"FuxCP: A Constraint Programming Based Tool Formalizing Fux's Musical Theory of Counterpoint"

Wafflard, Thibault

ABSTRACT

This master's thesis presents FuxCP, a tool for computer-aided contrapuntal composition. The objective is to assist composers without programming skills by automating repetitive and time-consuming tasks. The tool is based on constraint programming with Gecode and formalizes musical rules as constraints. Thanks to this approach, the tool provides transparency and control over the generated solutions, allowing composers to shape their desired music. This thesis focuses on formalizing the rules of two-voice counterpoint from Fux's Gradus ad Parnassum. The research highlights the advantages of constraint programming over other approaches, as it allows the tool to "understand" the generated music. The thesis covers the formalization of counterpoint species-specific rules as mathematical constraints, the evaluation of the tool compared to Fux, and suggestions for future development. The conclusion emphasizes the importance of a comprehensive set of rules for formalization, the need for additional constraints on melodic development, and the potential for more expert solvers in other musical genres. The findings indicate the potential of constraint programming in enhancing computer-aided composition across various musical styles.

CITE THIS VERSION

École polytechnique de Louvain

FuxCP: A Constraint Programming Based Tool Formalizing Fux's Musical Theory of Counterpoint

Author : **Thibault Wafflard**
Supervisor : **Peter Van Roy**
Readers : **Yves Deville**, **Karim Haddad**, **Damien Sprockeels**

Academic year 2022–2023
Master [120] in Computer Science

Abstract

This master's thesis presents FuxCP, a tool for computer-aided contrapuntal composition. The objective is to assist composers without programming skills by automating repetitive and time-consuming tasks. The tool is based on constraint programming with Gecode and formalizes musical rules as constraints. Thanks to this approach, the tool provides transparency and control over the generated solutions, allowing composers to shape their desired music. This thesis focuses on formalizing the rules of two-voice counterpoint from Fux's *Gradus ad Parnassum*. The research highlights the advantages of constraint programming over other approaches, as it allows the tool to "understand" the generated music. The thesis covers the formalization of counterpoint species-specific rules as mathematical constraints, the evaluation of the tool compared to Fux, and suggestions for future development. The conclusion emphasizes the importance of a comprehensive set of rules for formalization, the need for additional constraints on melodic development, and the potential for more expert solvers in other musical genres. The findings indicate the potential of constraint programming in enhancing computer-aided composition across various musical styles.

Acknowledgements

Thanks to Peter Van Roy, my supervisor, for giving me the opportunity to write this thesis combining two of my passions: computer science and music.

Thanks to Damien Sprockeels for his pragmatic feedback, his devotion to the project, and his great contribution to computer music.

Thanks to Karim Haddad from IRCAM for proposing the *Gradus ad Parnassum* as a reference book.

Thanks to Yves Deville for reading this thesis.

Thanks to UCLouvain for allowing me to finish my studies on such an instructive project.

Thanks to Justine Nagant for supporting me throughout this long ordeal.

It is thanks to the energy and time of all these people that this thesis exceeded my expectations.

Contents

Introduction

This thesis is part of a long-standing project between UCLouvain and IRCAM (*Institut de Recherche et Coordination Acoustique/Musique*)[1]. It is one more step towards a complete tool capable of creating music from a musical style to assist composers without programming skills by automating repetitive, arduous, or time-consuming tasks. On the technical level, it is a question of formalizing musical rules in discrete mathematics in order to represent them in the form of constraints. The tool uses constraint programming (CP) to find solutions satisfying previously chosen musical rules. It is a composition assistant and not an independent composer. Ideally, the tool serves as a first draft for the composer so that he can adapt the solution or seek a new solution that better meets his needs.

Currently, this tool is achieved through the use of Gecode[2] for the CP part and OpenMusic[3] (OM) for the user interface. Gecode is an open-source C++ toolkit for developing constraint-based systems and applications. While OM is an open-source Lisp graphical programming environment for music composition. The link between Gecode and OM is done via the wrapper GiL[4].

But why constraint programming? It is an innovative approach in the field of music computing. Today, there are several machine learning (ML) models capable of reproducing certain styles of music more or less faithfully (e.g. MuseNet by OpenAI[5] or Music Transformer by Magenta[6]). Unlike machine learning, constraint programming allows full transparency on how solutions are generated and full control over the musical rules that apply[1]. In a way, this paradigm allows the model to "understand" the music it generates. This is practical to leave to the composer the possibility of making the music he truly wants. CP is a very powerful paradigm still under-exploited in the field of computer music allowing to generate musically correct solutions in a few seconds, or even less.

Before embarking on an overly complex formalization of music, it is necessary to prove that this kind of tool is feasible with a musical style that is not too complex and fairly strict. It is for this reason that this thesis focuses on the formalization of the rules of Johann Joseph Fux's *Gradus ad Parnassum*[8] on two-voice counterpoint. Counterpoint is a musical style, mostly developed during the Baroque era, which consists of having several melodies played at the same time[9]. These melodies are harmonically interdependent but melodically independent[10]. They are all built from a *cantus firmus,* i.e. the "given song" which determines the context of the musical piece. It is a horizontal construction of music which is partly opposed to the common harmonic vertical approach.

The *Gradus ad Parnassum* is a good book to start this project for several reasons. First, this work is recognized as a pillar of the theory of contrapuntal composition. These rules are therefore still applied by contemporary composers. Second, Fux's work is separated into several chapters according to the species of counterpoint. With-

[1]Opacity in ML is a recurring problem that is still one of the main challenges today. For example, Ferreira and Whitehead [7] explain that "it is very hard to control such models in order to guide the compositions towards a desired goal".

out going into details, this allows to formalize each type of counterpoint iteratively without getting confused.

To summarize, the purpose of this thesis is, on the one hand, to create a tool capable of generating a counterpoint based on a *cantus firmus,* on the other hand, to determine the advantages and disadvantages of CP for composition computer-aided.

This work will be divided into several parts: it will first briefly present the work taking place in the vast field of computer-assisted composition. Then, the musical and technical knowledge for a good understanding of the work will be established. Of course, most of this paper consists in formalizing in the form of mathematical constraints the different rules that can be extracted from Fux's work. For this, a chapter will be devoted to the system of variables on which all the constraints are created. There will follow a chapter by species detailing each rule in natural language and then in mathematical constraints. Before concluding, the evaluation of all species will be done to determine if the CP approach was realistic or not. Finally, criticisms and suggestions will be given in order to continue this big project as well as possible.

Related Work

Before getting to the heart of the matter, it is important to underline the existence of work related to the field of computer music. Indeed, there are already some works using approaches relatively similar to that of this paper. It will then quickly describe some examples of work in the field of ML to explain problems related to this approach that can be partially solved with constraint programming. Finally, the main references of this thesis will be given to get an idea of its writing environment.

About Counterpoint and Constraint Programming

First, in 1984, Schottstaedt wrote *Automatic Species Counterpoint*[11]. His work describes an expert system, i.e. a sequence of if-then statements representing the rules of *Gradus ad Parnassum*. There are several similarities with the present work. Where an expert system represents knowledge through if-then statements[12], the CP makes it possible to represent this knowledge in a more complete and mathematical way. Moreover, the inference engines of the 80s were not as smart as the algorithm used in the present work. Indeed, his searches are more linear in their choice by running more standard algorithms such as "best-first search".

While his work is a precursor of this thesis, it did not really influence this thesis but rather demonstrates certain issues still present today. For example, Schottstaedt uses a penalty list system to represent preferences between several choices. As will be explained later, there is a relatively similar system for representing the cost of certain situations in this formalization. The results of his report were poor given the number of mediocre solutions that the expert system offered and the amount of rules put in place.

Secondly, in *An Interactive Constraint-Based Expert Assistant for Music Composition*, Ovans and Davison [13] attempted the same approach by using the CP to represent the first species of counterpoint (comprising only whole notes). Their goal was that the user could interact with the graph representing the search space. It was therefore a tool where the direction of the search was partly directed by a human. It is an interesting approach but different from that of this paper which consists rather in making the user interact via the preference of certain musical notions. Moreover, the first species of counterpoint is not sufficient to form an opinion of the use of CP in musical creation. However, it is quite interesting to see that their conclusions have great similarities with those described at the end of this thesis.

The subject is not recent and several readings can be recommended for those wishing to learn more. For example, Pachet and Roy [14] describe musical harmony as constraints. Or, more recently, Sandred [15] establishes the different researches and constraints satisfaction problem solvers over time.

About Machine Learning

In terms of machine learning, most of the research is more recent. As explained in the introduction, several music generation products/models already exist such as OpenAI's MuseNet and Magenta's Music Transformer. These are therefore technologies that are extremely different from the one presented in this paper. These models also aim to generate music but do not necessarily aim to be highly configurable tools by their users. These technologies can be useful in other contexts, which is beyond the scope of this thesis. However, the analysis of Briot and Pachet must be highlighted because it explains well the issues encountered with this technology:

> "[A] direct application of deep learning to generate content rapidly reaches limits as the generated content tends to mimic the training set without exhibiting true creativity. Moreover, deep learning architectures do not offer direct ways for controlling generation (e.g., imposing some tonality or other arbitrary constraints). Furthermore, deep learning architectures alone are autistic automata which generate music autonomously without human user interaction, far from the objective of interactively assisting musicians to compose and refine music." Briot and Pachet [16].

Also, one of the researchers who worked on the Music Transformer[17] tried to generate counterpoints by convolution [18]. The results are not very convincing but acceptable. The problem, in this case, is that the model is trained from a fairly limited database which prevents the model from getting out of its comfort zone. All the approaches remain interesting knowing that these technologies are not enemies because they can collaborate to fill the weaknesses of one another.

About this Thesis

Finally, this thesis is based on the work of previous years:

- Lapière [19] presented an interface for using Gecode functions in Lisp called GiL. He tested it with some rhythm-oriented constraints.
- Sprockeels [20] explored the use of constraint programming in OpenMusic using GiL. He made a tool capable of producing songs with basic harmonic and melodic constraints.
- Chardon, Diels, and Gobbi [21] created a tool capable of combining the strengths of the first two implementations while continuing to develop support for GiL.

This thesis is the first to be a "complete" representation of a particular musical style. As explained above, Fux's *Gradus ad Parnassum* was chosen as the basis for the work. The original text dates from 1725 and is written in Latin. Nobody on the research team speaks this dead language so it's obvious that translations were used instead. Two of them have been particularly used: the first is that of Chevalier [22], a French translation dating from 2019. It is from this work that the rules have been taken. The second is Mann [23]'s English translation dating from 1971. This one is interesting because it includes footnotes to better understand certain ambiguous rules. It is also from this work that most of the quotations are taken.

Chapter 1

Theoretical Background

A complicated point with this kind of subject is the vast field of knowledge that composes it. Indeed, to be able to fully understand the rest of the thesis, it is necessary to have a good knowledge of musical theory rather than an in-depth knowledge of constraint programming. Music theory is very rich and applies differently from one culture to another, while CP is a younger field and easier to popularize broadly. For these reasons, it will be tempted to explain in more depth the sometimes ambiguous musical notions.

The operation of Gecode and OpenMusic is not necessary for understanding the paper, so they do not have their own sections in this chapter. To learn more about these subjects, there is the documentation of Gecode[24], OpenMusic[25], and Sprockeels [20]'s thesis which covers the essence of these two tools.

Before moving on to formalization, it is important to recall the fundamentals of music and its notation. The technical theory of constraint programming will be explained right after.

1.1 Music Theory

1.1.1 Concept of Counterpoint

As explained in the introduction, counterpoint (ctp.) is a style predominantly cultivated during the Baroque period. Basically, it involves the simultaneous performance of multiple melodies[9]. These melodies exhibit melodic independence while maintaining harmonic interdependence[10].

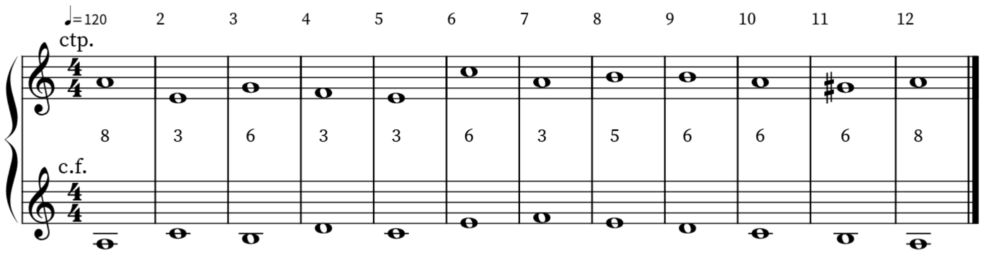


Figure 1.1: Example of a 1st species ctp. Score available here [26] and listen here [27].

More precisely, each melodic line functions autonomously, possessing its own distinct melodic character, rhythm, and contour. These individual voices are carefully crafted to interact with one another harmonically. While the melodic lines intertwine and intersect, they maintain their melodic independence, allowing each voice to be perceived as a distinct entity within the overall musical composition.

A key element in the construction of counterpoint is the use of a *cantus firmus,* which serves as a foundational melodic line or a "given song." It establishes the melodic and harmonic framework within which the additional voices are developed. Composers use the *cantus firmus* as a point of departure, building intricate melodic structures around it while adhering to specific rules and guidelines governing melodic and harmonic interactions.

In 1725, Fux [8], a renowned music theorist of the Baroque era, outlined a systematic approach to counterpoint in his influential work, *Gradus ad Parnassum*. He categorized counterpoint into five distinct species, each with its own set of rules and characteristics. These species are mainly recognizable by their rhythms. They are built iteratively on top of each other, so that the rules of the first species apply in part in the second species, up to the fifth one.

- First Species: Each note of the added voice corresponds to a single note of the *cantus firmus.* The goal is to maintain a strict one-to-one relationship between the voices, ensuring that no dissonances occur.

- Second Species: It involves adding two notes in the counterpoint voice for each note of the *cantus firmus.* The main added rule is that of allowing dissonant harmonies.

- Third Species: Four notes are played in the counterpoint voice against each note of the *cantus firmus.* This species introduces more movement and possibilities in the way the melody is handled.

- Fourth Species: Mainly composed of syncopations, it focuses on rhythmic displacement and anticipation. The counterpoint voice introduces syncopated rhythms by placing notes on weak beats.

- Fifth Species: Also known as the florid counterpoint, it combines elements of the previous four species. It allows for greater freedom in the use of note durations, rhythmic patterns, and melodic embellishments. This species showcases the composer's skill in crafting intricate and ornate melodic lines while maintaining the fundamental principles of counterpoint.

Figure 1.2: Example of a 5$^{\text{th}}$ species ctp. Score available here [28] and listen here [27].

Don't worry, more examples will be shown in due course.

1.1.2 Equivalent American vs British English Terms

Depending on the reader, terms used in music may vary between America, England, and translations. Here is a summary of the equivalent terms depending on the language:

- Measure ≡ Bar
- Whole step ≡ Tone
- Half step ≡ Semitone
- Whole note ≡ Semibreve
- Half note ≡ Minim
- Quarter note ≡ Crotchet
- Eighth note ≡ Quaver
- Sixteenth note ≡ Semi-quaver

1.1.3 Music Concepts

The following definitions are mainly there to help if the reader does not fully understand the nuances between certain terms in the next chapters. The definitions are sorted so that the first are global and the last are on more specific points.

Staff A staff refers to the set of horizontal lines and spaces upon which musical notes and symbols are written on scores. The staff typically consists of five lines and four spaces, with each line and space representing a specific pitch (see figure 1.3).

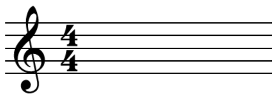


Figure 1.3: Staff with a treble clef (*clef de Sol*), an empty key signature and a 4/4 time signature.

Note On sheet music, a note is a symbol used to represent a specific pitch and duration of a sound. Notes are written on staff and can be represented by a variety of symbols. Generally speaking, a note refers to a certain frequency played at a certain time.

Beat A beat is the underlying pulse that organizes the passage of time within a musical composition. It serves as a fundamental unit of measurement, establishing the division of time into equal segments. The beat provides a sense of stability and acts as the rhythmic foundation upon which melodies, harmonies, and other musical elements are built.

Measure A measure is a section of music that is delimited by vertical bar lines in sheet music. With a common 4/4 time signature, a measure is made up of four beats.

Pitch Pitch refers to the highness or lowness of a sound. Pitch is determined by the frequency of the sound wave and is measured in hertz (Hz). Higher pitched sounds have a higher frequency than lower pitched sounds.

MIDI *Musical Instrument Digital Interface* is a standard protocol for communication between musical instruments and computers. What is commonly called "MIDI values" refers to the different possible MIDI notes ranging from 0 ($C_{-1} \equiv 8.175799\ Hz$) to 127 ($G_9 \equiv 12543.85\ Hz$)[29]. The notes of an 88-key piano are limited to A_0 to C_8. The list of MIDI values can be found in table B.1.

Semitone A semitone, also known as a half step, is the smallest interval (the distance between two notes) in Western music. It represents the distance between two adjacent notes on a keyboard or guitar.

Step A step is a melodic interval of one semitone (minor second) or one tone (major second)[30] between two consecutive notes of a musical scale[31]. Melodies that move by steps are *stepwise*.

