min(notes-array) <= 12)
        (gil::g-linear sp '(1 -1) max-min-array gil::IRT_LQ 12)
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;
; NOTE-ON-CHORDS constraint ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;

; <sp> is the space,<notes> is the variable array on which the constraint is executed
; <input-rhythm> is the voice object of the rhythm for the melody we want to find
; <chords> is the voice object of the chords
(defun note-on-chord (sp notes input-rhythm chords)
    ; get the starting time of each of the notes of the melody and the chords
    (let ((melody-starting-times (voice-onsets input-rhythm))
        (chords-starting-times (voice-onsets chords))
        (chord-counter 0); counter to know which chord we are currently looking at
        (variable-counter 0)); counter to keep track of the note
        (dolist (c chords-starting-times); go through the chords starting times
            ;go through the input-rhythm starting times
            (dolist (m (subseq melody-starting-times variable-counter))
                (cond
                    ((< m c) ; increment the counter for variables
                        (setf variable-counter (+ variable-counter 1))
                    )
                    ((= m c) ; post the constraint on that specific variable
                        (apply-constraint-noc sp notes variable-counter chord-counter chords
    )
                        (setf variable-counter (+ variable-counter 1))
                        (return )
                    )
                    ((> m c) ; break the loop and go to the next chord
                        (return )
                    )
                )
            )
            (setf chord-counter (+ 1 chord-counter))
        )
    ))
(defun apply-constraint-noc (sp notes variable-id chord-id input-chords)
    (let ((chord-pitch (to-midi (om::lmidic (nth chord-id (om::chords input-chords))))); get
     the values of the notes of the chord
            intervals mode inversion admissible-notes)
        (sort chord-pitch #'<) ; sort the note values in increasing order
```

```
316         ; get the intervals in semitones between the different notes of the chord
317         (setf intervals (list
318             (- (second chord-pitch) (first chord-pitch))
319             (- (third chord-pitch) (second chord-pitch))))
320         ; get the mode of the chord from the intervals between the notes
321         (setf mode (first (get-mode-and-inversion intervals)))
322         ; get the inversion of the chord
323         (setf inversion (second (get-mode-and-inversion intervals)))
324         ;get the notes playable on that chord
325         (setf admissible-notes (get-admissible-notes chord-pitch mode))
326         ; post the constraint that the domain of that variable is the admissible notes
327         (gil::g-dom sp (nth variable-id notes) admissible-notes)
328         ; post the constraint that the interval between the note of the melody and the
        highest note of the chord is maximum 1 octave
329         (gil::g-rel sp (nth variable-id notes) gil::IRT_LQ (+ (max (first chord-pitch)) 12))
330     ))
```

C.2 melodizer-csp.lisp

```
1  (in-package :mldz)
2
3  ;;;;;;;;;;;;
4  ; VARIATOR ;
5  ;;;;;;;;;;;;
6
7  ; <input-chords> is a voice object with the chords on top of which the melody is played
8  ; <input-melody> is the melody that we wish to vary
9  ; <key> is the key of the tonality
10 ; <mode> is the mode of the tonality
11 ; <fraction> is the proportion of notes from the melody that must be the same
12 (defmethod variator (input-chords input-melody key mode fraction)
13     (let ((sp (gil::new-space)); create the space
14             pitch intervals dfs tstop sopts nidentical initial-melody)
15
16         ; initialize the variables
17         (setq pitch (gil::add-int-var-array sp (om::n-pulses input-melody) 0 127))
18         (setq intervals (gil::add-int-var-array sp (- (length pitch) 1) -127 127))
19
20         ; temporary, should be set by the user
21         (setq nidentical (ceiling (* fraction (length pitch))))
22
23         ; get the initial melody's notes
24         (setf initial-melody
25             (loop :for n :from 0 :below (length (om::chords input-melody)) :by 1 collect
26                 (first (om::lmidic (nth n (om::chords input-melody))))))
27         )
28         (setf initial-melody (to-midi initial-melody))
29
30         ; connect pitches to intervals
31         (loop :for j :from 0 :below (length intervals) :do ;for each interval
32             (let (temp)
33                 (setq temp (gil::add-int-var-array sp 3 -12 108)); temporary variables
34
35                 (gil::g-rel sp (first temp) gil::IRT_EQ (nth j pitch)); temp[0] = pitch[j]
36                 ; temp[1] = pitch[j+1]
37                 (gil::g-rel sp (second temp) gil::IRT_EQ (nth (+ j 1) pitch))
38                 ; temp[2] = intervals[j]
39                 (gil::g-rel sp (third temp) gil::IRT_EQ (nth j intervals))
40                 ; -pitch[j] + pitch[j+1] - intervals[j] = 0
41                 (gil::g-linear sp '(-1 1 -1) temp gil::IRT_EQ 0)
42             )
43         )
44
45         ; post the constraints
46
47         (range-restriction sp pitch input-chords)
48
49         (interval-between-adjacent-notes sp pitch intervals)
50
```

```
        (in-tonality sp pitch key mode)

        (note-on-chord sp pitch input-melody input-chords)

        (harmonic-interval-chord sp pitch input-melody input-chords)

        ; constraint to ensure that the number of notes that are identical is right
        (at-least-n sp pitch initial-melody nidentical)

        ; branching
        (gil::g-branch sp pitch gil::INT_VAR_DEGREE_MAX gil::INT_VAL_RND)
        (gil::g-branch sp intervals gil::INT_VAR_SIZE_MIN gil::INT_VAL_RND)

        ;time stop
        (setq tstop (gil::t-stop)); create the time stop object
        (gil::time-stop-init tstop 500); initialize it (time is expressed in ms)

        ;search options
        (setq sopts (gil::search-opts)); create the search options object
        (gil::init-search-opts sopts); initialize it
        (gil::set-n-threads sopts 1); set the number of threads to be used during the search
        (gil::set-time-stop sopts tstop); set the timestop object

        ; search engine
        (setq se (gil::search-engine sp (gil::opts sopts) gil::DFS))

        (print "Variator CSP constructed"); print in the listener
        ; return
        (list se pitch tstop sopts intervals)
    )
)

;;;;;;;;;;;;;;;;
; MELODY-FINDER ;
;;;;;;;;;;;;;;;;

; <input> is a voice object with the chords on top of which the melody will be played
; <rhythm> the rhythm of the melody to be found in the form of a voice object
; <optional-constraints> is a list of optional constraint names
; <global interval> is the global interval that the melody should cover if the mostly
; increasing/decreasing constraint is selected
; <key> is the key in which the melody is
; <mode> is the mode of the tonality (major, minor)
; This function creates the CSP by creating the space and the variables, posting the
; constraints and the branching, specifying the search options and creating the search
    engine.
(defmethod melody-finder (input rhythm optional-constraints &optional (global-interval nil)
    (key 60.0) (mode "major"))
    (let ((sp (gil::new-space)); create the space;
        pitch intervals dfs tstop sopts)

        ;initialize the variables
        (setq pitch (gil::add-int-var-array sp (om::n-pulses rhythm) 0 127))
        ; this can be as large as possible given the domain of pitch, to keep all the
        ; restrictions in the constraint part.
        (setq intervals (gil::add-int-var-array sp (- (length pitch) 1) -127 127))

        ; connect pitches to intervals
        (loop :for j :from 0 :below (length intervals) :do ;for each interval
            (let (temp)
                (setq temp (gil::add-int-var-array sp 3 -12 108)); temporary variables

                (gil::g-rel sp (first temp) gil::IRT_EQ (nth j pitch)); temp[0] = pitch[j]
                ; temp[1] = pitch[j+1]
                (gil::g-rel sp (second temp) gil::IRT_EQ (nth (+ j 1) pitch))
                ; temp[2] = intervals[j]
                (gil::g-rel sp (third temp) gil::IRT_EQ (nth j intervals))
                ; -pitch[j] + pitch[j+1] - intervals[j] = 0
                (gil::g-linear sp '(-1 1 -1) temp gil::IRT_EQ 0)
            )
        )
```