Types of notes Within a common 4/4 time signature (see figure 1.4):

- A **whole note** represents a long duration of sound and lasts four beats.
- A **half note** represents a medium duration of sound and lasts two beats.
- A **quarter note** represents a short duration of sound and lasts one beat.

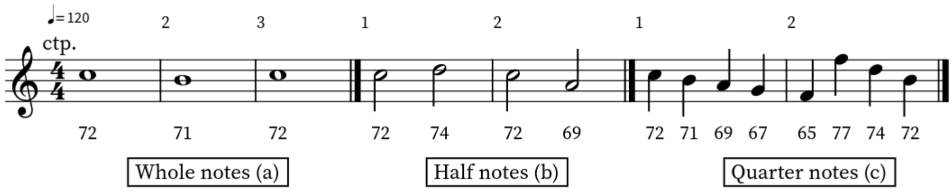


Figure 1.4: The 3 main types of notes used in the counterpoint.

Syncopation The displacement of the main beat of a measure. It creates an off-balance rhythm through the accenting of normally unaccented beats.

Mordent A mordent is a type of ornament referring to a quick alternation between a note and its upper (upper mordent) or lower neighbor (lower/inverted mordent)[32].

Intervals In Western tonal music, the intervals making up an octave are separated into 12 semitones. Table 1.1 shows the MIDI values corresponding to these intervals.

Interval	Unison/Octave	Second		Third		Fourth	Tritone	Fifth	Sixth		Seventh	
Type	Perfect	Minor	Major	Minor	Major	Perfect	$\sharp 4^{th}$ / $\flat 5^{th}$	Perfect	Minor	Major	Minor	Major
Value	0	1	2	3	4	5	6	7	8	9	10	11

Table 1.1: MIDI values of the intervals over an octave range.

Tonic The tonic is the first note of a scale and serves as the foundation or the "home" for the other notes in the scale. It is this note that gives the name of the scale.

Scale A scale is a series of intervals arranged in ascending or descending order. The most common scales in Western music are the major and minor scales. Each scale has a unique pattern of whole and half steps between the notes.

Key A key refers to a specific scale and tonic. For example, a piece of music in the key of C major would use the major scale and have C as the tonic. The key of a piece of music determines the overall tonality and harmony of the piece.

Mode A mode is a type of scale that can be seen as a derivation from a parent scale. Basically, they are alternatives to common scales so that the tonic and the other note functions have been shifted. The modes in Western music are the Dorian, Phrygian, Lydian, Mixolydian, Aeolian, and Locrian modes. Like any scale, each mode has a unique pattern of whole and half steps between the notes.

Diatonic A diatonic scale is a scale made up of seven different pitches, where each pitch corresponds to a letter in the musical alphabet (A, B, C, D, E, F, G). A note is considered diatonic if it belongs to the key range of the piece.

Chromatic A chromatic scale is a scale that includes all the notes separated by a semitone. A chromatic scale contains 12 notes in total, including all the notes in a diatonic scale and additional notes between each of the diatonic scale notes. Chromatic notes are often used to add dissonance or tension to a piece of music.

Borrowed note A borrowed note is a non-diatonic note borrowed from another key or mode and used temporarily in a piece of music. Borrowed notes can be used to add variety and interest to a melody or harmony. They can also be used to create a sense of tension or dissonance, which can then be resolved back to the original key or mode.

Degree A degree is the relative position of a note in a scale to the tonic. By default, one degree aside from a note is the closest next note available in the diatonic scale. A degree can be expressed for both melody and harmony (even as chords). The degrees make it possible to understand and convert any tonality through a relative system[33]. By convention, they are written with Roman numerals from I (the tonic) to VII (the sensible). For example, in C major, C (i.e. the tonic) is the I degree while G (i.e. the dominant, the fifth) is the V degree. Transposed to F major, this would give F the I degree and C the V degree. Also, melodies that progress by joint degrees are equivalent to stepwise melodies.

Thesis Aka downbeat. With a common 4/4 time signature, the thesis is the first beat of any measure.

Arsis Aka upbeat. With a common 4/4 time signature, the arsis is the third beat of any measure.

Skip The melodic interval which, unlike the step, is greater than one tone. The term is rather used to refer to the third melodic interval because it is equivalent to *skip* a key on a piano but no convention exists. "Leap" can therefore also be used for the same purpose.

Leap The melodic interval which, unlike the step, is greater than one tone. The term is rather used to refer to melodic intervals larger than a third in contrast with the term "skip". Although, no convention exists so "skip" can also be used for the same purpose.

Diminution An intermediate note that exists between two notes separated by a skip of a third. In other words, a note that fills the space in third skip. This intermediate note is not necessarily below the previous one. Actually, the term refers to the division of a note into several shorter ones (i.e. "passage notes")[34].

1.2 Constraint Programming Prerequisites

This thesis is more focused on mathematical formalization than on the extensive use of constraint programming. Indeed, as will be explained later, optimization is not a point that was particularly highlighted during this work. The limits of Gecode were not reached within the framework of this work. There was thus no need to investigate in that specific direction. Despite this, it is still important to understand what constraint programming is.

1.2.1 Constraint Programming Concept

Constraint programming is an approach to solve complex combinatorial problems by specifying them as logical relations, called constraints[35]. This kind of problem is called *constraint satisfaction problem* or CSP. It is solved by using a combination of inference and search. CP is a powerful paradigm in the field of computer science that addresses complex decision-making and optimization problems.

Problems are represented as a set of variables, each with a domain of possible values, and constraints that define the allowable combinations of values for these variables. Constraints capture logical relationships between variables, reflecting the problem's requirements.

The solver part has to find a solution that satisfies all the given constraints by determining the values for the variables that do not violate previously posted constraints. In practice, CP engines employ constraint propagation techniques to enforce the constraints and reduce the search space by propagating the effects of variable assignments.

1.2.2 Branching

Branching refers to the selection of variables and their values during the exploration of the solution space. It involves selecting a variable and dividing its domain into smaller subsets or branches, each representing a possible assignment.

When solving a CSP with Gecode, there are two fundamental search strategies employed: depth-first search and branch-and-bound. These strategies are guided by variable selection heuristics and value selection heuristics. Without going into details, depth-first search assigns values to variables regarding the heuristics and then backtracks when constraints cannot be satisfied. The branch-and-bound strategy extends depth-first search by incorporating an additional mechanism to post a new constraint specified in advance[1].

The choice of variable for branching is guided by variable selection heuristics, which determine the order in which variables are considered during the search process. These heuristics aim to select the most promising variable at each step, leading to faster convergence. Common variable selection heuristics include selecting the variable with the fewest remaining values in its domain (minimum domain size) or selecting the variable involved in the most constraints (maximum degree). Value selection heuristics determine the order in which values are assigned to variables. These heuristics aim to prioritize values that are more likely to lead to a successful solution.

1.2.3 Advantages

To summarize the benefits that lead to the use of CP, it can be said that it mainly stands out for its expressiveness, transparency, flexibility, and efficiency. CP allows problem solvers to express problems in a natural manner, enabling direct communication of knowledge and requirements. It provides transparency in the problem-solving process, making search algorithms and strategies explicit and understandable. CP supports flexible problem modeling, allowing the incorporation of additional constraints, objectives, and problem-specific knowledge. It can efficiently explores the solution space using constraint propagation and search techniques, reducing the search space and quickly discarding infeasible solutions. These advantages make CP valuable for addressing a wide range of real-world decision-making and optimization problems.

[1] With the current version of GiL, it has never been possible to make branch-and-bound work from Lisp. Several attempts were tested but all failed.

Chapter 2

Introduction to the Formalization of Fux's Theory

The formalization of Fux is done in several steps:

1. **Spot the right rules in the *Gradus Ad Parnassum*.** Fux tended to explain certain rules of music so that they were easy to understand and use for the musicians of the time. This implies that sometimes several rules can be reduced to one. On the other hand, some of the rules of music are not written as such in the book because they are implicit. For example, it goes without saying that counterpoint belongs to a certain key and scale, but this is never explicitly written in the book. In order not to create misunderstanding, it was decided to write them explicitly and separately in the next sections.

2. **Formalize the rules in natural language in a way that is easy to express as constraints.** Indeed, the *Gradus Ad Parnassum* is a work dedicated to a 17th century audience. It is necessary to read it with a critical eye and to translate it into modern language. That is, to reduce several rules into one, or at times, some rules are expressed in inclusive terms, whereas it is easier for a mathematician or computer scientist to write them in an equivalent way with exclusive terms or vice versa. Examples will be given in section 3.1.

3. **On the one hand, write the rules in discrete mathematics.** This is a crucial step in order to be able to use these rules precisely in other contexts and with other programming languages. This will also allow us to check whether solutions exist mathematically. Indeed, some rules may be contradictory and, consequently, no solution would be possible. It is important to keep in mind that some rules are written in a way that can be easily written with the Gecode tool.

4. **On the other hand, write the rules in constraint programming language.** The final goal of this thesis is to have constraints fixed according to Fux's rules and to find the best possible solutions with Gecode.

2.1 Array Logic and Notation

It is particularly important to understand how arrays are constructed. The rest of the paper relies entirely on the nuances and particularities of this logic and notation. This section is intended for mathematicians and computer scientists.

2.1.1 Logic of the arrays

The majority of the variables are arrays representing "in order" the different constrained values linked to the solution. The solution to the problem is an array of MIDI notes lists representing counterpoint. Before starting, two constants[1] must be defined[2]:

- m as the number of measures of the *cantus firmus* and the counterpoint;
- s_m as the maximum number of notes possible in the counterpoint, i.e. the size of the main arrays used to store Gecode variables. $s_m = m + 3 \times (m-1)$ and by extension, $s_{m-1} = (m-1) + 3 \times (m-2)$.

Intuitively one would separate an array into m lists of each measure with the different notes of a measure inside. Here the reverse applies. With a C-like representation, the access to a variable will be done as $[beat][measure]$ instead of $[measure][beat]$. This is more convenient for applying constraints in Lisp with GiL. Indeed, since the number of beats used by species varies, it is then easier to separate the arrays by lists of beats in order to be able to initialize only those which are treated in the problem. Since these arrays are initialized not with simple integers but with `IntVar` objects from Gecode, these constraint variables would definitely be initialized in the constraint space, which would not be ideal.

All the arrays related directly to the counterpoint are stored in arrays of size s_m (or s_{m-1} for the melody arrays as will be explained later). These arrays are composed of four lists, each representing the corresponding beat all along the measures of the song. The first is of size m while the other three are of size $m-1$ since they do not have a note in the last measure of the counterpoint which is only composed of a single whole note. E.g. `notes[0][9]`[3] would represent the note in the first beat of the tenth measure.

If the chosen species of counterpoint uses only **whole notes**, i.e. the first one, each note in first beat of each measure lasts **four beats**. Consequently, the lists of notes in the second, third and fourth beats are not used because these notes would already be represented by the one in first beat. The same logic applies to the other species: the second and fourth species only use the **first** and **third** "beat lists" because a note lasts **two beats**. While the third and fifth species are the only ones to use the **four available** beat lists because a note (can) last(s) **one beat**. See figure 2.1 (the corresponding midi value is annotated below each note) and table 2.1 for clarity.

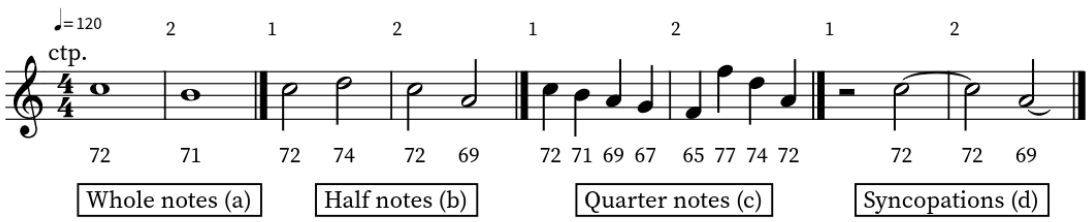


Figure 2.1: The 3 types of notes (N.B.: b ≡ d) over 8 beats for the 4th first species.

Syncopations have been added to illustrate that they work in the same way as half notes. The fifth species repeats the first four ones so it is not shown here. It will be explained in detail in chapter 7.

[1]Careful, these are constants from the point of view of the Gecode solver. They are variables defined once with the input but which are never set as constraint variables in the CSP.

[2]These constants are defined more precisely in subsection 2.2.1.

[3]This array exists only as an example. Here the notation corresponds neither to Lisp notation nor to mathematical notation.

beat, measure	$1^{st}, 1^{st}$	$2^{nd}, 1^{st}$	$3^{rd}, 1^{st}$	$4^{th}, 1^{st}$	$1^{st}, 2^{nd}$	$2^{nd}, 2^{nd}$	$3^{rd}, 2^{nd}$	$4^{th}, 2^{nd}$
Whole notes	72	$\emptyset$	$\emptyset$	$\emptyset$	71	$\emptyset$	$\emptyset$	$\emptyset$
Half notes	72	$\emptyset$	74	$\emptyset$	72	$\emptyset$	69	$\emptyset$
Quarter notes	72	71	69	67	65	77	74	72
Syncopations	$\emptyset$	$\emptyset$	72	$\emptyset$	72	$\emptyset$	69	$\emptyset$

Table 2.1: Relative MIDI values of figure 2.1.

2.1.2 Notations of the arrays

Several notations exist to describe the elements of an array. The one chosen here is close to the computer notation with the indexing starting at zero.

- $A[i, j]$ for element j of list i of array A;
- $L[i]$ for element i of list L;
- $A[i]$ for list i of array A.

Note that another way is also used to represent all the positions of a table. Indeed, as it is shown in the previous subsection, an array representing all measures per beat can be merged as a long list representing all beats one after the other. Therefore, to clarify the notation, $\forall \rho \in positions(m)$ will be used to represent all non-empty positions of an array. For example, for the half notes in the previous table 2.1: $\rho \in \{[0,0],[2,0],[0,1],[2,1],\dots\}$. Moreover for notational purposes, $\rho + 1$ will denote the position of the next note such that if $A[\rho] = A[0,0]$ then $A[\rho+1] = A[2,0]$. To explain it properly, the set $\mathcal{B}$ and the constants b and d must be introduced.

$\mathcal{B}$ Set of beats in a measure used by the solver depending on the chosen species. $\mathcal{B}$ can be seen as the location or index of the notes written over a measure on a score.

$$\mathcal{B} = \begin{cases} \{0\} & \text{if species = 1} \\ \{0,2\} & \text{if species = \{2, 4\}} \\ \{0,1,2,3\} & \text{if species = \{3, 5\}} \end{cases} \tag{2.1}$$

This refers back to the previous table 2.1.

b Number of beat(s) in a measure used by the solver depending on the chosen species. b can be seen as the number of notes written over a measure on a score. b is related to $\mathcal{B}$ since $b = |\mathcal{B}|$.

$$b = \begin{cases} 1 & \text{if species = 1} \\ 2 & \text{if species = \{2, 4\}} \\ 4 & \text{if species = \{3, 5\}} \end{cases} \tag{2.2}$$

d Duration of a note in beat(s) depending on the chosen species. d can be seen as the space between the notes of a measure on a score. d is inversely proportional to b.

$$d = 4/b$$
$$\therefore d = \begin{cases} 4 & \text{if species = 1} \\ 2 & \text{if species = \{2, 4\}} \\ 1 & \text{if species = \{3, 5\}} \end{cases} \tag{2.3}$$

positions(upto) Function that returns the set of non-empty positions or indexes ordered depending on the species in such a way that all the positions would follow one another to represent all the beats of that species on a score in a single list.

$$\begin{aligned} positions(upto) = \bigcup_{\forall i \in \mathcal{B}, \forall j \in [0, upto)} [i, j] \\ \text{s.t. } \forall x \in [1, 3], \forall y \in [1, upto) \\ [i, j] <_s [i + x, j] <_s [i, j + y] \\ \text{where } <_s \text{ means the sorting order} \end{aligned} \tag{2.4}$$

By extension, $\rho + z >_s \rho$ such that:

$$\begin{gathered} \forall z \in \mathbb{N}^+, \forall \rho = [i, j] \in positions(upto) \\ \rho + z = [i + zd, j + nextm(i + zd)] \\ \text{where } nextm() \text{ is a function that returns the correct number of measure(s) to add.} \end{gathered} \tag{2.5}$$

2.2 Definitions of the Constants, Costs, Variables and Functions

This section is more intended for mathematicians and computer scientists too. Those who don't wish to read the mathematical parts should still broadly understand the variables of harmonic intervals, melodic intervals and motions (**H**, **M** and **P** in section 2.2.3). Subsections 2.2.1 and 2.2.3 describes the various names used in the mathematical parts and in the Lisp code of the solver (immediately to their right, e.g. **n** `*total-cp-len`). These subsections explain also how those constants and variables work. Unless otherwise stated, all domains of constants and variables belong to the domain of integers $\mathbb{N}$.

2.2.1 Constants

Constants are only constant with respect to the Gecode solver, so they are deduced before a solution is sought by the latter.