```
120
121         ; post the constraints
122
123         ; mandatory constraints
124
125         (range-restriction sp pitch input)
126
127         (interval-between-adjacent-notes sp pitch intervals)
128
129         (in-tonality sp pitch key mode)
130
131         (note-on-chord sp pitch rhythm input)
132
133         (harmonic-interval-chord sp pitch rhythm input)
134
135         ; optional constraints
136         (post-optional-constraints optional-constraints sp pitch intervals global-interval)
137
138         ; branching
139         (gil::g-branch sp pitch gil::INT_VAR_DEGREE_MAX gil::INT_VAL_RND)
140
141         ; the branching on intervals depends on the optional constraints. If there are none,
142         ; do default branching
143         (cond
144             ((or (find "strictly-increasing-pitch" optional-constraints :test #'equal)
145                 (find "increasing-pitch" optional-constraints :test #'equal)
146                 (find "mostly-increasing-pitch" optional-constraints :test #'equal))
147                 (gil::g-branch sp intervals gil::INT_VAR_SIZE_MIN gil::INT_VAL_SPLIT_MAX)
148             )
149             ((or (find "strictly-decreasing-pitch" optional-constraints :test #'equal)
150                 (find "decreasing-pitch" optional-constraints :test #'equal)
151                 (find "mostly-decreasing-pitch" optional-constraints :test #'equal))
152                 (gil::g-branch sp intervals gil::INT_VAR_SIZE_MIN gil::INT_VAL_SPLIT_MIN)
153             )
154             (T ; default behaviour
155                 (gil::g-branch sp intervals gil::INT_VAR_SIZE_MIN gil::INT_VAL_RND)
156             )
157         )
158
159         ;time stop
160         (setq tstop (gil::t-stop)); create the time stop object
161         (gil::time-stop-init tstop 500); initialize it (time is expressed in ms)
162
163         ;search options
164         (setq sopts (gil::search-opts)); create the search options object
165         (gil::init-search-opts sopts); initialize it
166         (gil::set-n-threads sopts 1); set the number of threads to be used during the search
167         (gil::set-time-stop sopts tstop); set the timestop object to stop the search
168
169         ; search engine
170         (setq se (gil::search-engine sp (gil::opts sopts) gil::DFS))
171
172         (print "melody-finder CSP constructed")
173         ; return
174         (list se pitch tstop sopts intervals)
175     )
176 )
177
178 ;posts the optional constraints specified in the list
179 ; TODO CHANGE LATER SO THE FUNCTION CAN BE CALLED FROM THE STRING IN THE LIST AND NOT WITH A
      SERIES OF IF STATEMENTS
180 (defun post-optional-constraints (optional-constraints sp notes intervals global-interval)
181     (if (find "all-different-notes" optional-constraints :test #'equal)
182         (all-different-notes sp notes)
183     )
184     (if (find "strictly-increasing-pitch" optional-constraints :test #'equal)
185         (strictly-increasing-pitch sp notes)
186     )
187     (if (find "strictly-decreasing-pitch" optional-constraints :test #'equal)
188         (strictly-decreasing-pitch sp notes)
189     )
```

```
    (if (find "increasing-pitch" optional-constraints :test #'equal)
        (increasing-pitch sp notes)
    )
    (if (find "decreasing-pitch" optional-constraints :test #'equal)
        (decreasing-pitch sp notes)
    )
    (if (find "mostly-increasing-pitch" optional-constraints :test #'equal)
        (mostly-increasing-pitch sp notes intervals global-interval)
    )
    (if (find "mostly-decreasing-pitch" optional-constraints :test #'equal)
        (mostly-decreasing-pitch sp notes intervals global-interval)
    )
)

;;;;;;;;;;;;;;;
; SEARCH-NEXT ;
;;;;;;;;;;;;;;;

; <l> is a list containing the search engine for the problem and the variables
; <rhythm> is the input rhythm as given by the user
; <melodizer-object> is a melodizer object
; this function finds the next solution of the CSP using the search engine given as an
    argument
(defmethod search-next (l rhythm melodizer-object)
    (let ((se (first l))
         (pitch* (second l))
         (tstop (third l))
         (sopts (fourth l))
         (intervals (fifth l))
         (check t); for the while loop
         sol pitches)

        (om::while check :do
            (gil::time-stop-reset tstop);reset the tstop timer before launching the search
            (setq sol (gil::search-next se)); search the next solution
            (if (null sol)
                ; check if there are solutions left and if the user wishes to continue
                    ; searching
                (stopped-or-ended (gil::stopped se) (stop-search melodizer-object) tstop)
                (setf check nil); we have found a solution so break the loop
            )
        )
        ; store the values of the solution
        (setq pitches (to-midicent (gil::g-values sol pitch*)))
        (print "solution found")

        ;return a voice object that is the solution we just found
        (make-instance 'voice
            :tree rhythm
            :chords pitches
            :tempo (om::tempo (input-rhythm melodizer-object))
        )
    )
)

; determines if the search has been stopped by the solver because there are no more
; solutions or if the user has stopped the search
(defun stopped-or-ended (stopped-se stop-user tstop)
    ; if the search has not been stopped by the TimeStop object, there is no more solutions
    (if (= stopped-se 0)
        (error "There are no more solutions.")
    )
    ;otherwise, check if the user wants to keep searching or not
    (if stop-user
        (error "The search has been stopped. Press next to continue the search.")
    )
)
```

C.3 melodizer-utils.lisp

```
(in-package :mldz)

; converts a list of MIDI values to MIDIcent
(defun to-midicent (l)
    (if (null l)
        nil
        (cons (* 100 (first l)) (to-midicent (rest l)))
    )
)

; convert from MIDIcent to MIDI
(defun to-midi (l)
    (if (null l)
        nil
        (cons (/ (first l) 100) (to-midi (rest l)))
    )
)

;converts the value of a note to its name
(defmethod note-value-to-name (note)
    (cond
        ((eq note 60) "C")
        ((eq note 61) "C#")
        ((eq note 62) "D")
        ((eq note 63) "Eb")
        ((eq note 64) "E")
        ((eq note 65) "F")
        ((eq note 66) "F#")
        ((eq note 67) "G")
        ((eq note 68) "Ab")
        ((eq note 69) "A")
        ((eq note 70) "Bb")
        ((eq note 71) "B")
    )
)

;converts the name of a note to its value
(defmethod name-to-note-value (name)
    (cond
        ((string-equal name "C") 60)
        ((string-equal name "C#") 61)
        ((string-equal name "D") 62)
        ((string-equal name "Eb") 63)
        ((string-equal name "E") 64)
        ((string-equal name "F") 65)
        ((string-equal name "F#") 66)
        ((string-equal name "G") 67)
        ((string-equal name "Ab") 68)
        ((string-equal name "A") 69)
        ((string-equal name "Bb") 70)
        ((string-equal name "B") 71)
    )
)

; finds the smallest element of a list
(defun min-list (L)
    (cond
        ((null (car L)) nil); the list is empty -> return nil
        ((null (cdr L)) (car L)); the list has 1 element -> return it
        (T
            (let ((head (car L)); default behavior
                 (tailMin (min-list (cdr L))))
                (if (< head tailMin) head tailMin)
            )
        )
    )
)

; finds the biggest element of a list
```

```
(defun max-list (L)
    (cond
        ((null (car L)) nil); the list is empty -> return nil
        ((null (cdr L)) (car L)); the list has 1 element -> return it
        (T
            (let ((head (car L)); default behavior
                 (tailMax (max-list (cdr L))))
                (if (> head tailMax) head tailMax)
            )
        )
    )
)

; function to update the list of solutions in a pop-up menu without having to close and
; re-open the window
; TODO find a more efficient way to do this
(defun update-pop-up (self my-panel data position size output)
  (om::om-add-subviews my-panel
    (om::om-make-dialog-item
      'om::om-pop-up-dialog-item
      position ;(om::om-make-point 5 130)
      size ;(om::om-make-point 320 20)
      "list of solutions"
      :range (loop for item in (make-data-sol data) collect (car item))
      ;:value (mode (object self)); change so it goes to the newest added solution?
      :di-action #'(lambda (m)
                     (cond
                         ((string-equal output "output-solution")
                             ; set the output solution to the currently selected solution
                             (setf (output-solution (om::object self))
                                 (nth (om::om-get-selected-item-index m) data))
                             (let ((indx (om::om-get-selected-item-index m)))
                                 (om::openeditorframe ; open the editor of the solution
                                     (om::omNG-make-new-instance
                                         (nth indx data)
                                         (format nil "melody ~D" (1+ indx)); name-window
                                     )
                                 )
                             )
                         )
                         ((string-equal output "output-motif")
                             (setf (output-motif (om::object self))
                                 (nth (om::om-get-selected-item-index m) data))
                             (let ((indx (om::om-get-selected-item-index m)))
                                 (om::openeditorframe
                                     (om::omNG-make-new-instance
                                         (output-motif (om::object self))
                                         (format nil "motif ~D" (1+ indx)); name-window
                                     )
                                 )
                             )
                         )
                         ((string-equal output "output-phrase")
                             (setf (output-phrase (om::object self))
                                 (nth (om::om-get-selected-item-index m) data))
                             (let ((indx (om::om-get-selected-item-index m)))
                                 (om::openeditorframe
                                     (om::omNG-make-new-instance
                                         (output-phrase (om::object self))
                                         (format nil "phrase ~D" (1+ indx)); name-window
                                     )
                                 )
                             )
                         )
                         ((string-equal output "output-period")
                             (setf (output-period (om::object self))
                                 (nth (om::om-get-selected-item-index m) data))
                             (let ((indx (om::om-get-selected-item-index m)))
                                 (om::openeditorframe
                                     (om::omNG-make-new-instance
                                         (output-period (om::object self))
```

```
                                        (format nil "period ~D" (1+ indx))
                                    )
                                )
                            )
                        )
                    )
      )
    )
  )
)

;function to get the starting times (in seconds) of the notes
; from karim haddad (OM)
(defmethod voice-onsets ((self voice))
  "on passe de voice a chord-seq juste pour avoir les onsets"
    (let ((obj (om::objfromobjs self (make-instance 'om::chord-seq))))
        (butlast (om::lonset obj))
    )
)