Cons $_{(all,\ p,\ imp)}$ `ALL_CONS`, `P_CONS`, `IMP_CONS`

Set representing all consonances, perfect consonances and imperfect consonances respectively. By default, the notation $Cons \equiv Cons_{all}$.

$$\begin{aligned} Cons_p &:= \{0, 7\} \\ Cons_{imp} &:= \{3, 4, 8, 9\} \\ Cons_{all} &:= Cons_p \cup Cons_{imp} \equiv \{0, 3, 4, 7, 8, 9\} \end{aligned} \tag{2.6}$$

species `species`

Chosen species of counterpoint. $species \in \{1, 2, 3, 4, 5\}$.

m `*cf-len`

Number of measures which is equivalent to the number of notes in the *cantus firmus*. $m \in [3, 17]$. 3 because the solver needs al least 3 measures to work properly. 17 is arbitrary and comes from $4 \times 4 + 1$, i.e. a commun number of measure $\times$ a number not too large for the computation + one final measure.

n `*total-cp-len`

Number of notes in the counterpoint depending on the chosen species. $n \in [1, b(m-1)+1]$ because the last measure has necessarily a whole note.

s_m Maximum number of notes contained in the counterpoint, all species combined, i.e. if the counterpoint contained only quarter notes, with the exception of the last note being a whole note.

$$s_m = m + 3 \times (m-1) \text{ and } s_{m-1} = (m-1) + 3 \times (m-2) \tag{2.7}$$

Used as the size for an array containing one list of size m (or $m-1$) the notes in thesis and three lists of size $m-1$ (or $m-2$) the other beats. The difference with n is that s does not depend on b.

Cf `*cf`

List of size m representing the MIDI notes of the *cantus firmus.*

$$\begin{gathered} \forall j \in [0, m) \\ Cf[j] \in [0, 127] \end{gathered} \tag{2.8}$$

M_{cf} `*cf-brut-m-intervals`

List of size $m-1$ representing the melodic intervals between the consecutive notes of the *cantus firmus.*

$$\begin{gathered} \forall j \in [0, m-1) \\ M_{cf}[j] = Cf[j+1] - Cf[j] \\ \text{where } M_{cf} \in [-127, 127] \end{gathered} \tag{2.9}$$

lb `RANGE_LB`

Lower bound of the range of the notes of the counterpoint. $lb \in [0, ub)$.

ub `RANGE_UB`

Upper bound of the range of the notes of the counterpoint. $ub \in (lb, 127]$.

$\mathcal{R}$ `*cp-range`

Range of the notes of the counterpoint. $\mathcal{R} := [lb, ub]$.

borrow *DFLT: <major>*[4]

Determines the "borrowing scale", i.e. the additional notes that the counterpoint can have in relation to the tonic of the piece. More details will be given on what are the borrowed notes in section 2.3.1.

$$borrow \in \{none, major, minor\} \tag{2.10}$$

[4] *DFLT: <value>* means the default value in the tool.

$\mathcal{N}^{(\mathcal{R})}_{(all,\ key,\ brw)}$ `*extended-cp-domain`, `*scale`, `*borrowed-scale`.

Set of values available for the notes of the counterpoint. $\mathcal{N}_{key}$ represents the notes of the key provided by the user's score. $\mathcal{N}_{brw}$ represents the additional borrowed notes that the counterpoint can have in relation to the tonic of the piece. $\mathcal{N}_{all}$ represents the union of the two previous sets. If $borrow = none$ then $\mathcal{N}_{brw} = \emptyset$ and $\mathcal{N}_{all} = \mathcal{N}_{key}$. $\mathcal{N}^{\mathcal{R}}_{(all,\ key,\ brw)}$ represents the set of notes bounded to the range, i.e. the intersection of $\mathcal{N}_{(all,\ key,\ brw)}$ and $\mathcal{R}$. By default, $\mathcal{N}$ refers to $\mathcal{N}_{all}$ not bounded to the range.

$$\begin{gathered}
\mathcal{N}_{key} := buildScale(key, scale) \\
\mathcal{N}_{brw} := \begin{cases} \emptyset & \text{if } borrow = none \\ buildScale(Cf[0] \ mod \ 12, \text{"}borrowed\text{"}) & \text{if } borrow = major \\ buildScale([Cf[0] + 3] \ mod \ 12, \text{"}borrowed\text{"}) & \text{if } borrow = minor \end{cases} \\
\mathcal{N}_{all} := \mathcal{N}_{key} \cup \mathcal{N}_{brw} \\
\mathcal{N}^{\mathcal{R}}_{(all,\ key,\ brw)} := \mathcal{N}_{(all,\ key,\ brw)} \cap \mathcal{R}
\end{gathered} \tag{2.11}$$

Where $buildScale(key, scale)$ (see function 2.24) is a function that returns the set of notes in the key based on the $scale$ used. Also more details on the borrowed notes will be given in section 2.3.1.

2.2.2 Costs

The costs are constants chosen by the user that have default values supposed to represent Fux's preferences.

pref and cost `*params*`

A preference can have 7 levels of intensity ranging from "no cost" to "forbidden". For any cost $cost$ and any preference $pref$, it can be defined that:

$$cost = \begin{cases} 0 & \text{if } pref = \text{no cost} \\ 1 & \text{if } pref = \text{low cost} \\ 2 & \text{if } pref = \text{medium cost} \\ 4 & \text{if } pref = \text{high cost} \\ 8 & \text{if } pref = \text{last resort} \\ 2m & \text{if } pref = \text{cost prop. to length} \\ 64m & \text{if } pref = \text{forbidden} \end{cases} \tag{2.12}$$

$64m$ is a ridiculously huge value that will never be reached by all the other costs combined even if they were all high.

Cond$_{costs}$ and cost$_{Cond}$ All costs work the same way: a list of boolean variables, called $Cond$ for the explanation, determines whether it is true that a certain cost should be established for this specific condition in certain locations. The list of assigned costs for this condition is noted $Cond_{costs}$. The elements of $Cond_{costs}$ are thus equivalent to any cost $cost$ depending on the preference $pref$ chosen for the condition $Cond$. The different costs for the different types of conditions each have their own identifier noted $cost_{Cond}$. It is this value that changes depending on the user's preference. To sum up:

$$\forall \rho \in Positions(Cond)$$
$$Cond_{costs}[\rho] = \begin{cases} cost_{Cond} & \text{if } Cond[\rho] \text{ is true} \\ 0 & \text{otherwise} \end{cases}$$
$$\text{where } Positions(Cond) \text{ is the set of positions where the condition } Cond \text{ applies}$$
$$\text{and where } cost_{Cond} \in dom(cost) \tag{2.13}$$

$\mathcal{C}$ and τ `*cost-factors`, `*total-cost`.

The heuristic of the solver leads to find a solution while minimizing the total cost. The latter is represented by τ while $\mathcal{C}$ is a set of integers representing all the sums of the different lists of costs. τ is thus the sum of all the elements of $\mathcal{C}$. If $Costs$ is the set of all the different $Cond_{costs}$ lists then:

$$\begin{gathered} \mathcal{C} = \bigcup_{\forall \chi \in Costs} \sum_{\forall c \in \chi} c \\ \tau := \sum_{\forall \sigma \in \mathcal{C}} \sigma \\ \min \tau \end{gathered} \tag{2.14}$$

By definition, for any forbidden $pref$ to be indeed *forbidden*, the following constraint must be added:

$$\sum_{\forall \sigma \in \mathcal{C}} \sigma < 64m \tag{2.15}$$

2.2.3 Variables

Variables are fully deduced by the Gecode solver and their values can be evaluated only after a solution has been found.

Many variables have a general definition so that they can be used in all equations, this does not mean that all possible combinations have been defined in the Lisp code but only those that are actually used. For example, there is no need to have access to all possible melodic intervals in the solver, however the mathematical notation would allow it.

If some letters are not defined, it means that they have already been defined in the constants or in the previous variables.

Cp `*cp`

Array of size s_m representing the MIDI notes of the counterpoint. This array is thus composed of four lists, each representing a beat on all the measures of the song. As explained above, this is how all the other arrays related to the countrepoint (i.e. the Cp array) are constructed.

For example, for a whole notes counterpoint: the relevant Cp would be only the list $Cp[0]$. For a half notes counterpoint: it would be the merge of $Cp[0]$ and $Cp[2]$. For a quarter notes counterpoint: it would be the merge of $Cp[0]$, $Cp[1]$, $Cp[2]$ and $Cp[3]$.

$$\forall i \in \mathcal{B}, \forall j \in [0, m) : Cp[i, j] \in \mathcal{N}^{\mathcal{R}} \tag{2.16}$$

Figure 2.2 shows a popularization of the tool's logic vis-à-vis these arrays of variables.

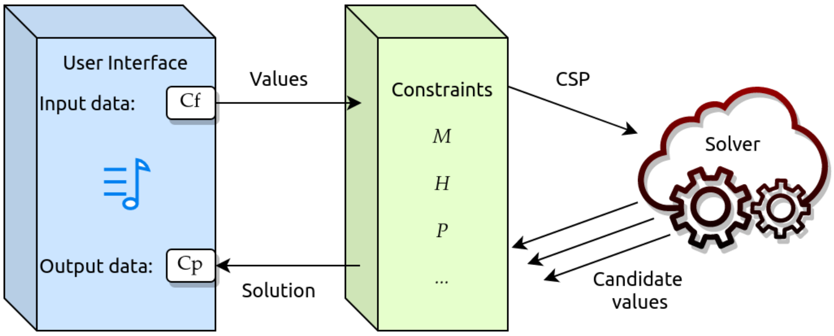


Figure 2.2: Popularization of the tool's logic vis-à-vis the arrays of variables.

$\mathbf{H}_{(abs)}$ `*h-intervals, *h-intervals-abs.`

Array of size s_m representing each harmonic interval between the counterpoint and the *cantus firmus.* There are four lists of harmonic intervals, each representing a beat along the whole counterpoint. The harmonic intervals are calculated so that they represent the absolute difference between the pitch of the counterpoint and the pitch of the *cantus firmus.* Since the values are absolute, it does not matter if the *cantus firmus* is lower or upper, the intervals will always be calculated according to the lowest note. Any harmonic interval is calculated according to the notes played at the same time in the *cantus firmus* and the counterpoint. Therefore, up to four notes in the counterpoint can be calculated with respect to the same note in the *cantus firmus.*

Two versions of that array-variable exist: the main one H which is modulo 12 and H_{abs} which is not. It is always true that $H = H_{abs} \ mod \ 12$. Unless mentioned, when talking about "harmonic intervals" or "harmonies", it refers to the variables of the array H.

$$
\begin{gathered}
\forall i \in \mathcal{B}, \forall j \in [0, m) \\
H_{abs}[i,j] = |Cp[i,j] - Cf[j]| \\
H[i,j] = H_{abs}[i,j] \ mod \ 12 \\
\text{where } H_{abs}[i,j] \in [0, 127], H[i,j] \in [0, 11]
\end{gathered}
\tag{2.17}
$$

12 representing the number of semitones in an octave. This allows the interval between a note and any note higher at different octaves to always be the same. This implies that $H \in$ table 1.1 values. For example, for the gap between C_4 (60) and G_4 (67) and the gap between C_4 (60) and G_5 (79), the H_{abs} values will be 7 and 19 while the H values will be 7 and 7.

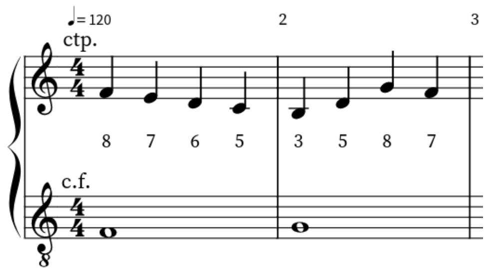


Figure 2.3: Traditionally written harmonies between the ctp. and the *cantus firmus.*

Beware that the numbers noted on figure 2.3 are those used on scores. They refer to the names of the intervals and not to the relative MIDI values. By contrast, table 2.2 below shows the MIDI values of the intervals for this figure.

measure j	$H_{abs}[0,j]$	$H_{abs}[1,j]$	$H_{abs}[2,j]$	$H_{abs}[3,j]$
0	12	11	9	7
1	4	7	12	10

Table 2.2: Relative MIDI values of figure 2.3.

$\mathbf{M}_{(brut)}^{(x)}$ `*m-intervals, *m-intervals-brut, *m2-intervals, ...`

Arrays of size s_{m-x} representing each melodic interval between a note of the counterpoint at a specific beat and another further note of the counterpoint at another specific beat. The melodic intervals are calculated so that they represent the difference between the two notes involved.

The array is noted M^x where x is the number of d[5] beat(s) that separates the initial note to the further one. x represents the desired number of notes between the current note and the one of interest to calculate the melodic interval. In other words, $M^x[i,j]$ represents the melodic interval between the note at beat i in measure j and the note at beat $[(i+xd)\ mod\ 4]$ in measure $[j+nextm(i+xd)]$. If x is not present then its default is 1. For example, with whole notes (i.e. $d=4$): $M[0,5]$ represents the melodic interval between the note in the sixth measure ($j=5$) and the note in the seventh measure ($j=6$).

There are two versions of that array-variable: the main one M^x which is absolute and M^x_{brut} which is not. It is always true that $M^x = |M^x_{brut}|$. Unless mentioned, when talking about "melodic intervals" or "melodies", it refers to the variables of the array M^1. See figure 2.4 (the corresponding midi value is annotated below each note) and table 2.3 for clarity.

$$\begin{gathered}
\forall x \in \{1,2\}, \forall i \in \mathcal{B}, \forall j \in [0, m-x) \\
M^x_{brut}[i,j] = Cp[(i+xd)\ mod\ 4, j+nextm(i+xd)] - Cp[i,j] \\
M^x[i,j] = |M^x_{brut}[i,j]| \\
\text{where } M^x_{brut}[i,j] \in [-12,12], M^x[i,j] \in [0,12]
\end{gathered} \tag{2.18}$$

The intervals are limited to 12 because the octave leap is the maximum that can be reached.

Figure 2.4: The 3 types of notes that can be used in the counterpoint.

In the solver, melodic intervals used are stored in several lists by beat pair, e.g. one list for all the intervals between the first and second beats of all measures. The constraints to represent these calculations are done separately from one table to another with the same function. From example, all the melodic intervals between the fourth beat note and the next first beat note in the thrid species are computed like in equation 2.19:

[5]Duration of a note in beat(s) depending on the chosen species (see d in above section 2.2.1).

$M^x_{(brut)}$	Whole notes (a)	Half notes (b)	Quarter notes (c)
$M[0,0]$	1 (-1)	2 (2)	1 (-1)
$M[1,0]$	$\emptyset$	$\emptyset$	2 (-2)
$M[2,0]$	$\emptyset$	2 (-2)	2 (-2)
$M[3,0]$	$\emptyset$	$\emptyset$	2 (-2)
$M[0,1]$	1 (1)	3 (-3)	12 (12)
$M^2[0,0]$	(0)	0 (0)	3 (-3)
$M^2[2,0]$	$\emptyset$	5 (-5)	4 (-4)

Table 2.3: Some relative MIDI values of figure 2.4 with $x = \{1, 2\}$.

$$\begin{gathered} \forall j \in [0, m-1) \\ M_{brut}[3,j] = Cp[0,j+1] - Cp[3,j] \\ M[3,j] = |M_{brut}[3,j]| \end{gathered} \tag{2.19}$$

P `*motions`

Array of size $4 \times (m-1)$ representing each motion between two consecutive measures. The letter P is for *passage* since M is already taken. Contrary, oblique and direct motions are represented by 0, 1 and 2 respectively.

$$\begin{gathered} \forall x \in \{1,2\}, \forall i \in \mathcal{B}, \forall j \in [0, m-1), x := b - i \\ P[i,j] = \begin{cases} 0 & \text{if } (M^x_{brut}[i,j] > 0 > M_{cf}[j]) \vee (M^x_{brut}[i,j] < 0 < M_{cf}[j]) \\ 1 & \text{if } M^x_{brut}[i,j] = 0 \vee M_{cf}[j] = 0 \\ 2 & \text{if } (M^x_{brut}[i,j] > 0 \wedge M_{cf}[j] > 0) \vee (M^x_{brut}[i,j] < 0 \wedge M_{cf}[j] < 0) \end{cases} \end{gathered} \tag{2.20}$$

$x := b - i$ represents the fact that the motion is obtained between the current note and the first note of the next measure. For example, with quarter notes, the gap between the third note and the first note of the next measure is defined as: $b = 4$, $i = 2$ and $x = 4 - 2 = 2$. The first note of the next measure is therefore 2 notes away.

The motions require relatively many constraints to be computed. Indeed, a boolean variable is needed for each type of direction of the counterpoint melody (3) as well as that of the *cantus firmus* (3). This gives 3*3 different possibilities to be divided into 3 categories of motions for each measure. This is not a problem in itself but with GiL, any boolean operation must be computed via a constrained boolean variable. Ideally one should use argument variables provided by Gecode that are intended to be temporary variables. Implementing this in GiL would probably improve performance.

IsCfB `*is-cf-bass-arr`

Boolean array of size s_m representing if the *cantus firmus* is below. Each list of this array represents a beat along the whole counterpoint and is calculated by comparing the pitch of the counterpoint with the pitch of the *cantus firmus* at the same time.

$$\begin{gathered} \forall i \in \mathcal{B}, \forall j \in [0, m) \\ IsCfB[i,j] = \begin{cases} \top & \text{if } Cp[i,j] \geq Cf[j] \\ \bot & \text{otherwise} \end{cases} \end{gathered} \tag{2.21}$$

By default, if both notes are the same then the *cantus firmus* is considered as the bass.