; returns the list of intervals defining a given mode
(defun get-scale (mode)
    (cond
        ((string-equal mode "ionian (major)")
            (list 2 2 1 2 2 2 1)
        )
        ((string-equal mode "dorian")
            (list 2 1 2 2 2 1 2)
        )
        ((string-equal mode "phrygian")
            (list 1 2 2 2 1 2 2)
        )
        ((string-equal mode "lydian")
            (list 2 2 2 1 2 2 1)
        )
        ((string-equal mode "mixolydian")
            (list 2 2 1 2 2 1 2)
        )
        ((string-equal mode "aeolian (natural minor)")
            (list 2 1 2 2 1 2 2)
        )
        ((string-equal mode "locrian")
            (list 1 2 2 1 2 2 2)
        )
        ((string-equal mode "harmonic minor")
            (list 2 1 2 2 1 3 1)
        )
        ((string-equal mode "pentatonic")
            (list 2 2 3 2 3)
        )
        ((string-equal mode "chromatic")
            (list 1 1 1 1 1 1 1 1 1 1 1 1)
        )
    )
)

; function to get all of a given note (e.g. C)
(defun get-all-notes (note)
    (let ((acc '()) (backup note))
        (om::while (<= note 127) :do
            (setq acc (cons note acc)); add it to the list
            (incf note 12)
        )
        (setf note (- backup 12))
        (om::while (>= note 0) :do
            (setq acc (cons note acc)); add it to the list
            (decf note 12)
        )
        acc
    )
)
```

```
212
213 ; function to get all notes playable on top of a given chord ADD CHORD TYPES
214 (defun get-admissible-notes (chords mode inversion)
215     (let ((return-list '()))
216         (cond
217             ; on top of a major chord, you can play either of the notes
218             ; from the chord though the preferred order is 1-5-3
219             ((string-equal mode "major")
220                 (setf return-list (reduce #'cons
221                     (get-all-notes (first chords))
222                     :initial-value return-list
223                     :from-end t
224                 ))
225                 (setf return-list (reduce #'cons
226                     (get-all-notes (second chords))
227                     :initial-value return-list
228                     :from-end t
229                 ))
230                 (setf return-list (reduce #'cons
231                     (get-all-notes (third chords))
232                     :initial-value return-list
233                     :from-end t
234                 ))
235             )
236             ; on top of a minor chord, you can play either of the notes
237             ; from the chord though the preferred order is 1-5-3
238             ((string-equal mode "minor")
239                 (setf return-list (reduce #'cons
240                     (get-all-notes (first chords))
241                     :initial-value return-list
242                     :from-end t
243                 ))
244                 (setf return-list (reduce #'cons
245                     (get-all-notes (second chords))
246                     :initial-value return-list
247                     :from-end t
248                 ))
249                 (setf return-list (reduce #'cons
250                     (get-all-notes (third chords))
251                     :initial-value return-list
252                     :from-end t
253                 ))
254             )
255             ; only the third can be played on top of diminished chords
256             ((string-equal mode "diminished")
257                 (cond
258                     ((= inversion 0)
259                         (setf return-list (reduce #'cons
260                             (get-all-notes (second chords))
261                             :initial-value return-list
262                             :from-end t
263                         ))
264                     )
265                     ((= inversion 1)
266                         (setf return-list (reduce #'cons
267                             (get-all-notes (first chords))
268                             :initial-value return-list
269                             :from-end t
270                         ))
271                     )
272                     ((= inversion 2)
273                         (setf return-list (reduce #'cons
274                             (get-all-notes (third chords))
275                             :initial-value return-list
276                             :from-end t
277                         ))
278                     )
279                 )
280             )
281         )
282     )
```

```
)

; function to get the mode of the chord (major, minor, diminished,...) and the inversion
; (0 = classical form, 1 = first inversion, 2 = second inversion)
(defun get-mode-and-inversion (intervals)
    ; possible intervals in midi for major chords
    (let ((major-intervals (list (list 4 3) (list 3 5) (list 5 4)))
        ; possible intervals in midi for minor chords
        (minor-intervals (list (list 3 4) (list 4 5) (list 5 3)))
        ; possible intervals in midi for diminished chords
        (diminished-intervals (list (list 3 3) (list 3 6) (list 6 3))))
        (cond
            ((position intervals major-intervals :test #'equal); if the chord is major
                (list "major" (position intervals major-intervals :test #'equal))
            )
            ((position intervals minor-intervals :test #'equal); if the chord is minor
                (list "minor" (position intervals minor-intervals :test #'equal))
            )
            ; if the chord is diminished
            ((position intervals diminished-intervals :test #'equal)
                (list "diminished" (position intervals diminished-intervals :test #'equal))
            )
        )
    )
)

;makes a list (name voice-instance) from a list of voices:
;(from Karim Haddad)
(defun make-data-sol (liste)
  (loop for l in liste
        for i from 1 to (length liste)
        collect (list (format nil "solution ~D: ~A"  i l) l)))


; taken from rhythm box
; https://github.com/blapiere/Rhythm-Box
(defun rel-to-gil (rel)
"Convert a relation operator symbol to a GiL relation value."
    (cond
        ((eq rel '=) gil::IRT_EQ)
        ((eq rel '=/=) gil::IRT_NQ)
        ((eq rel '<) gil::IRT_LE)
        ((eq rel '=<) gil::IRT_LQ)
        ((eq rel '>) gil::IRT_GR)
        ((eq rel '>=) gil::IRT_GQ)
    )
)


; this is not used but kept in case it is needed
; shuffles a list
; from https://gist.github.com/shortsightedsid/62d0ee21bfca53d9b69e
(defun list-shuffler (input-list &optional accumulator)
  "Shuffle a list using tail call recursion."
  (if (eq input-list nil)
      accumulator
      (progn
  (rotatef (car input-list)
     (nth (random (length input-list)) input-list))
  (list-shuffler (cdr input-list)
         (append accumulator (list (car input-list)))))))
```

C.4 melodizer.lisp

```
(in-package :mldz)

;;;====================
;;;= MELODIZER OBJECT =
;;;====================
```

```
6
7 (om::defclass! melodizer ()
8   ;attributes
9   ((input-chords :accessor input-chords :initarg :input-chords :initform
10  (make-instance 'voice) :documentation
11    "The input chords on top of which the melody will be played in the form of a voice
      object.")
12
13    (input-rhythm :accessor input-rhythm :initarg :input-rhythm :initform
14    (make-instance 'voice) :documentation
15      "The rhythm of the melody or a melody in the form of a voice object. ")
16
17    (key :accessor key :initarg :key :initform 60 :documentation
18      "The key the melody is in (default : C).")
19
20    (mode :accessor mode :initarg :mode :initform "ionian (major)" :documentation
21      "The mode the melody is in (default : major).")
22
23    (tool-mode :accessor tool-mode :initarg :tool-mode :initform "Melody-Finder"
24    :documentation
25      "The mode of the tool. melody-finder allows to generate melodies based on a rhythm
      input, while variation-maker allows to make variations of a melody")
26
27    (variety :accessor variety :initarg :variety :initform 1 :documentation
28      "The minimal variety we want for the solution, expressed as a number of notes. (does
      nothing for now)")
29
30    (global-interval :accessor global-interval :initarg :global-interval :initform "1"
31    :documentation
32      "global interval that the produced melody should cover (optional constraint)")
33
34    (optional-constraints :accessor optional-constraints :initarg :optional-constraints :
      initform (list) :documentation
35      "a list of optional constraint names that should be added to the problem")
36
37    (result :accessor result :initarg :result :initform (list) :documentation
38      "A temporary list holder to store the result of the call to the CSPs, shouldn't be
      touched.")
39
40    (stop-search :accessor stop-search :initarg :stop-search :initform nil :documentation
41      "A boolean variable to tell if the user wishes to stop the search or not.")
42
43    (solution :accessor solution :initarg :solution :initform nil :documentation
44      "The current solution of the CSP in the form of a voice object.")
45
46    (solutions-list :accessor solutions-list :initarg :solution-list :initform '()
47    :documentation
48      "The list of all the solutions saved by the user for the current search.")
49
50    (motifs-list :accessor motifs-list :initarg :motifs-list :initform '() :documentation
51      "The list of motifs created by the user.")
52
53    (phrases-list :accessor phrases-list :initarg :phrases-list :initform '() :documentation
54      "The list of phrases created by the user.")
55
56    (periods-list :accessor periods-list :initarg :periods-list :initform '() :documentation
57      "The list of periodes created by the user.")
58
59    (output-solution :accessor output-solution :initarg :output-solution :initform nil
60    :documentation
61      "The selected solution for the current search.")
62
63    (output-motif :accessor output-motif :initarg :output-motif :initform nil :documentation
64      "The selected motif.")
65
66    (output-phrase :accessor output-phrase :initarg :output-phrase :initform nil
67    :documentation
68      "The selected phrase.")
69
70    (output-period :accessor output-period :initarg :output-period :initform nil
71    :documentation
```