IsCons$_{(all,\ p,\ imp)}$ `*is-cons-arr`

Boolean array of size s_m representing if harmonic intervals are consonances, perfect consonantes or imperfect consonances. Each list of this array represents a beat along the whole counterpoint and is calculated by checking that harmonies belong to the corresponding set of consonances. By default, $IsCons \equiv IsCons_{all}$.

$$\forall i \in \mathcal{B}, \forall j \in [0, m)$$
$$IsCons[i,j]_{(all,\ p,\ imp)} = \begin{cases} \top & \text{if } H[i,j] \in Cons_{(all,\ p,\ imp)} \\ \bot & \text{otherwise} \end{cases} \tag{2.22}$$

2.2.4 Fonctions

Functions are a way to improve the readability of some more complex mathematical notations. The majority remain relatively simple.

nextm(x) Returns the number of measure(s) to add in 4/4 time signature depending on the number of beat x.

$$nextm(x) = \begin{cases} 1 + nextm(x-4) & \text{if } x \geq 4 \\ 0 & \text{otherwise} \end{cases} \tag{2.23}$$

buildScale(key, scale) Returns the set of notes in the key based on the $scale$ used. key is a value between 0 and 11 such that $0 \equiv C$ and $11 \equiv B$.

$$\forall x \in [-11, 127], \forall \delta := key + x \in [0, 127]$$
$$buildScale(key, scale) = \begin{cases} \bigcup_{\delta \ mod\ 12 \in key + \{0,2,4,5,7,9,11\}} \delta & \text{if } scale = \text{major} \\ \bigcup_{\delta \ mod\ 12 \in key + \{0,2,3,5,7,8,10\}} \delta & \text{if } scale = \text{minor} \\ \bigcup_{\delta \ mod\ 12 \in key + \{0,5,9,11\}} \delta & \text{if } scale = \text{borrowed} \end{cases}$$
$$\text{where } key \in [0, 11], scale \in \{"major", "minor", "borrowed"\} \tag{2.24}$$

N.B.: $buildScale(key, "minor") \equiv buildScale([key+3] \ mod\ 12, "major")$.

Membership function $e \in E$ State that e belongs to E. Technically, that's a fact but, *in the context of this paper*, this function can be used as a boolean function to evaluate an implication. It is then considered that this function returns a boolean value that is true if e is in the set E.

$$E := \{e_0, \ldots, e_n\}$$
$$e \in E = \begin{cases} \top & \text{if } (e = e_0) \lor \cdots \lor (e = e_n) \\ \bot & \text{otherwise} \end{cases} \tag{2.25}$$

As a result, when an expression uses only $\in$, it implies that this expression is true, i.e the element must belong to the set: $e \in E \equiv (e \in E \iff \top)$. This refers directly to the way Gecode allows this constraint. It may not follow convention, but it will be simpler and still used with common sense.

In the code, the constraints are often expressed separately for each element. For example, for a constraint cst which is applied if $e \in \{x, y, z\}$, it would state:

$$(e = x) \implies cst; \quad (e = y) \implies cst; \quad (e = z) \implies cst$$

2.3 Implicit General Rules of Counterpoint

In this section, all the following rules are implicit, sometimes taken from Fux's examples, and sometimes from music theory in general.

2.3.1 Formalization in English

G1 *Harmonic intervals are always calculated from the lower note.*

Indeed, any harmonic interval is a calculation of the absolute difference between two notes. This implies that they adapt to where the counterpoint is in relation to the *cantus firmus*. .

G2 *The number of measures of the counterpoint must be the same as the number of measures of the cantus firmus.*

The goal is to compose complete counterpoints which last the same time as the *cantus firmus*.

G3 *The counterpoint must have the same time signature and the same tempo as the cantus firmus.*

The notes must be played in sync.

G4 *The counterpoint must be in the same key as the cantus firmus.*

This is a fundamental rule of music in general. Since the music of the Baroque period does not follow the same standards as today's music, this rule is a bit more complicated than it seems. Indeed, it often happens that Fux gives examples with accidentals, i.e. notes that do not belong to the diatonic scale. There are therefore notes "borrowed" from other scales which do not appear as a basis for the key signature.

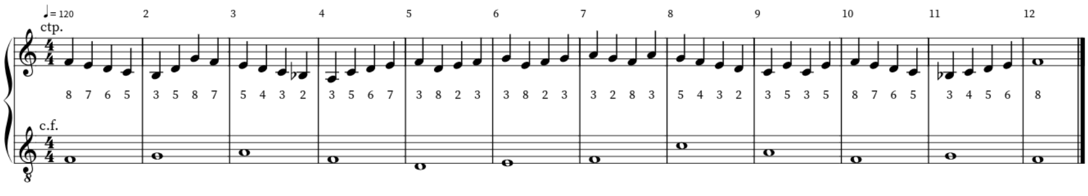


Figure 2.5: Example of a C major key signature starting on F with $B\flat$'s [23, p.54].

This makes it somewhat difficult to determine the precise domain of notes available for counterpoint. It is possible to determine the logic behind these borrowed notes. One way of looking at it is as follows: Fux composes with several different modes throughout his work: the F (Lydian) mode, the D (Dorian) mode, and others. In the rules of the first species (see section 3.1 at **1.H4**), it will be seen that Fux determines the use of a mode according to the first note of the *cantus firmus* in relation to the key of the musical work. Since the nature of a mode can be either major or minor, some notes can be borrowed from the major or minor diatonic scale of the first note of the *cantus firmus* respectively.

In figure 2.5, the key is C major, i.e. $[C, D, E, F, G, A, B]$. These notes can therefore be used in counterpoint, but that is not all. Since the first note is an F, this implies that the tonic of this work is F, although it uses the major scale of C, so it is an example of the use of the F mode, the Lydian mode. The Lydian mode being a major mode, some notes of the diatonic major scale of F can be used sparingly by counterpoint. Looking at several examples given by Fux, the notes borrowed are I ($[F]$[6] necessarily included since it determines the tonic of the work), IV ($[B\flat]$ the fourth), VI ($[D]$ note of the relative minor) and VII ($[E]$ the sensible which is most often used in the penultimate measure). These notes are probably not arbitrary, but for the purposes of this work, it is simply the examples provided by Fux that allow to say that these notes can be used sparingly if necessary.

If the key notes and the borrowed notes are merged, then the following set of notes is got: $[C, D, E, F, G, A, B\flat, B]$. Since the modes are variations of the diatonic scale, only a few notes are added in the end (one in this case). It is more complicated to understand when exactly these borrowed notes are used. Fux explains that these notes can be used to avoid certain intervals at certain times, which otherwise the melody would harshly imply the relationship of *mi* against *fa* [23, p.35]. Again, his approach to music is probably stricter than the current one, especially when his music was intended to be religious songs. That is why this setting is user-definable.

G5 *The range of the counterpoint must be consistent with the instrument used.*

This rule is relatively arbitrary and should be managed by the software user. Fux's treatise is mainly concerned with sung counterpoint, although it is applicable to any instrument. Most of the time, counterpoint is composed either in a higher register or in a lower register and more rarely both simultaneously. For performance reasons, the range in the software is limited to a size of one and a half octaves, i.e. 18 semitones. Which is in itself completely consistent with the style of counterpoint. The user still has the choice of the general pitch of the generated melody.

G6 *Chromatic melodies are forbidden.*

In this work, a melody is considered chromatic when three notes in a row are separated by semitones in the same direction. For example, $C \rightarrow C\sharp \rightarrow D$ or $C \rightarrow B \rightarrow B\flat$ are chromatic melodies. As a rule, this should never happen because the diatonic scale does not have those intervals. However, it might be possible to compose chromatic melodies by using borrowed notes in the use of certain modes.

G7 *Melodic intervals should be small.*

The purpose of a melody is to be melodious, but how to define that? This question is several centuries old and still does not have an answer that suits everyone. In his treatise, Fux argues that one should never neglect the beauty of singing. As a result according to his examples, most melodies consist of stepwise[7] motions with occasional leaps. One solution to represent this is to a give higher cost to larger melodic intervals. The appropriate cost function will be discussed in each chapter of species.

2.3.2 Formalization into Constraints

G1 *Harmonic intervals are always calculated from the lower note.*
Already handled by making the difference value absolute as seen in section 2.2.3 for the **H** variable.

G2, G3 *Same number of measures and same time signature.*
Only 4/4 time signatures are currently considered. The array Cp is therefore composed of four lists as explained in section 2.2.3 at **Cp**.

[6] Notes corresponding to the example are put in square brackets.
[7] Which moves by scale steps (i.e. one tone or one semitone)[30].

Listing 2.1: Definition of Cp in the first species.

```
(defvar *cp (list nil nil nil nil))
; ...
;; FIRST SPECIES ;;
; setting the first list of *cp with
;   integer *cf-len as size
;   set *extended-cp-domain as available notes
(setf (first *cp)
    (gil::add-int-var-array-dom *sp* *cf-len *extended-cp-domain))
```

G4 *The counterpoint must be in the same key as the cantus firmus.*

This rule is already handled by the creation of the set $\mathcal{N}$ as shown in section 2.2.3. The example of the actual rule given above will clarify the explanations. Let k be the value of the key determined by the key signature, i.e. 60 for C; and t the tonic of the piece, i.e. $Cf[0] = 65$. Then:

$$\mathcal{N}_{key} = buildScale(k \ mod \ 12, "major") = \{0, 2, 4, 5, 7, 9, 11, 12, \dots, 127\}$$
$$\mathcal{N}_{brw} = buildScale(t \ mod \ 12, "borrowed") = \{2, 4, 5, \mathbf{10}, 14, \dots, 125\}$$
$$\therefore \mathcal{N}_{all} = \{0, 2, 4, 5, 7, 9, \mathbf{10}, 11, 12, \dots, 127\}$$

To ensure that borrowed notes are used sparingly, they must be given a cost to use. Let $OffKey$ be the set of notes outside the key and $OffKey_{costs}$ the list of costs associated with each note. The cost for a note will be *<no cost>* or $cost_{OffKey}$ (*DFLT: <high cost>*).

$$\begin{gathered} OffKey = [0, 1, 2, \dots, 127] \setminus \mathcal{N}_{key} \\ \forall \rho \in positions(m) \\ OffKey_{costs}[\rho] = \begin{cases} cost_{OffKey} & \text{if } Cp[\rho] \in OffKey \\ 0 & \text{otherwise} \end{cases} \\ \text{moreover } \mathcal{C} = \mathcal{C} \cup \sum_{c \in OffKey_{costs}} c \end{gathered} \tag{2.26}$$

This equation is trivial but requires several adjustments in the program. Indeed, there is no boolean constraint in Gecode that assign the value *true* to a variable if an element belongs to a set[8]. This can be solved by creating the following constraints (see code sample 2.2). The idea is to add a 1 each time the candidate element $\equiv$ a member of the set. If the sum of this list ≥ 1 then the candidate appears at least once in the set.

Listing 2.2: Function that constrains b-member to be true if candidate is in member-list.

```
(defun add-is-member-cst (candidate member-list b-member)
    (let (
        (results (gil::add-int-var-array *sp* (length member-list) 0 1)) ; where candidate == m
        (sum (gil::add-int-var *sp* 0 (length member-list))) ; sum(results)
    )
        (loop for m in member-list for r in results do
            (let (
                (b1 (gil::add-bool-var *sp* 0 1)) ; b1 = (candidate == m)
            )
```

[8] To our knowledge, Gecode provides only a constraint such that an element must be a member of a certain set. Ideally, we would need a reified version of this constraint to allow a boolean associated with the result.

```
                (gil::g-rel-reify *sp* candidate gil::IRT_EQ m b1) ; b1 = (candidate == m)
                (gil::g-ite *sp* b1 ONE ZERO r) ; r = (b1 ? 1 : 0)
            )
        )
        (gil::g-sum *sp* sum results) ; sum = sum(results)
        (gil::g-rel-reify *sp* sum gil::IRT_GR 0 b-member) ; b-member = (sum >= 1)
)   )
```

G5 *The range of the counterpoint must be consistent with the instrument used.*

This rule is already handled by the creation of the set $\mathcal{N}^{\mathcal{R}} = \mathcal{N} \cap \mathcal{R}$ as shown in section 2.2.3. When Cp is created its domain is set to $\mathcal{N}^{\mathcal{R}}_{all}$ as seen in the code sample 2.1: `*extended-cp-domain` refers to the set $\mathcal{N}^{\mathcal{R}}_{all}$.

G6 *Chromatic melodies are forbidden.*

A three-note melody is chromatic if the interval between the first, second and third notes is one semitone in the same direction each time. This can be translated into the two following constraints.

$$\begin{gathered} \forall \rho \in positions(m-2) \\ (M_{brut}[\rho] = 1 \wedge M_{brut}[\rho+1] = 1) \iff \perp \\ (M_{brut}[\rho] = -1 \wedge M_{brut}[\rho+1] = -1) \iff \perp \end{gathered} \tag{2.27}$$

Listing 2.3: Function that prevents chromatic melodies.

```
; add melodic interval constraints such that there is no chromatic interval:
;   - no m1 == 1 and m2 == 1 OR
;   - no m1 == -1 and m2 == -1
; @m-intervals-brut: list of all the melodic intervals
(defun add-no-chromatic-m-cst (m-intervals-brut)
    (loop
        for m1 in m-intervals-brut
        for m2 in (rest m-intervals-brut) do
        (let (
            (b1 (gil::add-bool-var *sp* 0 1)) ; s.f. (m1 == 1)
            (b2 (gil::add-bool-var *sp* 0 1)) ; s.f. (m2 == 1)
            (b3 (gil::add-bool-var *sp* 0 1)) ; s.f. (m1 == -1)
            (b4 (gil::add-bool-var *sp* 0 1)) ; s.f. (m2 == -1)
        )
            (gil::g-rel-reify *sp* m1 gil::IRT_EQ 1 b1) ; b1 = (m1 == 1)
            (gil::g-rel-reify *sp* m2 gil::IRT_EQ 1 b2) ; b2 = (m2 == 1)
            (gil::g-op *sp* b1 gil::BOT_AND b2 0) ; not(b1 and b2)
            (gil::g-rel-reify *sp* m1 gil::IRT_EQ -1 b3) ; b3 = (m1 == -1)
            (gil::g-rel-reify *sp* m2 gil::IRT_EQ -1 b4) ; b4 = (m2 == -1)
            (gil::g-op *sp* b3 gil::BOT_AND b4 0) ; not(b3 and b4)
)   )   )
```

The previous function takes care of setting this constraint using GiL. This is a classical example that shows how constraints on all notes of the counterpoint are set when there is no distinction to be made between beats. In this case, `m-intervals-brut` always represent all the melodic intervals of the counterpoint and not the melodic intervals of a single beat as will often be the case later on. Indeed, one must always adapt to the rule to make it as simple as possible.

The functions often all look the same, a `let` block declaring the local variables, which are often all the booleans required to determine a situation. Then comes the execution block where the constraints determining the booleans (`g-rel-reify`) and the restrictive constraints (`g-op` states that `b1` and `b2` must not happen) are set. In the end, putting several constraints one after the other is the same thing as having these same constraints gathered in one separated by $\vee$.

G7 *Melodic intervals should be small.*

Just a global minimization of the melodic intervals could be asked to Gecode during the search for solutions but this would not be fully consistent with the stepwise principle. Having a stepwise melody considers that an interval of a semitone is worth the same as having one of a whole tone. It was decided to give the user full control over the costs of the melodic intervals. Indeed, the latter largely determine the melodies produced by the tool. From Fux's examples, the default costs for melodic intervals would be:

- the second intervals with no cost;
- the third, fourth and octave[9] intervals with *DFLT: <low cost>*;
- the other intervals with *DFLT: <medium cost>*.

$$\forall \rho \in positions(m-1)$$
$$Mdeg_{costs}[\rho] = \begin{cases} cost_{secondMdeg} & \text{if } M[\rho] \in \{0,1,2\} \\ cost_{thirdMdeg} & \text{if } M[\rho] \in \{3,4\} \\ cost_{fourthMdeg} & \text{if } M[\rho] = 5 \\ cost_{tritoneMdeg} & \text{if } M[\rho] = 6 \\ cost_{fifthMdeg} & \text{if } M[\rho] = 7 \\ cost_{sixthMdeg} & \text{if } M[\rho] \in \{8,9\} \\ cost_{seventhMdeg} & \text{if } M[\rho] \in \{10,11\} \\ cost_{octaveMdeg} & \text{if } M[\rho] = 12 \end{cases} \tag{2.28}$$
$$\text{moreover } \mathcal{C} = \mathcal{C} \cup \sum_{c \in Mdeg_{costs}} c$$

The case of the melodic tritone will be explained later in rule **1.M1**.

2.4 Types of rules

Three types of rules are distinguished in the next chapters:

- **Harmonic rules**: harmonic rules concern the harmonic intervals between the different voices, i.e. the harmony created by the *cantus firmus* and the counterpoint of the same measure. They are often the most important and the most numerous. These rules are noted by the letter **H**.
- **Melodic rules**: melodic rules refer to the melodic intervals of counterpoint or *cantus firmus*, which usually correspond to the gap between two consecutive notes of the same voice. These rules are noted by the letter **M**.
- **Motion or Harmonic and Melodic rules**: these rules use both of the above types of intervals. They are more complex and often relate to specific motions. These rules are noted by the letter **P** for *passage* since *M* is already taken[10].