```
72        "The selected period.")
73
74      (melody :accessor melody :initarg :melody :initform nil :documentation
75        "The final melody generated by combining different elements.")
76    )
77    (:icon 225)
78    (:documentation "This class implements Melodizer.
79          Melodizer is a constraints based application aiming to improve composer's expression
       and exploration abilities
80          by generating interesting and innovative melodies based on a set of constraints
      expressing musical rules.
81          More information and a tutorial can be found at https://github.com/sprockeelsd/
      Melodizer")
82 )
83
84
85 ; the editor for the object
86 (defclass melodizer-editor (om::editorview) ())
87
88 (defmethod om::class-has-editor-p ((self melodizer)) t)
89 (defmethod om::get-editor-class ((self melodizer)) 'melodizer-editor)
90
91 (defmethod om::om-draw-contents ((view melodizer-editor))
92   (let* ((object (om::object view)))
93     (om::om-with-focused-view
94       view
95       ;;; DRAW SOMETHING ?
96     )
97   )
98 )
99
100 ; To access the melodizer object, (om::object self)
101
102 (defmethod initialize-instance ((self melodizer-editor) &rest args)
103   ;;; do what needs to be done by default
104   (call-next-method) ; start the search by default?
105   (make-my-interface self)
106 )
107
108 ; function to create the tool's interface
109 (defmethod make-my-interface ((self melodizer-editor))
110
111   ; create the main view of the object
112   (make-main-view self)
113
114   ;;;;;;;;;;;;;;;;
115   ;;; sub views ;;;
116   ;;;;;;;;;;;;;;;;
117
118   (let*
119     (
120       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
121       ;;; setting the different regions of the tool ;;;
122       ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
123
124       ; The coordinates here are coordinates in the main view
125
126       ; part of the display for everything that has to do with input
127       (input-panel (om::om-make-view 'om::om-view
128         :size (om::om-make-point 450 300)
129         :position (om::om-make-point 5 30)
130         :bg-color om::*azulito*)
131       )
132       ; part of the display for everything that has to do with the search for solutions
133       (search-panel (om::om-make-view 'om::om-view
134         :size (om::om-make-point 450 300)
135         :position (om::om-make-point 460 30)
136         :bg-color om::*azulito*)
137       )
138       ; part of the display for everything that has to do with adding new constraints to the
      problem
```

```
      (constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 905 400)
        :position (om::om-make-point 5 335)
        :bg-color om::*azulito*)
      )
      ; part of the display to put different solutions together
      (solution-assembly-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 450 705)
        :position (om::om-make-point 915 30)
        :bg-color om::*azulito*)
      )

      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different sub-panels for additional constraints ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      ; coordinates are local to the constraints-panel
      (general-constraints-panel
        (om::om-make-view
          'om::om-view
          :size (om::om-make-point 220 375)
          :position (om::om-make-point 5 20)
          :bg-color (om::om-make-color 0 0.6 0.3))
      )
      (motif-maker-constraints-panel
        (om::om-make-view
          'om::om-view
          :size (om::om-make-point 220 375)
          :position (om::om-make-point 230 20)
          :bg-color (om::om-make-color 0 0.6 0.3))
      )
      (phrase-maker-constraints-panel
        (om::om-make-view
          'om::om-view
          :size (om::om-make-point 220 375)
          :position (om::om-make-point 455 20)
          :bg-color (om::om-make-color 0 0.6 0.3))
      )
      (period-maker-constraints-panel
        (om::om-make-view
          'om::om-view
          :size (om::om-make-point 220 375)
          :position (om::om-make_point 680 20)
          :bg-color (om::om-make-color 0 0.6 0.3))
      )
    )

    ; create the input panel
    (setf elements-input-panel (make-input-panel self input-panel))

    ; create the search panel
    (setf elements-search-panel (make-search-panel self search-panel solution-assembly-panel
    ))

    ; create the constraints panel
    (setf elements-constraints-panel (make-constraints-panel self constraints-panel))
    ;create the general constraints panel
    (setf elements-general-constraints-panel
        (make-general-constraints-panel self general-constraints-panel))
    ; create the motif maker constraints panel
    (setf elements-motif-maker-constraints-panel
        (make-motif-maker-constraints-panel self motif-maker-constraints-panel))
    ; create the phrase maker constraints panel
    (setf elements-phrase-maker-constraints-panel
        (make-phrase-maker-constraints-panel self phrase-maker-constraints-panel))
    ; create the period maker constraints panel
    (setf elements-period-maker-constraints-panel
        (make-period-maker-constraints-panel self period-maker-constraints-panel))

    ; create the solution assembly panel
    (setf elements-solution-assembly-panel
```

```
        (make-solution-assembly-panel self solution-assembly-panel))

    ;;; add new panel here


    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      input-panel
      search-panel
      constraints-panel
      solution-assembly-panel
    )
    ; add subviews to the constraints panel
    (om::om-add-subviews
      constraints-panel
      general-constraints-panel
      motif-maker-constraints-panel
      phrase-maker-constraints-panel
      period-maker-constraints-panel
    )
  )
  ; return the editor
  self
)



    ;;;;;;;;;;;;;;;;
    ;;; main view ;;;
    ;;;;;;;;;;;;;;;;

; this function creates the elements for the main panel
(defun make-main-view (editor)
  ; background colour
  (om::om-set-bg-color editor om::*om-light-gray-color*) ;;pour changer le bg color. om peut
     fabriquer sa propre couleur: (om-make-color r g b)

  ; title
  (om::om-add-subviews
    editor
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 650 2)
      (om::om-make-point 120 20)
      "Melodizer"
      :font om::*om-default-font3b*
    )
  )
)

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;;; creating the input panel ;;;    final positions done
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;this function creates all the elements of the input-panel (buttons, pop-up-menus,...)
; the coordinates are local to the input panel
(defun make-input-panel (editor input-panel)
  (om::om-add-subviews
    input-panel

    ; title
    (om::om-make-dialog-item
       'om::om-static-text
      (om::om-make-point 187 2)
      (om::om-make-point 120 20)
      "Parameters"
      :font om::*om-default-font1b*
    )

    ;pop-up list to select the mode of the tool (melodizer, accompagnement finder, ...)
```

```
    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 15 30)
      (om::om-make-point 200 20)
      "Tool Mode selection"
      :value(tool-mode (om::object editor))
      :range '("Melody-Finder" "Variation maker")
      :di-action #'(lambda (m)
                     ; set the tool-mode according to the choice of the user
                     (setf (tool-mode (om::object editor))
                         (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
      )
    )

    ;pop-up list to select the key of the melody
    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 15 75)
      (om::om-make-point 200 20)
      "Key selection"
      :range '("C" "C#" "D" "Eb" "E" "F" "F#" "G" "Ab" "A" "Bb" "B")
      :value (note-value-to-name (key (om::object editor)))
      :di-action #'(lambda (m)
        ; set the key according to the choice of the user
        (setf (key (om::object editor)) (name-to-note-value
            (nth (om::om-get-selected-item-index m) (om::om-get-item-list m))))
      )
    )

    ;pop-up list to select the mode of the melody
    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 15 105)
      (om::om-make-point 200 20)
      "Mode selection"
      :range '("ionian (major)" "dorian" "phrygian" "lydian" "mixolydian"
      "aeolian (natural minor)" "locrian" "pentatonic" "harmonic minor" "chromatic")
      :value (mode (om::object editor))
      :di-action #'(lambda (m)
        ; set the mode according to the choice of the user
        (setf (mode (om::object editor))
        (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
      )
    )

    ;button to edit the input chords
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 240 71)
      (om::om-make-point 200 20)
      "Edit input chords"
      :di-action #'(lambda (b)
        (om::openeditorframe ; open a voice window displaying the input chords
          (om::omNG-make-new-instance
            (input-chords (om::object editor))
            "input chords" ; name of the window
          ); changes are saved when closed
        )
      )
    )

    ;button to edit the input rhythm
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 240 101)
      (om::om-make-point 200 20)
      "Edit melody rhythm"
      :di-action #'(lambda (b)
        (om::openeditorframe ; open a voice window displaying the input rhythm
          (om::omNG-make-new-instance
            (input-rhythm (om::object editor))
```

```
            "input rhythm" ; name of the window
          )
        )
      )
    )

    ;text for the slider
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 161 150)
      (om::om-make-point 200 20)
      "Variety of the solutions"
      :font om::*om-default-font1*
    )

    ; slider to express how different the solutions should be (100 = completely different,
    ; 1 = almost no difference)
    (om::om-make-dialog-item
      'om::om-slider
      (om::om-make-point 123 170) ; position
      (om::om-make-point 200 20) ; size
      "Slider"
      :range '(1 100)
      :increment 1
      :value (* 100
        (/ (variety (om::object editor)) (n-pulses (input-rhythm (om::object editor)))))
      :di-action #'(lambda (s)
                     ; set the value of variety to  the value of the pointer times
                     ; n-values / 100, rounded down
                     (setf
                       (variety (om::object editor))
                       (floor ;division rounding down
                         (*
                           (om::om-slider-value s)
                           (om::n-pulses (input-rhythm (om::object editor)))
                         )
                         100
                       )
                     )
      )
    )