The notation of the rules is: **S.TX** where **S** is the species, **T** is the type of rule (H, M or P), and **X** is the number of the rule. For example, the sixth harmonic rule of the first species is written **1.H6**.

[9] The melodic octave interval is important to be able to quickly return to a comfortable pitch.

[10] This way of classifying is only intended to clarify the idea behind a rule. It remains quite abstract and subjective because some rules are classified as melodic while they also use harmonic constraints. In no case does Fux make a delimitation between his explanations in this way.

Chapter 3

First Species of Counterpoint

> "With God's help, then let us begin composition for two voices. We take as a basis for this given melody or *cantus firmus*, which we invent ourselves or select from a book of chorales. To each of these notes, now, should be set a suitable consonance in a voice above [. . .]." Mann [23, p.27]

The first species of counterpoint consists of one note by measure, note against note. In other words, only whole notes.

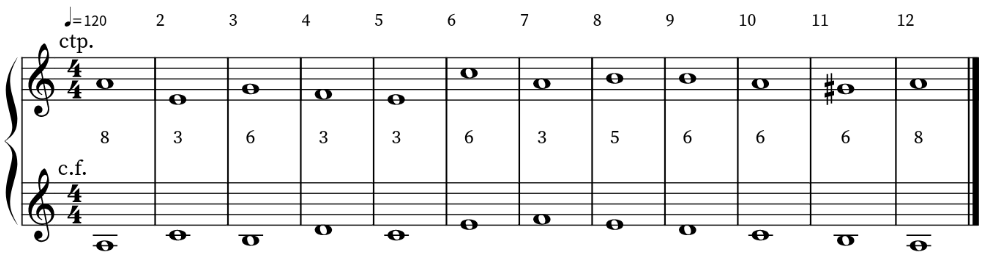


Figure 3.1: Example of a 1st species ctp. Score available here [26] and listen here [27].

As a reminder, *unless mentioned*, harmonic and melodic intervals are considered in absolute values. Moreover, harmonic intervals are modulo 12, so an octave interval is equivalent to a unison interval (see section 2.2.3).

3.1 Formalization in English

3.1.1 Harmonic Rules of the First Species

1.H1 *All harmonic intervals must be consonances*[1]. Chevalier [22, p.53]

> "[The master addressing his pupil] I shall explain to you. It is the simplest composition of two voices [. . .] which, having notes of equal length, consists only of consonances." Mann [23, p.27]

1.H2 *The first harmonic interval must be a perfect consonance*[2]. [22, p.54]

Perfect consonances are not those that bring the most harmony but those that give the most sense of stability and rest. They clarify the key and provide a strong foundation for the entire musical work. This rule applies to all species.

1.H3 *The last harmonic interval must be a perfect consonance.* [22, p.54]

Same logic as the previous rule. This one also applies to all species.

[1] This excludes dissonances which are seconds, fourths, and sevenths.

[2] Perfect consonances are fifths and octaves (or unisons).

1.H4 *The key tone is tuned according to the first note of the cantus firmus.* [22, p.56]

As seen in section 2.3.1, Fux sees the modes as variations of a single scale with different tonics. While the key signature gives the usable diatonic notes, the first note of the *cantus firmus* gives the tonic of the piece. This implies that some notes, the borrowed ones, will be available accidentally (e.g. $\sharp$ and $\flat$ in the key of C major) in relation to the tonic of the piece as explained in rule **G4**.

This rule also implies that the bass at the first and last note must be the tonic. To explain it another way, this means that if the counterpoint is in the lower part, only octave or unison harmonic intervals are available for the first and last note because of rules **1.H2** and **1.H3**. A wrong example would be figure 3.2.

Figure 3.2: Ctp. not keeping the key tone set by the *cantus firmus.*

G is used as a bass note to make a fifth instead of the D note required to keep the key of the *cantus firmus* [3]. This rule applies to all species.

1.H5 *The counterpoint and the cantus firmus cannot play the same note at the same time except in the first and last measure.* [22, p.62]

It does not mean that the harmonic interval cannot be equal to zero because an octave can occur. But unison in the strict sense of the term cannot be used in this case. This rule applies to all species for all thesis[4] notes.

1.H6 *Imperfect consonances*[5] *are preferred to perfect consonances.* [22, p.54]

Preferred means that all consonances are allowed but some cost, or "punishment", will be associated with the use of perfect consonances. This rule applies to all species for all thesis notes.

1.H7 *If the cantus firmus is in the lower part, then the harmonic interval of the penultimate note must be a major sixth.* [22, p.54]

This rule seems a bit strange at first, but there is a rational explanation for this. Indeed, traditional *cantus firmus* almost always end with a descending melody of one degree, for example, $E \rightarrow D$ or $F \rightarrow E$ (figure 3.3).

From this example, the rule makes sense because the major sixths of E and F are $C\sharp$[6] and D respectively. These notes are only one degree away from the tonic and lead perfectly by contrary motion to the tonics E and F. However, this implies several things. First, if big leaps are to be avoided in general, the last consonance will necessarily be an octave or unison because, as explained above, the closest note is necessarily the tonic.

[3] As it is, the work would be in G Mixolydian instead of D Dorian.

[4] Thesis means the note on the downbeat.

[5] Imperfect consonances are thirds and sixths.

[6] $C\sharp$ is a leading-tone to D. Leading-tone is a note that resolves to the next note, one semitone higher (or lower). It begins to be used in the late Middle Ages [36].

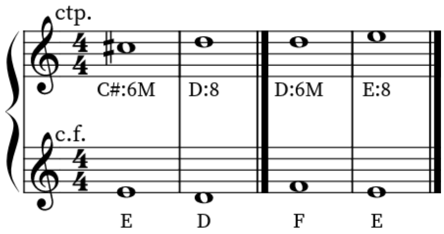


Figure 3.3: *Cantus firmus* ending with descending melodic intervals.

Secondly, if a composer wants to use the tool to compose from a *cantus firmus* that does not have the particularity of ending on a melody descending by one degree, then the solutions will not be very coherent on the penultimate measure. This point will be explained in more detail later. This rule applies to all species.

1.H8 *If the cantus firmus is in the upper part, then the harmonic interval of the penultimate note must be a minor third.* [22, p.54]

This rule goes hand in hand with the previous one. Indeed, a minor third is an inverted major sixth[7]. With the previous example, the notes of the counterpoint used would be exactly the same but this time would be below the *cantus firmus* (see figure 3.4). This rule applies to all species.

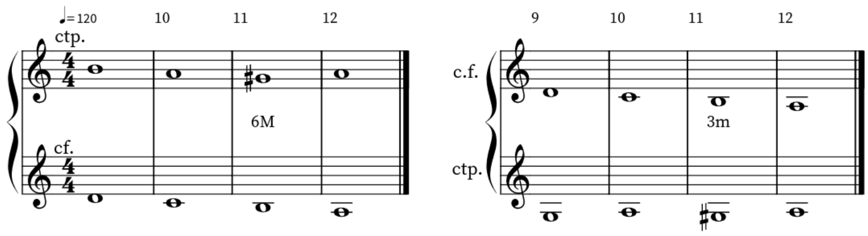


Figure 3.4: Equivalence between 6thM and 3rdm in penultimate measures, 1st species.

3.1.2 Melodic Rules of the First Species

1.M1 *Tritone[8] melodic intervals are forbidden.* [22, p.59]

The tritone, sometimes called the devil's interval by some [23, p.35], is a three-tone interval just below the perfect fifth [38]. It brings a lot of dissonance that was often avoided in the melody. This is a common rule of classical music in the broad sense but it is more used in today's music, so it can be deactivated. This rule applies to all species.

1.M2 *Melodic intervals cannot exceed a minor sixth interval.* [22, p.61]

> "[The master addressing his pupil] You shouldn't be so impatient, though I am most glad about your care not to depart from the rules. But how should you avoid those small errors for which you have yet had no rules? [...] you used a skip of a major sixth, which is prohibited in strict counterpoint where everything should be as singable as possible." Mann [23, p.37]

[7]If the octave interval is defined by 12 semitones, then the minor third is 3 and the major sixth is 9. The same note is found because $(Cf - 3)\ mod\ 12 = (Cf + 9)\ mod\ 12$. In other words, any note is the minor third of its major sixth.

[8]If you want to hear what is a tritone, you can check the video *What is a Tritone? Tritone Explained in 2 Minutes* (*Music Theory*) at `https://youtu.be/JJIO-Jr0E8o` [37].

As Fux explains, this rule applies especially to singers. As explained in rule **G7**, it is not very melodious to make big leaps in the melody anyway. This rule applies to all species with some exceptions.

3.1.3 Motion Rules of the First Species

1.P1 *Perfect consonances cannot be reached by direct motion.* [22, p.51, 57]

This rule is a good example of Fux overloading the explanations for perhaps a better understanding of the yesteryear audience.

> "First rule: From one perfect consonance to another perfect consonance one must proceed in contrary or oblique motion.
> Second rule: From a perfect consonance to an imperfect consonance one may proceed in any of the three motions.
> Third rule: From an imperfect consonance to a perfect consonance one must proceed in contrary or oblique motion.
> Fourth rule: From one imperfect consonance to another imperfect consonance one may proceed in any of the three motions." Mann [23, p.22]

As Martini [39, p.23] explains, these rules can be reduced to one such that the direct motion into perfect consonances is the only forbidden progression. Figure 3.5 violates the rule. This rule applies to all species.

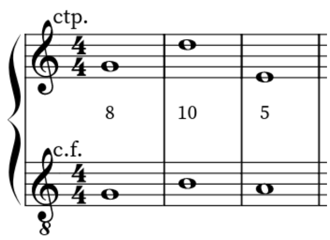


Figure 3.5: Perfect consonance reached by direct motion.

1.P2 *Contrary motions are preferred to oblique motions which are preferred to direct motions.* [22, p.53]

According to Fux, this would avoid making mistakes. Since the purpose of counterpoint is to have different melodies, it is understandable that contrary motion is preferable as the melodies will naturally differ. He is nevertheless criticized for the use of oblique motions which are, by some authors, forbidden.

Sachs and Dahlhaus [9] say that "The repetition of a note, causing oblique motion, is sometimes permitted only in the cantus, but may be used in either part (or even in both simultaneously, as a repeated note); it is not however the recommended 'next step'." Fabre [40][9] explains that the treatises of Marcel Bitsch[42], Marcel Dupré[43], or those of the 19th century, proscribe the repetition of a note.

Since the preference of the motion is different according to the musical context, this parameter is manageable by the user. This rule applies to all species.

1.P3 *At the start of any measure, an octave cannot be reached by the lower voice going up and the upper voice going down more than a third skip.* [22, p.61-62]

[9]Jean-Louis Fabre has a long experience teaching and practicing music. He has taught piano, music writing, and analysis at the conservatory and more [41].

This rule may seem arbitrary because it is. The original rule forbids this *battuta* octave[10] no matter how far the upper voice travels. Fux explains that "it is of little importance"[23, p.39] because he has found no particular reason for this rule, which is respected by authoritative composers. However, he thinks that the octave reached by a leap in the same context should be avoided.

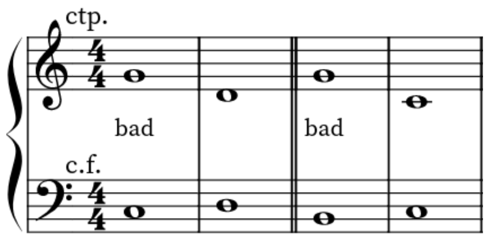


Figure 3.6: Example of battuta octaves.

On the right of figure 3.6, the octave is reached by a skip which is not good. While the example on the left is admitted by Fux. This rule applies to all species with some exceptions.

3.2 Formalization into Constraints

3.2.1 Harmonic Constraints of the First Species

1.H1 *All harmonic intervals must be consonances.*

$$\forall j \in [0, m) \quad H[0, j] \in Cons \tag{3.1}$$

This can be expressed with the constraint `(gil::g-member *sp* ALL_CONS_VAR h-intervals)` (see original code for more details).

1.H2, 1.H3 *The first and last harmonic intervals must be a perfect consonances.*

$$\begin{gathered} H[0,0] \in Cons_p \\ H[0, m-1] \in Cons_p \end{gathered} \tag{3.2}$$

1.H4 *The key tone is tuned according to the first note of the cantus firmus.*

Rule **G4** already handles the set of available additional notes. The only rule to add is that the first and last bass notes of the piece must have the same letter as the first note of the *cantus firmus* (i.e. unison or octaves).

$$\begin{gathered} \neg IsCfB[0,0] \implies H[0,0] = 0 \\ \neg IsCfB[0, m-1] \implies H[0, m-1] = 0 \end{gathered} \tag{3.3}$$

This is a good example of how implication works. `RM_IMP` on code sample 3.1 means that the boolean to its left implies the relation again to its left.

Listing 3.1: Function that constrains the first and last harmonies to be unisons or octaves.

```
; @h-interval: the harmonic interval array
; @is-cf-bass-arr: boolean variables indicating if cf is at the bass
(defun add-tonic-tuned-cst (h-interval is-cf-bass-arr)
    (let (
```

[10] Literally translated from Italian to "beaten". It refers to the downbeat.

```
5          (bf-not (gil::add-bool-var *sp* 0 1)) ; s.f. !(first is-cf-bass-arr)
6          (bl-not (gil::add-bool-var *sp* 0 1)) ; s.f. !(lastone is-cf-bass-arr)
7      )
8          ; bf-not = !(first is-cf-bass-arr)
9          (gil::g-op *sp* (first is-cf-bass-arr) gil::BOT_EQV FALSE bf-not)
10         ; bl-not = !(lastone is-cf-bass-arr)
11         (gil::g-op *sp* (lastone is-cf-bass-arr) gil::BOT_EQV FALSE bl-not)
12         ; bf-not => h-interval[0, 0] = 0
13         (gil::g-rel-reify *sp* (first h-interval) gil::IRT_EQ 0 bf-not gil::RM_IMP)
14         ; bl-not => h-interval[-1, -1] = 0
15         (gil::g-rel-reify *sp* (lastone h-interval) gil::IRT_EQ 0 bl-not gil::RM_IMP)
16 )   )
```

Since the negation of IsCfBass is required and Gecode does not offer a $\neg$ operation, it must be written in the form: $!p \equiv (p \iff \perp)$ where p is any predicate (see lines 9 and 11).

1.H5 *The counterpoint and the cantus firmus cannot play the same note at the same time except in the first and last measure.*

$$\forall j \in [1, m-1) \quad Cp[0, j] \neq Cf[j] \tag{3.4}$$

1.H6 *Imperfect consonances are preferred to perfect consonances.*

Only the cost for perfect consonance is definable (*DFLT: <low cost>*) which leaves a null cost for the imperfect consonances.

$$\begin{gathered} \forall j \in [0, m) \\ Pcons_{costs}[j] = \begin{cases} cost_{Pcons} & \text{if } H[0, j] \in Cons_p \\ 0 & \text{otherwise} \end{cases} \\ \text{moreover } \mathcal{C} = \mathcal{C} \cup \sum_{c \in Pcons_{costs}} c \end{gathered} \tag{3.5}$$

1.H7, 1.H8 *The harmonic interval of the penultimate note must be a major sixth or a minor third depending on the cantus firmus pitch.*

These two rules can be expressed with a single *if-then-else* constraint like this: `(gil::g-ite *sp* (penult *is-cf-bass-arr) NINE THREE (penult *h-intervals))`.

$$\begin{gathered} \rho := \max(positions(m)) - 1 \\ H[\rho] = \begin{cases} 9 & \text{if } IsCfB[\rho] \\ 3 & \text{otherwise} \end{cases} \\ \text{where } \rho \text{ represents the penultimate index of any counterpoint.} \end{gathered} \tag{3.6}$$

3.2.2 Melodic Constraints of the First Species

1.M1 *Tritone melodic intervals are forbidden.*

Instead of prohibiting this type of melodic interval, a cost is assigned (*DFLT: <forbidden>*) because it is a popular dissonant interval in today's music[11]. In addition, some less conventional *cantus firmus* than those of Fux might require a tritone on the last motion because of the number of constraints on the penultimate measure. This cost is managed by the user in the same way as the other melodic interval costs as described in the general rule **G7** at equation 2.28.

[11] Any major chord with a minor seventh has a tritone and this chord is the very basis of the blues [44]. It would be likely that users would arpeggiate on that with some melodic tritones.

$$\begin{array}{c}\forall \rho \in positions(m-1) \\ M[\rho] = 6 \implies Mdeg_{costs}[\rho] = cost_{tritoneMdeg}\end{array} \tag{3.7}$$

1.M2 *Melodic intervals cannot exceed a minor sixth interval.*

$$\forall j \in [0, m-1) \quad M[0,j] \leq 8 \tag{3.8}$$

For simple rules that apply to the whole list, it is possible to add a single line constraint like this: `(gil::g-rel *sp* m-intervals gil::IRT_LQ 8)`.

3.2.3 Motion Constraints of the First Species

1.P1 *Perfect consonances cannot be reached by direct motion.*

$$\forall j \in [0, m-1) \quad H[0, j+1] \in Cons_p \implies P[0,j] \neq 2 \tag{3.9}$$

This can be read as *if a harmony belongs to the perfect consonances then the motion to reach it is not direct* ($2 \equiv direct$, see **P** in section 2.2.3).

1.P2 *Contrary motions are preferred to oblique motions which are preferred to direct motions.*

- $cost_{con}$
  DFLT: <no cost>
- $cost_{obl}$
  DFLT: <low cost>
- $cost_{dir}$
  DFLT: <medium cost>

$$\begin{array}{c}\forall j \in [0, m-1) \\ P_{costs}[j] = \begin{cases} cost_{con} & \text{if } P[0,j] = 0 \\ cost_{obl} & \text{if } P[0,j] = 1 \\ cost_{dir} & \text{if } P[0,j] = 2 \end{cases} \\ \text{moreover } \mathcal{C} = \mathcal{C} \cup \sum_{c \in P_{costs}} c\end{array} \tag{3.10}$$

1.P3 *At the start of any measure, an octave cannot be reached by the lower voice going up and the upper voice going down more than a third skip.*

This rule can be represented by two sets of constraints. The first line of equation 3.11 represents the case where the counterpoint is on top while the second represents the case where the *cantus firmus* is on top.

$$\begin{array}{c}i := \max(\mathcal{B}), \forall j \in [0, m-1) \\ H[0, j+1] = 0 \land P[i,j] = 0 \land \begin{cases} M_{brut}[i,j] < -4 \land IsCfB[i,j] \iff \bot \\ M_{cf}[i,j] < -4 \land \neg IsCfB[i,j] \iff \bot \end{cases} \\ \text{where } i \text{ stands for the last beat index in a measure.}\end{array} \tag{3.11}$$

Chapter 4

Second Species of Counterpoint

The second species of counterpoint consists of two notes by measure, two notes against one note. In other words, only half notes.

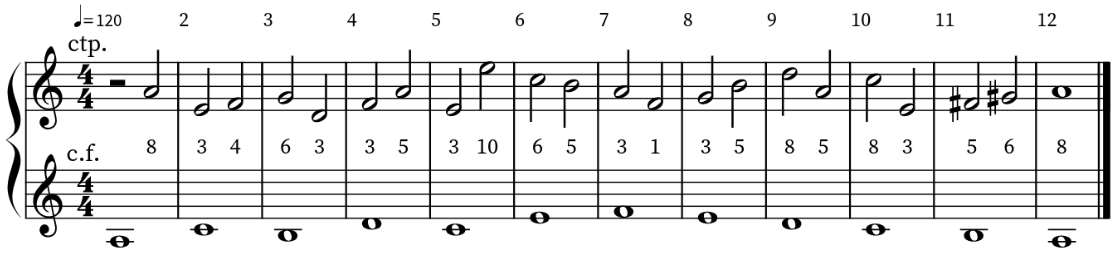


Figure 4.1: Example of a 2nd species ctp. Score available here [45] and listen here [27].

Since the second species is distinguished by a strong beat followed by a weak beat, the first species must be seen as a counterpoint composed of strong beats only. Therefore, all the rules of the first species that only apply per measure apply in thesis (e.g. rule **2.H1**). However, rule **1.M2** applies generally with the exception **2.M1**. Although the rules concerning the motions still hold, motions themselves are determined differently (see rule **2.P1**).

To sum up, first species harmonic rules are applied in thesis, while first species melodic rules are applied for all notes, and first species motions rules are adapted to the species.

4.1 Formalization in English

4.1.1 Harmonic Rules of the Second Species

2.H1 *Thesis*[1] *notes cannot be dissonant.* Chevalier [22, p.64]

As explained above, this rule is consistent with the **1.H1** one. Actually, it is written only to illustrate the associated logic because, in terms of constraints, the same are applied.

2.H2 *Arsis*[2] *harmonies cannot be dissonant except if there is a diminution*[3]*.* [22, p.64]

This might sound like a very restrictive rule but in reality, it is a common rule that applies itself in tonal music. In fact, any dissonance is surrounded by a consonance on each side.

Since rule **G7** insinuates that the melodic intervals are small, it makes perfect sense to go from one thesis consonance to the next thesis consonance through an arsis dissonance.

[1] Thesis means the note on the down beat.
[2] Arsis means the note on the upbeat.
[3] Diminution means an intermediate note that exists between two notes separated by a skip of a third.

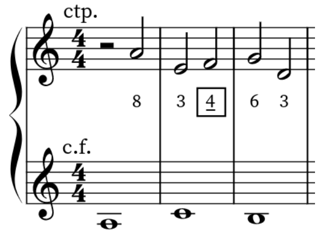


Figure 4.2: Diminution in arsis, 2nd species.

2.H3 *In addition to rules **1.H7** and **1.H8**, in the penultimate measure the harmonic interval of perfect fifth (unless exception **2.H4**) must be used for the thesis note.* [22, p.64-65]

The rules of the penultimate measure, although too strict for today's music (see rule **1.H7**), are still consistent with the other rules of the species. Since the penultimate note is a major sixth or a major third, the closest consonance in thesis is a fifth[4] (see figure 4.3).

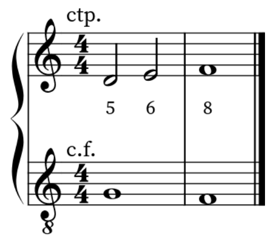


Figure 4.3: Basic penultimate measure, 2nd species.

2.H4 *In the penultimate measure, if the harmonic interval of fifth in thesis is not available, then a sixth interval must be used.* [22, p.69]

When Fux makes exceptions, it can get tricky so it is highly recommended to understand rules **G4** and **1.H4** and the notion of modes. It should also be noted that, at the end of Fux's examples, the *cantus firmus* tends to fall while the counterpoint tends to rise. It makes sense because the last motion must always be contrary[5].

Every musician knows that the seventh of the diatonic major scale does not have a perfect fifth in its key. That's why this rule exists. In figure 4.4a, the mode of E (i.e. the Phrygian mode) is used and the *cantus firmus* plays an F above. To have a perfect fifth, a $B\flat$ would have to be played, which is not available and is therefore replaced by an A to form a sixth.

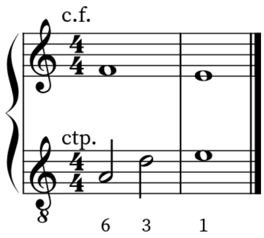
Where it gets tricky is when Fux shows this example (figure 4.4b) using the A mode (i.e. the aeolian mode, the relative minor). Why does Fux allow himself to use $F\sharp$ which gives a perfect fifth to B? As always the key used is C major (no $\sharp$ or $\flat$), but since the tonic is A the scale used will be extended to notes of the A major scale (i.e. $F\sharp$ and $G\sharp$)[6].

One might ask: why not a sixth as in the first example (figure 4.4a)? There are two reasons for this choice. First, the implicit rule **G6** that says chromaticism is forbidden

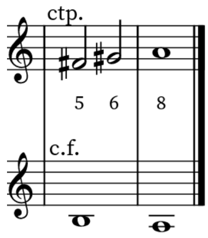
[4]With respect to the trend **G7** that says that the melody is stepwise.

[5]Or oblique in some cases. In Fux's examples, most of them tend to confirm this trend for the last two or even three notes of the counterpoint depending on the species. The examples given in this thesis are therefore strongly influenced by this idea which is omnipresent in the *Gradus ad Parnassum*.

[6]For more experienced musicians, this penultimate measure is immediately reminiscent of the melodic minor scale [46], which is common in classical music.



(a) 6th in thesis.



(b) ♯5th in thesis.

Figure 4.4: Different penultimate measures, 2nd species.

prevents a minor sixth because the melody would then be: $G \rightarrow G\sharp \rightarrow A$. Secondly, it could suggest that Fux prefers to go outside the diatonic scale to get a perfect fifth if the mode allows it rather than breaking the ground rule **2.H3**. More details regarding the costs will be given in the next mathematical section.

4.1.2 Melodic Rules of the Second Species

2.M1 *If the two voices are getting so close that there is no contrary motion possible without crossing each other, then the melodic interval of the counterpoint can be an octave leap*[7]. [22, p.67-68]

> "[...] if the parts have been led so close together that one does not know where to take them; and if there is no possibility of using contrary motion, this motion can be brought about by using the skip of [...] an octave [...]." Mann [23, p.45]

More explicitly, this case occurs when:

- the brut harmonic gap is a third or less;
- the *cantus firmus* is both below (/above) and rising (/falling).

Why a third? Because there is no more closed consonance than the latter[8].

According to Fux's examples, this rule applies only to $thesis \rightarrow arsis$ melodic intervals. Octave leaps seem to be unconditional in the case of $arsis \rightarrow thesis$ intervals. Moreover, it goes hand in hand with rule **G7** which says that melodic intervals should be small. Indeed, the octave skip allows to reset the pitch of the melody to go down (or up) again stepwisely (see figure 4.5).

Figure 4.5: Octave leap, 2nd species.

[7]The octave leap is quite natural and easy to sing because it is the first harmonic of the sound [47].

[8]Indeed, if the two voices are close, it is not possible to have a consonance other than unison (to be avoided) in this case.

2.M2 *Two consecutive notes cannot be the same.**[9]
In Fux's examples, none of them have oblique motions. This makes sense with the criticisms made for rule **1.P2**. This rule applies to the third species.

4.1.3 Motion Rules of the Second Species

2.P1 *If the melodic interval of the counterpoint between the thesis and the arsis is larger than a third, then the motion is perceived based on the arsis note.* [22, p.65-67]

Fux explains that the melodic interval between the note in thesis and the note in arsis determines which note will be kept in our mind. A third skip does not deviate enough from the thesis note to forget the latter. This implies that a perfect consonance to a perfect consonance cannot be saved by a third skip (see figure 4.6a) because the motion will be considered direct, which is not in accordance with rule **1.P1**. However, this rule allows the following situation in figure 4.6b.

(a) Bad direct motion with a 3rd skip. (b) Good contrary motion with a 4th leap.

Figure 4.6: Different motions based on different leaps, 2nd species.

2.P2 *Rule **1.P3** on the battuta octave is adapted such that it focuses on the motion from the note in arsis.**

Fux does not mention it in the second species. Instead of not applying the rule, it is adapted to prevent the same situation but considering only the note in arsis.

4.2 Formalization into Constraints

4.2.1 Harmonic Constraints of the Second Species

2.H1 *Thesis harmonies cannot be dissonant.*
As explained above, there is no constraint to add because it would be a duplicate of rule **1.H1**.

2.H2 *Arsis harmonies cannot be dissonant except if there is a diminution.*
Let $IsDim$ be a list of booleans of size $m-1$ representing if an arsis note is a diminution. A diminution can be described as follows: the interval between the notes in thesis is a third and the two intervals that compose it are seconds (one or two semitones).

$$\forall j \in [0, m-1)$$
$$IsDim[j] = \begin{cases} \top & \text{if } M^2[0,j] \in \{3,4\} \land M^1[0,j] \in \{1,2\} \land M^1[2,j] \in \{1,2\} \\ \bot & \text{otherwise} \end{cases} \tag{4.1}$$

[9]"*" means that this rule is implicit.

There is no need to use the brut melodic intervals to check if the melody always goes in the same direction[10]. This is because the constraint of third ensures the conditions to be met: $M^2[0,j] = \left|M^1_{brut}[0,j] + M^1_{brut}[2,j]\right|$. Besides, the constraint $<= 2$ can be used to represent $\in \{1, 2\}$ because the melodic intervals are never zero as will be seen later.

Listing 4.1: Function that constrains $IsDim$ to reprensent diminutions.

```
; @m-intervals-ta: the melodic interval between each thesis and its following arsis
; @m-intervals: the melodic interval between each thesis and its following thesis
; @m-intervals-arsis: the melodic interval between each arsis and its following thesis
; @is-dim-arr: the array of BoolVar to fill
(defun create-is-dim-arr (m-intervals-ta m-intervals m-intervals-arsis is-dim-arr)
    (loop
    for mta in m-intervals-ta ; inter(thesis, arsis)
    for mtt in m-intervals ; inter(thesis, thesis + 1)
    for mat in m-intervals-arsis ; inter(arsis, thesis + 1)
    for b in is-dim-arr ; the BoolVar to constrain
    do (let (
        (btt3 (gil::add-bool-var *sp* 0 1)) ; s.f. mtt == 3
        (btt4 (gil::add-bool-var *sp* 0 1)) ; s.f. mtt == 4
        (bta-2nd (gil::add-bool-var *sp* 0 1)) ; s.f. mat <= 2
        (btt-3rd (gil::add-bool-var *sp* 0 1)) ; s.f. mtt == 3 or 4
        (bat-2nd (gil::add-bool-var *sp* 0 1)) ; s.f. mta <= 2
        (b-and (gil::add-bool-var *sp* 0 1)) ; temporary BoolVar
    )
        (gil::g-rel-reify *sp* mtt gil::IRT_EQ 3 btt3) ; btt3 = (mtt == 3)
        (gil::g-rel-reify *sp* mtt gil::IRT_EQ 4 btt4) ; btt4 = (mtt == 4)
        (gil::g-rel-reify *sp* mta gil::IRT_LQ 2 bta-2nd) ; bta-2nd = (mta <= 2)
        (gil::g-rel-reify *sp* mat gil::IRT_LQ 2 bat-2nd) ; bat-2nd = (mat <= 2)
        (gil::g-op *sp* btt3 gil::BOT_OR btt4 btt-3rd) ; btt-3rd = btt3 || btt4
        (gil::g-op *sp* bta-2nd gil::BOT_AND btt-3rd b-and) ; temporay operation
        (gil::g-op *sp* b-and gil::BOT_AND bat-2nd b) ; b = bta-2nd && btt-3rd && bat-2nd
)   ))
```

To represent an action that produces only in one situation, this action must imply that situation. So it can be established that a dissonance in arsis implies a diminution like this:

$$\forall j \in [0, m-1) \quad \neg IsCons[2,j] \implies IsDim[j] \tag{4.2}$$

2.H3, 2.H4 *In the penultimate measure the harmonic interval of perfect fifth must be used for the thesis note if possible. Otherwise, a sixth interval should be used instead.*

If one wants to follow Fux's rules, it is important that the cost of leaving the diatonic scale is less than the cost of not having a fifth. For this, $cost_{penulthesis}$ is set to *<last resort>* which is greater than $cost_{OffKey}$ (*<high cost>*).

$$\begin{gathered} H[0, m-2] \in \{7, 8, 9\} \\ \therefore penulthesis_{cost} = \begin{cases} cost_{penulthesis} & \text{if } H[0, m-2] \neq 7 \\ 0 & \text{otherwise} \end{cases} \\ \text{moreover } \mathcal{C} = \mathcal{C} \cup penulthesis_{cost} \end{gathered} \tag{4.3}$$

4.2.2 Melodic Constraints of the Second Species

2.M1 *If the two voices are getting so close that there is no contrary motion possible without crossing each other, then the melodic interval of the counterpoint can be an octave leap.*

$$\begin{gathered} \forall j \in [0, m-1), \forall M_{cf}[j] \neq 0 \\ M[0,j] = 12 \implies (H_{abs}[0,j] \leq 4) \land (IsCfB[j] \iff M_{cf}[j] > 0) \end{gathered} \tag{4.4}$$

[10]The note would be a mere ornament like a suspended or added note instead of a diminution.

Where $H_{abs}[0, j] \leq 4$ states that there is no smaller consonance and $IsCfB[j] \equiv M_{cf}[j] > 0$ that the *cantus firmus* is getting closer to the counterpoint. As a reminder, M_{cf} is not absolute so $M_{cf} > 0$ states that the *cantus firmus* is necessarily rising.

2.M2 *Two consecutive notes cannot be the same.*

$$\forall \rho \in positions(m) \quad Cp[\rho] \neq Cp[\rho + 1] \tag{4.5}$$

4.2.3 Motion Constraints of the Second Species

2.P1 *If the melodic interval of the counterpoint between the thesis and the arsis is larger than a third, then the motion is perceived based on the arsis note.*

Let P_{real} be a list of size $m - 1$, with the same domain as a list of P, representing which motion is perceived between that coming from the thesis note and that coming from the arsis note. This implies that the costs of the motions and the first species constraints on the motions are deducted from P_{real}.

$$\forall j \in [0, m-1) \quad P_{real}[j] = \begin{cases} P[2, j] & \text{if } M[0, j] > 4 \\ P[0, j] & \text{otherwise} \end{cases} \tag{4.6}$$

Listing 4.2: Function that constrains P_{real} to represent the real motions.

```
; @m-intervals-ta: melodic intervals between the thesis and the arsis note
; @motions: motions perceived from the thesis note
; @motions-arsis: motions perceived from the arsis note
; @real-motions: motions perceived by the human ear
(defun create-real-motions (m-intervals-ta motions motions-arsis real-motions)
    (loop
    for tai in m-intervals-ta
    for t-move in motions
    for a-move in motions-arsis
    for r-move in real-motions
    do (let (
        (b (gil::add-bool-var *sp* 0 1)) ; s.f. (tai > 4)
    )
        (gil::g-rel-reify *sp* tai gil::IRT_GR 4 b) ; b = (tai > 4)
        (gil::g-ite *sp* b a-move t-move r-move) ; r-move = (b ? a-move : t-move)
)   ))
```

2.P2 *Rule **1.P3** on the battuta octave is adapted such that it focuses on the motion from the note in arsis.*

This constraint already had an adapted mathematical notation in the chapter of the first species. Note that this constraint would indeed use $P[2]$ and not P_{real}.