    ; button to reset the input
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 122 250)
      (om::om-make-point 200 20)
      "Reset input"
      :di-action #'(lambda (b)
        (setf (input-chords (om::object editor)) (make-instance 'voice))
        (setf (input-rhythm (om::object editor)) (make-instance 'voice))
      )
    )
  )
)

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;;; creating the search panel ;;;
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;this function creates all the elements of the search-panel (buttons, pop-up-menus,...)
; coordinates here are local to search-panel
(defun make-search-panel (editor search-panel solution-assembly-panel)
  (om::om-add-subviews
    search-panel

    ; title
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 205 2)
      (om::om-make-point 120 20)
```

```
      "Search"
      :font om::*om-default-font1b*
    )

    ;pop-up list to select the desired solution
    ;this is only for the start, as a new pop-up menu is created with every new solution
    ;/!\ if you move this, you also have to move the new ones that are generating every time
     the list is modified! see update-pop-up function
    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 18 130)
      (om::om-make-point 262 20)
      "Solution selection"
      :range (solutions-list (om::object editor))
      :di-action #'(lambda (m)
        ; set the output to the selected solution
        (setf (output-solution (om::object editor))
            (nth (om::om-get-selected-item-index m) (solutions-list (om::object editor))))
      )
    )

    ; button to see the selected solution from the ones we keep with the input chords

    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 285 126)
      (om::om-make-point 150 20)
      "See with chords"
      :di-action #'(lambda (b)
                    ; if there is no solution to add
                    (if (typep (output-solution (om::object editor)) 'null)
                      (error "There is no selected solution.")
                    )
                    (om::openeditorframe ; open the editor of the selected solution
                      (om::omNG-make-new-instance
                        (make-instance
                          'poly
                          :voices (list
                            (output-solution (om::object editor))
                            (input-chords (om::object editor))
                          )
                        )
                        (format nil "solution"); name of the window
                      )
                    )
      )
    )

    ; button to start or restart the search
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 15 50) ; position (horizontal, vertical)
      (om::om-make-point 130 20) ; size (horizontal, vertical)
      "Start"
      :di-action #'(lambda (b)
                    ; reset the solutions for the new search
                    (setf (solutions-list (om::object editor)) '())
                    (setf (solution (om::object editor)) nil)
                    (progn
                      (update-pop-up editor search-panel
                        (solutions-list (om::object editor)) (om::om-make-point 18 130)
                        (om::om-make-point 262 20) "output-solution")
                      (oa::om-invalidate-view editor)
                    )
                    ; reset the boolean that tells wether we want to stop the search or not
                    (setf (stop-search (om::object editor)) nil)
                    (cond
                      ((string-equal (tool-mode (om::object editor)) "Melody-Finder")
                        (let init; list to take the result of the call to melody-finder
                          ; get the search engine and the first solution of the CSP
                            (setq init (melody-finder (input-chords   (om::object editor))
```

```
                          (input-rhythm (om::object editor))
                          (optional-constraints (om::object editor))
                          (global-interval (om::object editor))
                          (key (om::object editor))
                          (mode (om::object editor))))
                        ; update the fields of the object to their new value
                        (setf (result (om::object editor)) init)
                      )
                    )
                    ((string-equal (tool-mode (om::object editor)) "Variation maker")
                      (let init
                        (setq init (variator (input-chords (om::object editor))
                          (input-rhythm (om::object editor))
                              (key (om::object editor))
                              (mode (om::object editor))
                              0.6))
                        (setf (result (om::object editor)) init)
                      )
                    )
                  )
    )
  )

  ; button to find the next solution
  (om::om-make-dialog-item
    'om::om-button
    (om::om-make-point 160 50) ; position
    (om::om-make-point 130 20) ; size
    "Next"
    :di-action #'(lambda (b)
                  (if (typep (result (om::object editor)) 'null)
                    (error "The problem has not been initialized. Please set the input and
                      press Start.")
                  )
                  (print "Searching for the next solution")
                  ;reset the boolean because we want to continue the search
                  (setf (stop-search (om::object editor)) nil)
                  ;get the next solution
                  ; start a new thread for the execution of the next method
                  (mp:process-run-function
                    "next thread" ; name of the thread, not necessary but useful
                    nil ; process initialization keywords, not needed here
                    (lambda () ; function to call
                      (setf (solution (om::object editor))
                          (search-next (result (om::object editor))
                              (om::tree (input-rhythm (om::object editor)))
                              (om::object editor)))
                      ; tempo = tempo(input)
                      (setf (om::tempo (solution (om::object editor)))
                          (om::tempo (input-rhythm (om::object editor))))
                      (om::openeditorframe ; open a voice window displaying the solution
                        (om::omNG-make-new-instance
                        (solution (om::object editor)); the new solution
                        "current solution" ; name of the window
                        )
                      )
                    )
                    ; arguments if necessary
                  )
    )
  )

  ; button to stop the search if the user wishes to
  (om::om-make-dialog-item
    'om::om-button
    (om::om-make-point 305 50)
    (om::om-make-point 130 20)
    "Stop"
    :di-action #'(lambda (b)
      ; set the boolean to true so when the timestop object tells the search engine it
      ; stopped, it can check and see the user stopped the search
```

```
562         (setf (stop-search (om::object editor)) t)
563       )
564     )
565
566     ; button to open the voice object editor of the current solution
567     (om::om-make-dialog-item
568       'om::om-button
569       (om::om-make-point 15 90); position
570       (om::om-make-point 202 20); size
571       "See solution"
572       :di-action #'(lambda (b)
573                      (if (typep (solution (om::object editor)) 'null)
574                        (error "There is no solution. Start the search to find solutions.")
575                      )
576                        (om::openeditorframe ; open a voice window displaying the solution
577                          (om::omNG-make-new-instance
578                            (solution (om::object editor)); the last solution
579                            "current solution" ; name of the window
580                          )
581                        )
582       )
583     )
584
585     ;button to add the solution to the list of solutions
586     (om::om-make-dialog-item
587       'om::om-button
588       (om::om-make-point 233 90) ; position
589       (om::om-make-point 202 20) ; size
590       "Keep Solution"
591       :di-action #'(lambda (b)
592                      (if (typep (solution (om::object editor)) 'null)
593                        (error "There is no solution to keep.")
594                      )
595                      ;add the element to the list
596                      ; if it's the first solution
597                      (if (typep (solutions-list (om::object editor)) 'null)
598                        (setf (solutions-list (om::object editor))
599                          (list (solution (om::object editor))))
600                        (nconc (solutions-list (om::object editor))
601                          (list (solution (om::object editor))))
602                      )
603                      (progn
604                        (update-pop-up editor search-panel
605                          (solutions-list (om::object editor)) (om::om-make-point 18 130)
606                          (om::om-make-point 262 20) "output-solution")
607                        (oa::om-invalidate-view editor)
608                      )
609       )
610     )
611
612     ; button add the selected solution to the list of motifs
613     (om::om-make-dialog-item
614       'om::om-button
615       (om::om-make-point 15 170) ; position (horizontal, vertical)
616       (om::om-make-point 130 20) ; size (horizontal, vertical)
617       "Add to motifs"
618       :di-action #'(lambda (b)
619                      (if (typep (output-solution (om::object editor)) 'null)
620                        (error "There is no motif to keep.")
621                      )
622                      (if (typep (motifs-list (om::object editor)) 'null)
623                        (setf (motifs-list (om::object editor))
624                          (list
625                            (make-instance 'poly
626                              :voices (list (output-solution (om::object editor))
627                                  (input-chords (om::object editor)))
628                            )
629                          ); initialize the list
630                        )
631                        (nconc (motifs-list (om::object editor))
632                          (list
```

```
                        (make-instance 'poly
                          :voices (list (output-solution (om::object editor))
                              (input-chords (om::object editor)))
                        )
                      ); create a list containing the new motif
                    ); add it to the end
                  )
                  (progn
                    (update-pop-up editor solution-assembly-panel
                        (motifs-list (om::object editor))
                        (om::om-make-point 18 130)
                        (om::om-make-point 262 20)
                        "output-motif"); update the pop-up menu
                    (oa::om-invalidate-view editor)
                  )
    )
  )

  ; button to add the selected solution to the list of phrases
  (om::om-make-dialog-item
    'om::om-button
    (om::om-make-point 160 170) ; position
    (om::om-make-point 130 20) ; size
    "Add to phrases"
    :di-action #'(lambda (b)
                  (if (typep (output-solution (om::object editor)) 'null)
                    (error "There is no phrase to keep.")
                  )
                  (if (typep (phrases-list (om::object editor)) 'null)
                    (setf (phrases-list (om::object editor))
                      (list
                        (make-instance 'poly
                          :voices (list (output-solution (om::object editor))
                              (input-chords (om::object editor)))
                        )
                      ); initialize the list
                    )
                    (nconc (phrases-list (om::object editor))
                      (list
                        (make-instance 'poly
                          :voices (list (output-solution (om::object editor))
                              (input-chords (om::object editor)))
                        )
                      ); create a list containing the phrase
                    ); add it to the end
                  )
                  (progn
                    (update-pop-up editor solution-assembly-panel
                      (phrases-list (om::object editor))
                      (om::om-make-point 18 230)
                      (om::om-make-point 262 20)
                      "output-phrase"); update the pop-up menu
                    (oa::om-invalidate-view editor)
                  )
    )
  )