Chapter 5

Third Species of Counterpoint

The third species of counterpoint consists of four notes by measure, four notes against one note. In other words, only quarter notes.

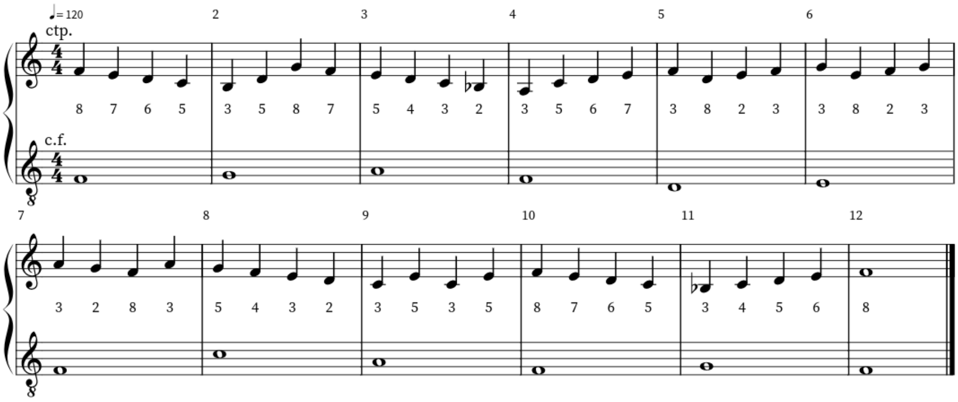


Figure 5.1: Example of a 3rd species ctp. Score available here [48] and listen here [27].

As in the previous chapter, the rules of the first species are applied to the thesis note, i.e. the first note of the group of four quarter notes. The first note of a measure is always the most important[1], it is the one that establishes the main harmony perceived by the human ear. To sum up, first species harmonic rules are applied in thesis, while first species melodic rules are applied for all notes, and first species motions rules are adapted to the species.

The third species is the one that starts to be vague in the explanations given by Fux. Admittedly, he probably didn't expect his work to be formalized through constraint programming. But even for musicians, there's no denying that some rules lack illustrative examples and are a bit skimmed over. In addition, the original treatise is in Latin and, despite access to several translations in French and English, the explanations do not always mean exactly the same thing, and everyone knows that the devil's in the details. This is reflected, for example, in the formalization of the first two harmonic rules, which are both created from fuzzy explanations and different translations.

5.1 Formalization in English

5.1.1 Harmonic rules of the third species

3.H1 *If five notes follow each other by joint degrees in the same direction, then the harmonic interval of the third note must be consonant.* Chevalier [22, p.73]

The following analysis is more the work of a historian than a computer scientist. The resulting formalization is therefore not the only way to go. As explained above, not all translations are equivalent. Chevalier's French translation, which is the most

[1] Unless there is syncopation as it will be explained in the next chapter.

recent and used as the main source in this thesis, says (see the original text in the appendix at A.3):

> If it happens that five quarter notes follow each other **by joint degrees**, either ascending or descending, the first one must be consonant, the second one may be dissonant, the third one again necessarily consonant, the fourth one may be dissonant **if** the fifth one is a consonance.

In contrast, Mann's English translation says:

> "[...] if fives quarters follow each other either ascending or descending, the first one [...]. The fourth one may be dissonant **if** the fifth is consonant [...]." Mann [23, p.50]

Alternatively, other older references as [49, p.51] and [50, p.4] from the XVIII century basically say:

> When five quarters follow one another **gradually** either rising or falling, the first, third **and** fifth note **must** be consonant. While the second and fourth may be dissonant.

Several issues arise from these previous sentences. First, Mann's English version does not say "gradually" or "by joint degree" which changes the rule itself. These terms make the constraint much more precise and therefore less restrictive. It can be said without too much hesitation that the rule must be applied only in the case of joint degrees because most translations propose a "gradually"[2]. Moreover, Fux's examples confirm this hypothesis.

Second problem: "**if** the fifth note is consonant". Why "if"? Actually, it's more complicated than that. For this rule, Fux does not explain if he is talking about:

(a) the four quarter notes of a measure plus the first one of the next measure;

(b) any five-note tuple;

(c) any independent five-note tuple that doesn't overlap with the previous one.

In Fux's examples, more than five notes follow each other several times, up to nine notes in a row in some. If the second assumption were true, then the following figure 5.2 from the book would not be correct.

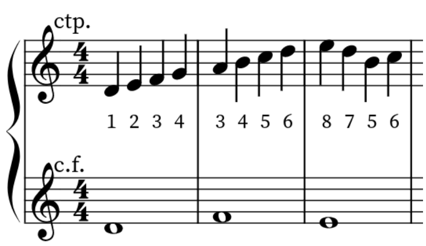


Figure 5.2: Nine quarters that follow each other gradually, 3rd species.

The third hypothesis (c) that states that Fux talks of *any five-note tuple as long as it is not itself in a previous five-note tuple* does not work either. Otherwise, figure 5.3 would not be right.

[2] In the original Latin text, Fux [8, p.63] states "continuò gradatim", which can be translated by "step by step".

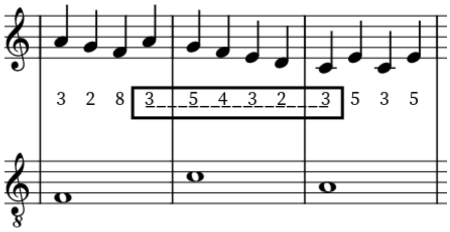


Figure 5.3: Six quarters that follow each other gradually where the 3^{rd} one is dissonant, 3^{rd} species.

It is clear that the third note is dissonant whereas with assumption (a), the rule would be maintained. As a result, it was decided that the first hypothesis was the right one. But it does not explain why it is said "if the fifth note is consonant". With this hypothesis, the fifth note is a thesis note and is therefore necessarily consonant thanks to rule **2.H1**. In the end, since saying that a note "may be dissonant" actually means that no constraint is added, the only additional constraint is the one on the third note.

3.H2 *If the third harmonic interval of a measure is dissonant then the second and the fourth interval must be consonant and the third note must be a diminution*[3]. [22, p.73-74]

Stepping back, this rule can be *partly* written in another more meaningful way: *any dissonance implies that it is surrounded by consonances*. Which makes sense in music because, in a melody, dissonances are often used to link the consonant notes of an explicit or implicit chord. The logical proof is given in the mathematical section 5.2 that follows.

3.H3 *It is best to avoid the second and third harmonies of a measure to be consonant with a one-degree melodic interval between them.* [22, p.74-75]

Fux calls this rule the *cambiata* note[4]. This rule is followed by composers of authority who stimulate the use of dissonances. As shown in figure 5.4, the seventh interval of the second note should be played rather than the sixth.

Figure 5.4: Use of the *cambiata* note in the second quarter.

3.H4 *In addition to rule **1.H8**, in the penultimate measure, if the cantus firmus is in the upper part, then the harmonic interval of the first note should be a minor third.* [22, p.75]

Fux, for some reason, does not always follow this rule, which he gives in a very crude way with a single example (figure 5.5a) to follow without further explanation. The only particularity of this measure is in the first and last note which are minor thirds, which is consistent.

However, Fux gives this example (figure 5.5b) which is not detailed. Luckily, Mann has footnoted that:

[3] An intermediate note that fills a skip of third.

[4] Literally translated from Italian to the "*exchanged* note". [23, p.51]

(a) Standard penultimate measure.

(b) Fux's deviation.

Figure 5.5: Different penultimate measures, 3rd species.

> "The forming of sequences (the so-called monotonia) ought to be avoided as far as possible. In the original [a] correction for the next to the last measure was added in manuscript". Mann [23, p.54]

This correction is yet another way of writing the penultimate measure. There is nothing wrong with Fux allowing deviations, that is what music is about in a way. But it makes systematic formalization more difficult. It was chosen to ignore this example and leave this rule optional because of its inconsistency with the rest.

5.1.2 Melodic rules of the third species

The melodic rule **2.M2** of the second species is applied to all notes.

3.M1 *Each note and its two beats further peer are preferred to be different.**[5]

This implicit rule is already generally present. It is kind of complementary to rule **2.M2** but in a softer way. It happens several times in Fux's work that the pupil prefers to put himself in difficulty to avoid monotony in the melody. An important aspect of this monotony can be found in the repetition of notes. In this species, it becomes important because not taking that into account could lead to having only two different notes per measure (see figure 5.6), which could be considered "boring". The cost of this parameter is still adjustable by the user.

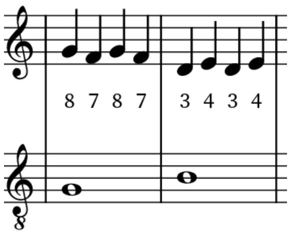


Figure 5.6: "Boring" example with only two different notes per measure, 3rd species.

5.1.3 Motion rules of the third species

3.P1 *The motion is perceived based on the fourth note.**

Fux stops talking about motions explicitly from the chapter on the third species. But the legacy of the first species, the idea of reaching perfect consonances by contrary motion, remains present in all his examples.

[5]"*" means that this rule is implicit.

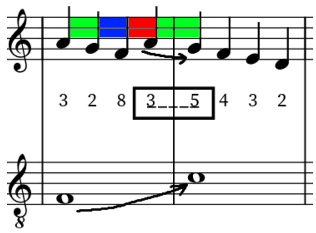


Figure 5.7: Contrary motion based on the fourth note. Colors represent that the motion is either contrary, oblique or direct.

The motion is here (figure 5.7) perceived from the note of the *cantus firmus* with the fourth note of the counterpoint of the corresponding measure[6]. In fact, the third species allows more flexibility in the motions because with more notes it is possible to go up during the first three notes to come down (or vice versa) just before the start of the next measure to obtain the desired motion as seen in figure 5.7.

5.2 Formalization into Constraints

5.2.1 Harmonic Constraints of the Third Species

3.H1 *If five notes follow each other by joint degrees in the same direction, then the harmonic interval of the third note must be consonant.*

$$\begin{gathered}\forall j \in [0, m-1) \\ \left(\bigwedge_{i=0}^{3} M[i,j] \leq 2\right) \wedge \left(\bigwedge_{i=0}^{3} M_{brut}[i,j] > 0 \vee \bigwedge_{i=0}^{3} M_{brut}[i,j] < 0\right) \\ \implies IsCons[2,j]\end{gathered} \tag{5.1}$$

On the one hand, the M is used for the "joint degrees" property while the M_{brut} for the "same direction" one.

3.H2 *If the third harmonic interval of a measure is dissonant then the second and the fourth interval must be consonant and the third note must be a diminution.*

To avoid negation in the code, which would require an additional step, the implication has been transformed into a logical or. The following constraints are set to be true.

$$\begin{gathered}\forall j \in [0, m-1) \\ IsCons[2,j] \vee (IsCons[1,j] \wedge IsCons[3,j] \wedge IsDim[j])\end{gathered} \tag{5.2}$$

where $IsDim[j] = \top$ when the 3$^{\text{rd}}$ note of the measure j is a diminution.

3.H3 *It is best to avoid the second and third harmonies of a measure to be consonant with a one-degree melodic interval between them.*

The default value of $cost_{Cambiata}$ is <*last resort*> because Fux almost seems to forbid it but without a real musical reason to justify this convention.

[6]Towards the next note of the *cantus firmus* with the first note of the counterpoint of the corresponding measure.

$$\forall j \in [0, m-1)$$
$$Cambiata_{costs}[j] = \begin{cases} cost_{Cambiata} & \text{if } IsCons[1,j] \wedge IsCons[2,j] \wedge M[1,j] \leq 2 \\ 0 & \text{otherwise} \end{cases} \quad (5.3)$$

3.H4 *In the penultimate measure, if the cantus firmus is in the upper part, then the harmonic interval of the first note should be a minor third.*

$$\neg IsCfB[m-2] \implies H[0, m-2] = 3 \quad (5.4)$$

5.2.2 Melodic Constraints of the Third Species

3.M1 *Each note and its two beats further peer are preferred to be different.*
This rule is implicit so the default value of $cost_{MtwobSame}$ is <*low cost*>.

$$\forall \rho \in positions(m-2)$$
$$MtwoSame_{costs}[i,j] = \begin{cases} cost_{MtwobSame} & \text{if } M^2[\rho] = 0 \\ 0 & \text{otherwise} \end{cases} \quad (5.5)$$

5.2.3 Motion Constraints of the Third Species

3.P1 *The motion is perceived based on the fourth note.*
This implies that the costs of the motions and the first species constraints on the motions are deducted from $P[3]$.

Chapter 6

Fourth Species of Counterpoint

The fourth species of counterpoint consists of syncopations[1], one note[2] shifted half a measure late against one note. In other words, only pairs of half notes[3].

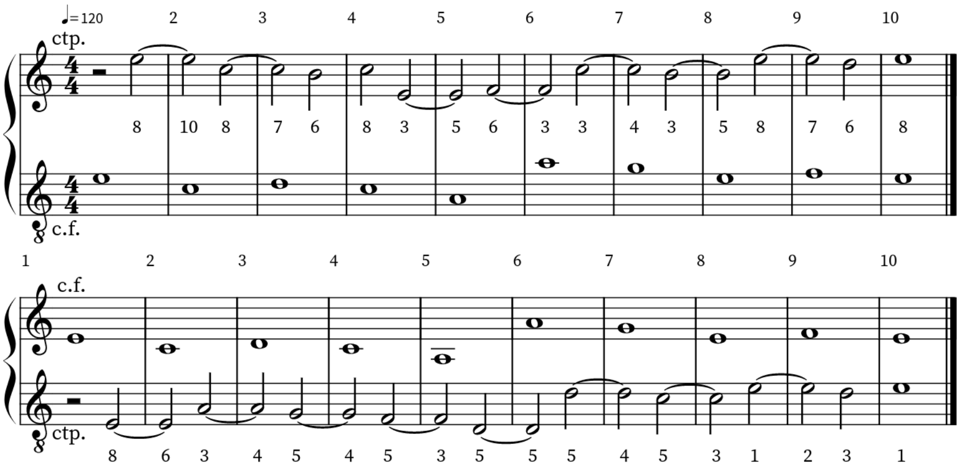


Figure 6.1: Two 4^{th} species ctp. Score available here [51] and listen here [27].

The fourth species is particular because it does not have more notes than the preceding species, it even has less. Indeed, this species is more like the first one. Here the syncopations are delays, which is roughly equivalent to using the first species with the whole note in thesis shifted in arsis which then lasts until the next arsis beat. While in the first species, all notes were consonant, here the syncopation requires more flexibility because the same whole note (here represented by a pair of half notes) is confronted with two different notes of the *cantus firmus.* First the second half of the first and then the first half of the second. If the syncopation is a delay of the note in thesis, then it is logical that the harmony it creates in arsis must be consonant (see rule **4.H1**). The specificity of the fourth species comes from the fact that dissonances can appear in thesis.

6.1 Formalization in English

For a better reading experience, the subsection on motion rules has been placed first as it is fundamental to understanding the other types of rules.

6.1.1 Motion Rules of the Fourth Species

For this species, no rule concerning the motions is given by Fux. Moreover, no invariant, which could have served as a basis for creating an implicit rule, has been found

[1]Syncopation creates an off-balance rhythm through the accenting of normally unaccented beats.
[2]Or rather two half notes with the same pitch.
[3]Except that the penultimate measure never has syncopation and it happens in certain measures that no syncopation is available.

in these examples. From another point of view, it could be seen that the motion created by a syncopation is nothing else than the oblique motion because one note stays in place while the other changes. This is of little importance because the rules concerning motions are somewhat adapted by rule **4.P2**.

4.P1 *Dissonant harmonies must be followed by the next lower consonant harmony.* Chevalier [22, p.78-81]

Any dissonant syncopation[4] should be resolved by moving downwards. This implies that if the *cantus firmus* is below, a second will resolve into a unison, narrowing the harmonic gap. Whereas if the *cantus firmus* is above, a second will resolve into a third, widening the harmonic gap. Figure 6.2 shows some examples of this rule.

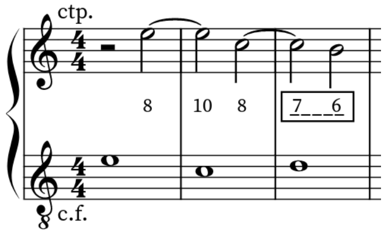


Figure 6.2: Dissonant syncopations resolved, 4th species.

4.P2 *If the cantus firmus is in the lower part then no second harmony can be preceded by a unison/octave harmony.* [22, p.79-80]

The idea behind this rule is that *no octave/unison harmony in arsis can be followed by an octave/unison harmony in the next arsis with a dissonant harmony in between*. It is a kind of adaptation of rule **1.P1** which says that perfect consonances cannot be reached by direct motion. Indeed, according to rule **4.P1**, a second that is dissonant must resolve into a unison. This would result in a unison sequence (see figure 6.3) if the retardation is removed, i.e. the second, which would violate rule **1.P1**.