  ; button to add the selected solution to the list of periods
  (om::om-make-dialog-item
    'om::om-button
    (om::om-make-point 305 170)
    (om::om-make-point 130 20)
    "Add to periods"
    :di-action #'(lambda (b)
                  (if (typep (output-solution (om::object editor)) 'null)
                    (error "There is no period to keep.")
                  )
                  (if (typep (periods-list (om::object editor)) 'null)
                    (setf (periods-list (om::object editor))
                      (list
                        (make-instance 'poly
```

```
                          :voices (list (output-solution (om::object editor))
                              (input-chords (om::object editor)))
                        )
                      ); initialize the list
                    )
                    (nconc (periods-list (om::object editor))
                      (list
                        (make-instance 'poly
                          :voices (list (output-solution (om::object editor))
                              (input-chords (om::object editor)))
                        )
                      ); create a list containing the period
                    ); add it to the end
                  )
                  (progn
                    (update-pop-up editor solution-assembly-panel
                      (periods-list (om::object editor))
                      (om::om-make-point 18 330)
                      (om::om-make-point 262 20)
                      "output-period"); update the pop-up menu
                    (oa::om-invalidate-view editor)
                  )
      )
    )
  )
)

    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
    ;;; creating the constraints panel ;;;
    ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; this function creates the elements of the main additional constraints panel
; coordinates here are local to constraint-panel
(defun make-constraints-panel (editor constraints-panel)
  (om::om-add-subviews
    constraints-panel

    ; title
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 380 2)
      (om::om-make-point 200 20)
      "Additional constraints"
      :font om::*om-default-font1b*
    )
  )
)

; this function creates the elements of the general constraints panel
; coordinates here are local to general-constraints-panel
(defun make-general-constraints-panel (editor general-constraints-panel)
  (om::om-add-subviews
    general-constraints-panel

    ; title
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 47 2)
      (om::om-make-point 200 20)
      "General constraints"
      :font om::*om-default-font1b*
    )

    ;checkbox for all-different constraint
    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 30) ; position
      (om::om-make-point 20 20) ; size
      "All different notes"
      :checked-p (find "all-different-notes" (optional-constraints (om::object editor)) :
    test #'equal)
```

```
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (push "all-different-notes"
                         (optional-constraints (om::object editor)))
                       (setf (optional-constraints (om::object editor))
                         (remove "all-different-notes"
                             (optional-constraints (om::object editor)) :test #'equal))
                     )
      )
    )

    ; name for all-different constraint
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 30 30)
      (om::om-make-point 200 20)
      "All different notes"
      :font om::*om-default-font1*
    )
  )
)

; this function creates the elements of the motif-maker constraints panel
; coordinates here are local to motif-maker-constraints-panel
(defun make-motif-maker-constraints-panel (editor motif-maker-constraints-panel)
  (om::om-add-subviews
    motif-maker-constraints-panel

    ; title
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 35 2)
      (om::om-make-point 200 20)
      "Motif Maker constraints"
      :font om::*om-default-font1b*
    )

    ;checkbox for strictly-increasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 30) ; position
      (om::om-make-point 20 20) ; size
      "Strictly increasing pitch"
      :checked-p (find "strictly-increasing-pitch"
        (optional-constraints (om::object editor)) :test #'equal)
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (push "strictly-increasing-pitch"
                         (optional-constraints (om::object editor)))
                       (setf (optional-constraints (om::object editor))
                         (remove "strictly-increasing-pitch"
                             (optional-constraints (om::object editor)) :test #'equal))
                     )
      )
    )

    ; name for strictly-increasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 30 30)
      (om::om-make-point 150 20)
      "Strictly increasing pitch"
      :font om::*om-default-font1*
    )

    ;checkbox for strictly-decreasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 50) ; position
      (om::om-make-point 20 20) ; size
      "Strictly decreasing pitch"
```

```
    :checked-p (find "strictly-decreasing-pitch"
      (optional-constraints (om::object editor)) :test #'equal)
    :di-action #'(lambda (c)
                   (if (om::om-checked-p c)
                     (push "strictly-decreasing-pitch"
                       (optional-constraints (om::object editor)))
                     (setf (optional-constraints (om::object editor))
                     (remove "strictly-decreasing-pitch"
                       (optional-constraints (om::object editor)) :test #'equal))
                   )
                   (print (optional-constraints (om::object editor)))
    )
  )

  ; name for strictly-decreasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 30 50)
    (om::om-make-point 200 20)
    "Strictly decreasing pitch"
    :font om::*om-default-font1*
  )

  ;checkbox for increasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-check-box
    (om::om-make-point 10 70) ; position
    (om::om-make-point 20 20) ; size
    "Increasing pitch"
    :checked-p (find "increasing-pitch"
      (optional-constraints (om::object editor)) :test #'equal)
    :di-action #'(lambda (c)
                   (if (om::om-checked-p c)
                     (push "increasing-pitch" (optional-constraints (om::object editor)))
                     (setf (optional-constraints (om::object editor))
                       (remove "increasing-pitch"
                           (optional-constraints (om::object editor)) :test #'equal))
                   )
    )
  )

  ; name for increasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 30 70)
    (om::om-make-point 150 20)
    "Increasing pitch"
    :font om::*om-default-font1*
  )

  ;checkbox for decreasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-check-box
    (om::om-make-point 10 90) ; position
    (om::om-make-point 20 20) ; size
    "Decreasing pitch"
    :checked-p (find "decreasing-pitch"
      (optional-constraints (om::object editor)) :test #'equal)
    :di-action #'(lambda (c)
                   (if (om::om-checked-p c)
                     (push "decreasing-pitch"
                       (optional-constraints (om::object editor)))
                     (setf (optional-constraints (om::object editor))
                       (remove "decreasing-pitch"
                           (optional-constraints (om::object editor)) :test #'equal))
                   )
    )
  )

  ; name for decreasing-pitch constraint
  (om::om-make-dialog-item
```

```
    'om::om-static-text
    (om::om-make-point 30 90)
    (om::om-make-point 200 20)
    "Decreasing pitch"
    :font om::*om-default-font1*
  )

  ;checkbox for mostly-increasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-check-box
    (om::om-make-point 10 110) ; position
    (om::om-make-point 20 20) ; size
    "Mostly increasing pitch"
    :checked-p (find "mostly-increasing-pitch"
      (optional-constraints (om::object editor)) :test #'equal)
    :di-action #'(lambda (c)
                   (if (om::om-checked-p c)
                     (push "mostly-increasing-pitch"
                       (optional-constraints (om::object editor)))
                     (setf (optional-constraints (om::object editor))
                       (remove "mostly-increasing-pitch"
                           (optional-constraints (om::object editor)) :test #'equal))
                   )
    )
  )

  ; name for mostly-increasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 30 110)
    (om::om-make-point 150 20)
    "Mostly increasing pitch"
    :font om::*om-default-font1*
  )

  ;checkbox for mostly-decreasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-check-box
    (om::om-make-point 10 130) ; position
    (om::om-make-point 20 20) ; size
    "Mostly decreasing pitch"
    :checked-p (find "mostly-decreasing-pitch"
      (optional-constraints (om::object editor)) :test #'equal)
    :di-action #'(lambda (c)
                   (if (om::om-checked-p c)
                     (push "mostly-decreasing-pitch"
                       (optional-constraints (om::object editor)))
                     (setf (optional-constraints (om::object editor))
                       (remove "mostly-decreasing-pitch"
                           (optional-constraints (om::object editor)) :test #'equal))
                   )
    )
  )

  ; name for mostly-decreasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 30 130)
    (om::om-make-point 150 20)
    "Mostly decreasing pitch"
    :font om::*om-default-font1*
  )

  ; name for the pop-up menu allowing to select the global interval for both mostly
  increasing/decreasing
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 10 150)
    (om::om-make-point 200 20)
    "Total interval (in semitones)"
    :font om::*om-default-font1*
```

```
    )

    ; pop-up menu for the selection of the global interval that the melody should go to for
    both
    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 10 170)
      (om::om-make-point 180 20)
      "Key selection"
      :range (loop :for n :from 1 :below 25 :by 1 collect (write-to-string n))
      :value (global-interval (om::object editor))
      :di-action #'(lambda (m)
        (setf (global-interval (om::object editor))
        (nth (om::om-get-selected-item-index m)
        (om::om-get-item-list m)))
      )
    )
  )
)

; this function creates the elements of the phrase-maker constraints panel
; coordinates here are local to motif-maker-constraints-panel
(defun make-motif-maker-constraints-panel (editor motif-maker-constraints-panel)
  (om::om-add-subviews
    motif-maker-constraints-panel

    ; title
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 35 2)
      (om::om-make-point 200 20)
      "Phrase Maker constraints"
      :font om::*om-default-font1b*
    )

    ;checkbox for strictly-increasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 30) ; position
      (om::om-make-point 20 20) ; size
      "Strictly increasing pitch"
      :checked-p (find "strictly-increasing-pitch"
        (optional-constraints (om::object editor)) :test #'equal)
      :di-action #'(lambda (c)
                    (if (om::om-checked-p c)
                      (push "strictly-increasing-pitch"
                        (optional-constraints (om::object editor)))
                      (setf (optional-constraints (om::object editor))
                        (remove "strictly-increasing-pitch"
                            (optional-constraints (om::object editor)) :test #'equal))
                    )
      )
    )

    ; name for strictly-increasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 30 30)
      (om::om-make-point 150 20)
      "Strictly increasing pitch"
      :font om::*om-default-font1*
    )

    ;checkbox for strictly-decreasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 50) ; position
      (om::om-make-point 20 20) ; size
      "Strictly decreasing pitch"
      :checked-p (find "strictly-decreasing-pitch"
        (optional-constraints (om::object editor)) :test #'equal)
```

```
    :di-action #'(lambda (c)
                   (if (om::om-checked-p c)
                     (push "strictly-decreasing-pitch"
                       (optional-constraints (om::object editor)))
                     (setf (optional-constraints (om::object editor))
                     (remove "strictly-decreasing-pitch"
                       (optional-constraints (om::object editor)) :test #'equal))
                   )
                   (print (optional-constraints (om::object editor)))
    )
  )

  ; name for strictly-decreasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 30 50)
    (om::om-make-point 200 20)
    "Strictly decreasing pitch"
    :font om::*om-default-font1*
  )

  ;checkbox for increasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-check-box
    (om::om-make-point 10 70) ; position
    (om::om-make-point 20 20) ; size
    "Increasing pitch"
    :checked-p (find "increasing-pitch"
      (optional-constraints (om::object editor)) :test #'equal)
    :di-action #'(lambda (c)
                   (if (om::om-checked-p c)
                     (push "increasing-pitch" (optional-constraints (om::object editor)))
                     (setf (optional-constraints (om::object editor))
                       (remove "increasing-pitch"
                           (optional-constraints (om::object editor)) :test #'equal))
                   )
    )
  )

  ; name for increasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 30 70)
    (om::om-make-point 150 20)
    "Increasing pitch"
    :font om::*om-default-font1*
  )

  ;checkbox for decreasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-check-box
    (om::om-make-point 10 90) ; position
    (om::om-make-point 20 20) ; size
    "Decreasing pitch"
    :checked-p (find "decreasing-pitch"
      (optional-constraints (om::object editor)) :test #'equal)
    :di-action #'(lambda (c)
                   (if (om::om-checked-p c)
                     (push "decreasing-pitch"
                       (optional-constraints (om::object editor)))
                     (setf (optional-constraints (om::object editor))
                       (remove "decreasing-pitch"
                           (optional-constraints (om::object editor)) :test #'equal))
                   )
    )
  )

  ; name for decreasing-pitch constraint
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 30 90)
```

```
      (om::om-make-point 200 20)
      "Decreasing pitch"
      :font om::*om-default-font1*
    )

    ;checkbox for mostly-increasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 110) ; position
      (om::om-make-point 20 20) ; size
      "Mostly increasing pitch"
      :checked-p (find "mostly-increasing-pitch"
        (optional-constraints (om::object editor)) :test #'equal)
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (push "mostly-increasing-pitch"
                         (optional-constraints (om::object editor)))
                       (setf (optional-constraints (om::object editor))
                         (remove "mostly-increasing-pitch"
                             (optional-constraints (om::object editor)) :test #'equal))
                     )
      )
    )

    ; name for mostly-increasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 30 110)
      (om::om-make-point 150 20)
      "Mostly increasing pitch"
      :font om::*om-default-font1*
    )

    ;checkbox for mostly-decreasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 130) ; position
      (om::om-make-point 20 20) ; size
      "Mostly decreasing pitch"
      :checked-p (find "mostly-decreasing-pitch"
        (optional-constraints (om::object editor)) :test #'equal)
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (push "mostly-decreasing-pitch"
                         (optional-constraints (om::object editor)))
                       (setf (optional-constraints (om::object editor))
                         (remove "mostly-decreasing-pitch"
                             (optional-constraints (om::object editor)) :test #'equal))
                     )
      )
    )

    ; name for mostly-decreasing-pitch constraint
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 30 130)
      (om::om-make-point 150 20)
      "Mostly decreasing pitch"
      :font om::*om-default-font1*
    )

    ; name for the pop-up menu allowing to select the global interval for both mostly
    increasing/decreasing
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 10 150)
      (om::om-make-point 200 20)
      "Total interval (in semitones)"
      :font om::*om-default-font1*
    )
```

```
1197      ; pop-up menu for the selection of the global interval that the melody should go to for
          both
1198      (om::om-make-dialog-item
1199        'om::pop-up-menu
1200        (om::om-make-point 10 170)
1201        (om::om-make-point 180 20)
1202        "Key selection"
1203        :range (loop :for n :from 1 :below 25 :by 1 collect (write-to-string n))
1204        :value (global-interval (om::object editor))
1205        :di-action #'(lambda (m)
1206          (setf (global-interval (om::object editor))
1207          (nth (om::om-get-selected-item-index m)
1208          (om::om-get-item-list m)))
1209        )
1210      )
1211    )
1212  )
1213
1214  ; this function creates the elements of the period-maker constraints panel
1215  ; coordinates here are local to period-maker-constraints-panel
1216  (defun make-period-maker-constraints-panel (editor period-maker-constraints-panel)
1217    (om::om-add-subviews
1218      period-maker-constraints-panel
1219
1220      ; title
1221      (om::om-make-dialog-item
1222        'om::om-static-text
1223        (om::om-make-point 30 2)
1224        (om::om-make-point 200 20)
1225        "Period Maker constraints"
1226        :font om::*om-default-font1b*
1227      )
1228    )
1229  )
1230
1231      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
1232      ;;; creating the solution assembly panel ;;;
1233      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
1234
1235  ; this function creates the elements of the solution-assembly panel
1236  ; coordinates here are local to solution-assembly-panel
1237  (defun make-solution-assembly-panel (editor solution-assembly-panel)
1238    (om::om-add-subviews
1239      solution-assembly-panel
1240
1241      ; title
1242      (om::om-make-dialog-item
1243        'om::om-static-text
1244        (om::om-make-point 167 2)
1245        (om::om-make-point 120 20)
1246        "Solution assembly"
1247        :font om::*om-default-font1b*
1248      )
1249
1250      ;pop-up list to select the desired motif
1251      ;this is only for the start, as a new pop-up menu is created with every new solution
1252      ;/!\ if you move this, you also have to move the new ones that are generating every time
           the list is modified! see update-pop-up function
1253      (om::om-make-dialog-item
1254        'om::pop-up-menu
1255        (om::om-make-point 18 130)
1256        (om::om-make-point 262 20)
1257        "Motif selection"
1258        :range (motifs-list (om::object editor))
1259        :di-action #'(lambda (m)
1260                       ; set the output to the selected solution
1261                       (setf (output-motif (om::object editor))
1262                           (nth (om::om-get-selected-item-index m)
1263                           (motifs-list (om::object editor))))
1264                       (let ((indx (om::om-get-selected-item-index m)))
1265                           (om::openeditorframe ; open the editor of the selected solution
```

```
                        (om::omNG-make-new-instance
                            (output-motif (om::object editor))
                            (format nil "motif ~D" (1+ indx)); name of the window
                        )
                    )
                )
        )
    )

    ; name for the pop-up list
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 20 110)
      (om::om-make-point 200 20)
      "Motifs"
      :font om::*om-default-font1*
    )

    ; button to add a motif before the current melody
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 15 155)
      (om::om-make-point 202 20)
      "Add before current melody"
      :di-action #'(lambda (b)
                    (if (typep (output-motif (om::object editor)) 'null)
                      (error "There is no motif to add.")
                    )
                    (if (typep (melody (om::object editor)) 'null)
                      (setf (melody (om::object editor)) (output-motif (om::object editor)))
                      (setf (melody (om::object editor))
                      (om::concat (output-motif (om::object editor))
                        (melody (om::object editor))))
                    )
      )
    )

    ; button to add a motif after the current melody
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 233 155)
      (om::om-make-point 202 20)
      "Add after current melody"
      :di-action #'(lambda (b)
                    (if (typep (output-motif (om::object editor)) 'null)
                      (error "There is no motif to add.")
                    )
                    (if (typep (melody (om::object editor)) 'null)
                      (setf (melody (om::object editor)) (output-motif (om::object editor)))
                      (setf (melody (om::object editor))
                      (om::concat (melody (om::object editor))
                        (output-motif (om::object editor))))
                    )
      )
    )

    ;button to reset the list of motifs
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 285 126)
      (om::om-make-point 150 20)
      "Reset"
      :di-action #'(lambda (b)
                    (setf (motifs-list (om::object editor)) nil)
                    (setf (output-motif (om::object editor)) nil)
                    (update-pop-up editor solution-assembly-panel
                        (motifs-list (om::object editor))
                        (om::om-make-point 18 130)
                        (om::om-make-point 262 20)
                        "output-motif"); update the pop-up menu
      )
```

```
1337    )
1338
1339    ; pop-up list to select the desired phrase
1340    (om::om-make-dialog-item
1341      'om::pop-up-menu
1342      (om::om-make-point 18 230)
1343      (om::om-make-point 262 20)
1344      "Phrase selection"
1345      :range (phrases-list (om::object editor))
1346      :di-action #'(lambda (m)
1347                     ; set the output to the selected solution
1348                     (setf (output-phrase (om::object editor))
1349                     (nth (om::om-get-selected-item-index m)
1350                     (phrases-list (om::object editor))))
1351                     (let ((indx (om::om-get-selected-item-index m)))
1352                       (om::openeditorframe
1353                           (om::omNG-make-new-instance
1354                               (output-phrase (om::object editor))
1355                               (format nil "phrase ~D" (1+ indx)); name of the window
1356                           )
1357                       )
1358                     )
1359      )
1360    )
1361
1362    ; name for the pop-up list
1363    (om::om-make-dialog-item
1364      'om::om-static-text
1365      (om::om-make-point 20 210)
1366      (om::om-make-point 200 20)
1367      "Phrases"
1368      :font om::*om-default-font1*
1369    )
1370
1371    ;button to add a phrase before the current melody
1372    (om::om-make-dialog-item
1373      'om::om-button
1374      (om::om-make-point 15 255)
1375      (om::om-make-point 202 20)
1376      "Add before current melody"
1377      :di-action #'(lambda (b)
1378                     (if (typep (output-phrase (om::object editor)) 'null)
1379                       (error "There is no phrase to add.")
1380                     )
1381                     (if (typep (melody (om::object editor)) 'null)
1382                       (setf (melody (om::object editor)) (output-phrase (om::object editor))
    )
1383                       (setf (melody (om::object editor))
1384                       (om::concat (output-phrase (om::object editor))
1385                       (melody (om::object editor))))
1386                     )
1387      )
1388    )
1389
1390    ;button to add  phrase after the current melody
1391    (om::om-make-dialog-item
1392      'om::om-button
1393      (om::om-make-point 233 255)
1394      (om::om-make-point 202 20)
1395      "Add after current melody"
1396      :di-action #'(lambda (b)
1397                     (if (typep (output-phrase (om::object editor)) 'null)
1398                       (error "There is no phrase to add.")
1399                     )
1400                     (if (typep (melody (om::object editor)) 'null)
1401                       (setf (melody (om::object editor)) (output-phrase (om::object editor))
    )
1402                       (setf (melody (om::object editor))
1403                       (om::concat (melody (om::object editor))
1404                       (output-phrase (om::object editor))))
1405                     )
```

```
    )
  )

  ;button to reset the list of phrases
  (om::om-make-dialog-item
    'om::om-button
    (om::om-make-point 285 226)
    (om::om-make-point 150 20)
    "Reset"
    :di-action #'(lambda (b)
                   (setf (phrases-list (om::object editor)) nil)
                   (setf (output-phrase (om::object editor)) nil)
                   (update-pop-up editor solution-assembly-panel
                       (phrases-list (om::object editor))
                       (om::om-make-point 18 230)
                       (om::om-make-point 262 20)
                       "output-motif"); update the pop-up menu
    )
  )

  ; pop-up list to select the desired period
  (om::om-make-dialog-item
    'om::pop-up-menu
    (om::om-make-point 18 330)
    (om::om-make-point 262 20)
    "Period selection"
    :range (periods-list (om::object editor))
    :di-action #'(lambda (m)
                   ; set the output to the selected solution
                   (setf (output-period (om::object editor))
                       (nth (om::om-get-selected-item-index m)
                       (periods-list (om::object editor))))
                   (let ((indx (om::om-get-selected-item-index m)))
                     (om::openeditorframe
                         (om::omNG-make-new-instance
                             (output-period (om::object editor))
                             (format nil "period ~D" (1+ indx))
                         )
                     )
                   )
    )
  )

  ; name for the pop-up list
  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 20 310)
    (om::om-make-point 200 20)
    "Periods"
    :font om::*om-default-font1*
  )

  ; button to add a period before the current melody
  (om::om-make-dialog-item
    'om::om-button
    (om::om-make-point 15 355)
    (om::om-make-point 202 20)
    "Add before current melody"
    :di-action #'(lambda (b)
                   (if (typep (output-period (om::object editor)) 'null)
                     (error "There is no period to add.")
                   )
                   (if (typep (melody (om::object editor)) 'null)
                     (setf (melody (om::object editor)) (output-period (om::object editor))
  )
                     (setf (melody (om::object editor))
                     (om::concat (output-period (om::object editor))
                     (melody (om::object editor))))
                   )
    )
  )
```

```
1476
1477    ; button to add a period after the current melody
1478    (om::om-make-dialog-item
1479      'om::om-button
1480      (om::om-make-point 232 355)
1481      (om::om-make-point 202 20)
1482      "Add after current melody"
1483      :di-action #'(lambda (b)
1484                     (if (typep (output-period (om::object editor)) 'null)
1485                       (error "There is no period to add.")
1486                     )
1487                     (if (typep (melody (om::object editor)) 'null)
1488                       (setf (melody (om::object editor)) (output-period (om::object editor))
    )
1489                       (setf (melody (om::object editor))
1490                       (om::concat (melody (om::object editor))
1491                         (output-period (om::object editor))))
1492                     )
1493      )
1494    )
1495
1496    ;button to reset the list of periods
1497    (om::om-make-dialog-item
1498      'om::om-button
1499      (om::om-make-point 285 326)
1500      (om::om-make-point 150 20)
1501      "Reset"
1502      :di-action #'(lambda (b)
1503                     (setf (periods-list (om::object editor)) nil)
1504                     (setf (output-period (om::object editor)) nil)
1505                     (update-pop-up editor solution-assembly-panel
1506                         (periods-list (om::object editor))
1507                         (om::om-make-point 18 330)
1508                         (om::om-make-point 262 20)
1509                         "output-motif"); update the pop-up menu
1510      )
1511    )
1512
1513    ;button to show the melody
1514    (om::om-make-dialog-item
1515      'om::om-button
1516      (om::om-make-point 15 430)
1517      (om::om-make-point 420 20)
1518      "Show melody"
1519      :di-action #'(lambda (b)
1520                     (if (typep (melody (om::object editor)) 'null)
1521                       (error "There is no melody currently.")
1522                     )
1523                     (om::openeditorframe ; open a voice window displaying the input chords
1524                       (om::omNG-make-new-instance
1525                         (melody (om::object editor))
1526                         "melody" ; name of the window
1527                       )
1528                     )
1529      )
1530    )
1531
1532    ;button to save the melody as a phrase
1533    (om::om-make-dialog-item
1534      'om::om-button
1535      (om::om-make-point 15 460)
1536      (om::om-make-point 202 20)
1537      "Save as phrase"
1538      :di-action #'(lambda (b)
1539                     (if (typep (melody (om::object editor)) 'null)
1540                       (error "There is no melody to keep.")
1541                     )
1542                     (if (typep (phrases-list (om::object editor)) 'null)
1543                       (setf (phrases-list (om::object editor))
1544                         (list (melody (om::object editor))))); initialize the list
1545                       (nconc (phrases-list (om::object editor))
```

```
                        (list (melody (om::object editor)))); add it to the end
                    )
                    (progn
                      (update-pop-up editor solution-assembly-panel
                        (phrases-list (om::object editor))
                            (om::om-make-point 18 230)
                            (om::om-make-point 262 20)
                            "output-phrase"); update the pop-up menu
                      (oa::om-invalidate-view editor)
                    )
      )
    )

    ; button to save the melody as a period
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 233 460)
      (om::om-make-point 202 20)
      "Save as period"
      :di-action #'(lambda (b)
                    (if (typep (melody (om::object editor)) 'null)
                      (error "There is no melody to keep.")
                    )
                    (if (typep (periods-list (om::object editor)) 'null)
                      (setf (periods-list (om::object editor))
                        (list (melody (om::object editor)))); initialize the list
                      (nconc (periods-list (om::object editor))
                        (list (melody (om::object editor)))); add it to the end
                    )
                    (progn
                      (update-pop-up editor solution-assembly-panel
                        (periods-list (om::object editor))
                        (om::om-make-point 18 330)
                        (om::om-make-point 262 20)
                        "output-period"); update the pop-up menu
                      (oa::om-invalidate-view editor)
                    )
      )
    )

    ; button to reset the melody
    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 15 490)
      (om::om-make-point 420 20)
      "Reset melody"
      :di-action #'(lambda (b)
        (setf (melody (om::object editor)) nil)
      )
    )

  )
)
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**