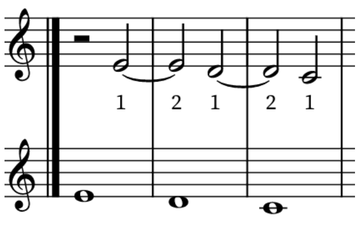


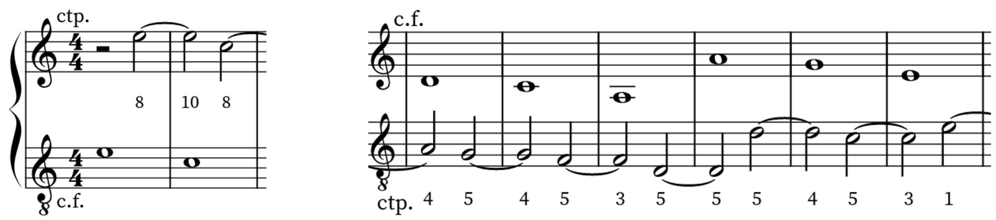
Figure 6.3: Seconds preceded by a unison, 4th species.

Now, let's dive into the in-depth logic of this rule. Although Fux's explanation is logical and the rule is applied in his examples, the logic itself is not applied to other similar problems later on. An example will speak for itself:

In figure 6.4a, a consonant syncopation consisting of an octave and a third[5] is then followed by an octave again. No problem, the rule is respected since no second has appeared, but why put an octave whereas if the delay is removed, one falls back into the same issue that originated this rule, (i.e. two consecutive arsis octaves)? Mann [23, p.95] suggests that "[…] in measures containing dissonant syncopations the essential part is the upbeat, the second, consonant, half." This can be paraphrased to say that the

[4] A dissonant syncopation is a syncopation that becomes dissonant at the changing note of the *cantus firmus*. It differs from a consonant syncopation which is strictly always consonant with the *cantus firmus*.

[5] Here the third is actually a tenth.



(a) Two consecutive arsis octaves. (b) Two consecutive arsis fifths.

Figure 6.4: Consecutive perfect consonances in arsis, 4th species.

human ear is only interested in the first consonance of a measure. This explains why the succession of octaves in the previous figure 6.4a is not one. Because the consonant third cuts off this impression.

What about fifths, which are also perfect consonances? In figure 6.4b, a consonant fifth ($G - D$) turns into a dissonant fourth ($G - C$) which is, as rule **4.P1** requires, resolved into a fifth again ($F - C$). There is clearly a succession of fifths. But for a reason that Fux does not detail but that Mann [23, p.57] points out: "In the case of fifths, however, the retardation can mitigate the effect of parallel motion. Successions of fifths may therefore be used with syncopations." Probably because the fifth brings a little harmony where the octave does not really[6]. It is therefore only the current rule **4.P2** specific to octaves that is admitted.

All this thinking is explained for a reason: the purpose of the final software is to assist a composer and that he can choose thanks to an obvious logic that some rules are obsolete in his own case. It is therefore preferable to have logical rules such as "no two perfect consonances in a row without another imperfect consonance in between". This rule would be more contextual, more global and would speak more to a composer. Here, the rule is adapted only for octaves so that it keeps the associated logic instead of explaining it in the form of forbidding a second after a unison.

6.1.2 Harmonic Rules of the Fourth Species

4.H1 *Arsis harmonies must be consonant.* [22, p.78]

Although explicitly described by Fux, this rule is only an adaptation of fundamental rule **1.H1** as explained above.

4.H2 *If the cantus firmus is in the upper part, then no harmonic seventh interval can occur.*

The origin of this rule is the same as rule **4.P2**. It is just less specific and therefore more restrictive because it does not depend on the previous or next harmony. Fux explains that this rule has no logical reason to exist. Nevertheless, the authoritative composers respected it, as did Fux as a result. It is optional for the previous reason.

4.H3 *For rule **1.H7** to be satisfied in the penultimate measure, if the cantus firmus is in the lower part, then the harmonic interval of the thesis note must be a seventh.*

The penultimate note cannot be a syncopation because the last note necessarily ends at the same time as the last note of the *cantus firmus* (see figure 6.5).

As usual in this case, the penultimate note is always a major sixth. The syncopation ending on the penultimate measure must be a dissonant seventh [7]. Following rule

[6]The octave is the simplest harmonic of its basic note with a frequency ratio of 2:1. Since it is the same note in a higher register it is not really about "harmony" as such. [52]

[7]Because of the structure of the *cantus firmus*, the seventh is often the tonic. This is a classic melodic progression at the end of a piece in tonal music that makes I - VII - I in degree (see *degree* in section 1.1.3).

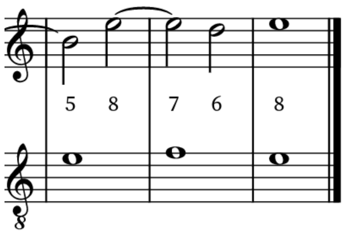


Figure 6.5: Penultimate measure, 4th species.

4.P1, the dissonance is resolved to the nearest consonance below.

4.H4 *For rule **1.H8** to be satisfied in the penultimate measure, if the cantus firmus is in the upper part, then the harmonic interval of the thesis note must be a second.*

The logic of the previous rule also applies to this one.

6.1.3 Melodic Rules of the Fourth Species

4.M1 *Arsis half notes should be the same as their next halves in thesis.*

In other words, *syncopations should occur if possible*. In theory, they are mandatory except in the penultimate measure. However, it happens that Fux breaks this rule to avoid monotony which is reflected by a repetition of a pattern in the musical work. This means that the cost of not putting a syncopation is lower than the cost of repeating the same syncopations. The difficulty is to know which cost best represents the monotony, which is quite subjective. Although all costs in the program have functional defaults, it's up to the composer to test various combinations to make the software shine. This will be shown in section 8.6.

4.M2 *Each arsis note and its two measures further peer are preferred to be different.*

This is a more or less implicit consequence of the previous rule and is also an adaptation of rule **3.M1**. For the same reason as the latter, it is better to avoid alternating only between two different syncopations. But this remains totally subjective because one could look for this very repetition in the syncopations. This is why the associated cost is customizable by the user.

6.2 Formalization into Constraints

Note that the arrays in index $[0, 0]$ are empty because the syncope arrives two beats late and leaves a silence in first thesis.

6.2.1 Motion Constraints of the Fourth Species

4.P1 *Dissonant harmonies must be followed by the next lower consonant harmony.*

There is no need to add the constraint $IsCons[2, j] = \top$ because it is already included by rule **4.H1** (see equation 6.3).

$$\forall j \in [1, m-1) \quad \neg IsCons[0, j] \implies M_{brut}[0, j] \in \{-1, -2\} \tag{6.1}$$

Listing 6.1: Function that constrains a dissonance to be followed by a consonance.

```
; @m-succ-intervals-brut: list of IntVar, s.f. brut melodic intervals
; @is-cons-arr: list of BoolVar, s.f. 1 -> the note is consonant
(defun add-h-dis-imp-cons-below-cst (m-succ-intervals-brut is-cons-arr)
```

```
    (loop for m in m-succ-intervals-brut for b in is-cons-arr do
    (let (
        (b-not (gil::add-bool-var *sp* 0 1)) ; s.f. !b (dissonance)
    )
        (gil::g-op *sp* b gil::BOT_EQV FALSE b-not) ; b-not = !b (dissonance)
        (gil::g-rel-reify *sp* m gil::IRT_LE 0 b-not gil::RM_IMP) ; b-not => m<0
        (gil::g-rel-reify *sp* m gil::IRT_GQ -2 b-not gil::RM_IMP) ; b-not => m>=-2
)   ))
```

4.P2 *If the cantus firmus is in the lower part then no second harmony can be preceded by a unison/octave harmony.*

$$\begin{gathered}\forall j \in [1, m-1) \\ IsCfB[j+1] \implies H[2,j] \neq 0 \wedge H[0,j+1] \notin \{1,2\}\end{gathered} \tag{6.2}$$

6.2.2 Harmonic Constraints of the Fourth Species

4.H1 *Arsis harmonies must be consonant.*

$$\forall j \in [0, m-1) \quad H[2,j] \in Cons \tag{6.3}$$

4.H2 *If the cantus firmus is in the upper part, then no harmonic seventh interval can occur.*

$$\forall j \in [1, m-1) \quad \neg IsCfB[j] \implies H[0,j] \notin \{10, 11\} \tag{6.4}$$

4.H3, 4.H4 *In the penultimate measure, the harmonic interval of the thesis note must be a major sixth or a minor third depending on the cantus firmus pitch.*

$$H[0, m-2] = \begin{cases} 9 & \text{if } IsCfB[m-2] \\ 3 & \text{otherwise} \end{cases} \tag{6.5}$$

6.2.3 Melodic Constraints of the Fourth Species

4.M1 *Arsis half notes should be the same as their next halves in thesis.*

The cost of not having syncope is by default <*last resort*>. It is because of costs like this that it is not really possible to compare the quality of two works of the same length just with the raw cost. Indeed, some *cantus firmus* may not have possibilities with syncopations only, which will artificially increase the total cost. It is therefore important to keep in mind that the costs are only relative to the *cantus firmus* used.

$$\forall j \in [0, m-1) \quad NoSync_{costs} = \begin{cases} cost_{NoSync} & \text{if } M[2,j] \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{6.6}$$

4.M2 *Each arsis note and its two measures further peer are preferred to be different.*

The default cost is <*high cost*> because monotony is very much avoided by Fux. It is unclear whether this cost should be higher than the cost of not having syncope.

$$\begin{gathered}\forall j \in [0, m-1) \\ MtwomSame_{costs} = \begin{cases} cost_{MtwomSame} & \text{if } Cp[2,j] = Cp[2,j+2] \\ 0 & \text{otherwise} \end{cases}\end{gathered} \tag{6.7}$$

Chapter 7

Fifth Species of Counterpoint

The fifth species of counterpoint, also called *florid counterpoint*, consists of a combination of the four preceding species but mainly of the third and fourth. Indeed, a florid counterpoint in Fux's work looks like an alternation between quarter notes and syncopations, with a few shorter syncopations and eighth notes (binding quarter notes). It is more uncommon to find half notes and it is difficult to determine whether they come from the second or the fourth species.

Figure 7.1: Two florid ctp. Score available here [28] and listen here [27].

The florid counterpoint is much free than its predecessors because the number of possibilities increases drastically with the possibility of adapting the species and thus the rhythm and the rules to obtain certain notes more easily than with the previous species. Here, more flexibility means easier to find a solution but more possibilities to explore for the solver. It is partly for these reasons that this chapter will be completely different from the previous ones. Where the others were formalizations of rules, this one is concerned with another problem: the relations between the different constraints of the previous species and the notion of rhythm that comes from them.

This chapter is mainly intended for computer scientists and mathematicians. This implies that the reader is aware of the different notions established in chapter 2 and that he understands the role of the variables previously introduced. Where the previous chapters were divided into two parts (natural language and constraints in the form of mathematics), the present chapter develops logic as a whole.

7.1 Problem Differences from Previous Species

Several points differentiate this species from others and influence the approach to be adopted:

- Fux does not describe new rules specific to this species, there are only new constraints linking the third and fourth species together. This is undoubtedly the lesson with the least information about its functioning.

- Fux shows variants at syncopations such that the second half note is replaced by a quarter note; and variations on quarter notes by replacing them with eighth notes to fill in third skips or add mordent[1].

- So far, the solver has no notion of rhythm, its only goal was to find a list of note pitches. Now it must also be able to calculate which species are used where so that a rhythm can be deduced.

- Since notes must be constrained differently depending on the species they are part of, all species constraints cannot simply be applied to all notes. Furthermore, it is impossible with Gecode to dynamically remove constraints after they have been applied[2]. Therefore, another way must be found to have the constraints applied fully dynamically.

7.2 Representation of Species as Constraints

As explained before, the only values that had to be calculated and explicitly provided by the solver were the list of MIDI notes that form the generated counterpoint. Since each species only has notes of equal duration (apart from the last note which is necessarily a whole note), there is no constraint determining whether a note must exist or not at a certain position. Moreover, in the lesson of the fifth species Fux gives only too little information on any rhythm to follow to extract hard constraints.

7.2.1 Naive Solution

A naive solution would be to individually generate solutions from the previous species and somehow merge them. The problem with this approach is that the flexibility offered by the fifth species would be lost. Indeed, certain notes of a certain species may be only accessible from the use of a note of another species. Therefore there would be no interaction between the species and the main asset of the solver would be lost, i.e. being able to find a better solution according to preferences and associated costs.

7.2.2 Species Array System

The only approach that seems correct is to create an array of integer variables the same size as the counterpoint array Cp. Each variable would then represent which species the note belongs to at the same location in Cp. In this case, all the variables will be used, i.e. as many as the number of notes in a counterpoint of the third species containing only quarter notes. If this array determines to which species the corresponding note belongs, it also determines if a note does *not* belong to any species. That is, whether a note at a certain beat of a certain measure exists or not. This is how the notion of rhythm appears. Caution, declaring that a note does not exist implies that it is not in the final result of the counterpoint that the user sees in the interface. But in reality, the note does indeed exist in the space of constraints for the solver. There is an important distinction between the notes displayed to the user and the notes calculated by the solver. All this will be explained in more detail later.

[1] A mordent is a type of ornament referring to a quick alternation between a note and its upper or lower neighbor.[32]

[2] It is possible to *add* constraints dynamically after the CSP has been created, but nothing has been found to perform the operation the other way around, which seems much more complex.

A mathematical formalization is necessary. Let S be an array of same size and structure as Cp[3] representing to which species belongs the note at the same index in the array Cp.

$$\forall \rho \in positions(m)$$
$$S[\rho] = \begin{cases} 0 & \text{if } Cp[\rho] \text{ is not constrained by any species} \\ 1 & \text{if } Cp[\rho] \text{ is constrained by the first species} \\ 2 & \text{if } Cp[\rho] \text{ is constrained by the second species} \\ 3 & \text{if } Cp[\rho] \text{ is constrained by the third species} \\ 4 & \text{if } Cp[\rho] \text{ is constrained by the fourth species} \end{cases} \tag{7.1}$$

Without going into details for the moment, the solver never generates solutions with $S[\rho] \in \{1, 2\}$ which gives in the current state a domain equal to $\{0, 3, 4\}$.

Figure 7.2: Representation of the species array S along a ctp., 5$^{\text{th}}$ species.

By analyzing figure 7.2, one may notice that some patterns are emerging: all syncopations are distinguished by $4 - 0 - 4$ while quarter notes are never followed by 0. These patterns are rhythm constraints imposed in the solver but for now let's leave that and assume that S has coherent values, i.e. syncopations and quarter notes where it is possible to have them.

Now let IsS_x be another array of same size and structure as Cp representing whether a note belongs to species x where x is the number assigned to the species in S just above.

$$\forall x \in \{0, 1, 2, 3, 4\}, \forall \rho \in positions(m)$$
$$IsS_x[\rho] = \begin{cases} \top & \text{if } S[\rho] = x \\ \bot & \text{otherwise} \end{cases} \tag{7.2}$$

For example, $IsS_0[i, j] = \top$ means that the note at the beat i from the measure j is not contrained by any species. This does not mean that no constraint is placed on this note, only that the constraints of the species placed on this note are in this case necessarily respected. When an $\vee\top$ is added to a constraint, it renders the original constraint useless because the whole thing then becomes a tautology which is equivalent to remove the original constraint.

7.3 Formalization of the Species Rhythm into Constraints

In order for the array S and IsS to have relevant values, i.e. values which respect a format making it possible to produce a coherent rhythm, there must be constraints imposing that certain species may or may not exist at certain positions. These constraints come from common sense and have been created from the examples of the *Gradus ad Parnassum*. The context is *no longer* Fux's music theory but computer logic. The first

[3] Size of s_m, composed of four lists each representing a beat over the entirety of the measures, as always.

four rules are mandatory for the proper functioning of the system while additional rules have been added to limit the possibilities of rhythm.

5.R1 *There must always be a note in thesis and in arsis, except the very first thesis and the very last arsis.*

No species would allow not to have a note in thesis and only the first species does not have a note in arsis, a species which is not used in florid counterpoint (the last whole note of the counterpoint is the same in all species and is therefore not considered a particularity of any species).

$$\begin{gathered} \forall j \in [0, m) \\ \neg IsS_0[0, j] \quad \text{where } j \neq 0 \\ \neg IsS_0[2, j] \quad \text{where } j \neq m - 1 \end{gathered} \tag{7.3}$$

5.R2 *The 4^th^ species can only exist in first and third beat.*

Indeed, the notes beginning or ending a syncopation in this species are always located in these beats.

$$\forall i \in \{1, 3\}, \forall j \in [0, m) \quad \neg IsS_4[i, j] \tag{7.4}$$

5.R3 *A 4th species in the third beat necessarily implies a 4th species in the first beat of the following measure and vice versa. The fourth beat should then have no note.*

This simply describes the usual syncopation which consists of the mandatory $4 - 0 - 4$ sequence (see figure 7.3).

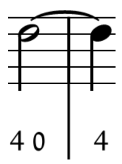


Figure 7.3: Syncopation implication in the S array, 5th species.

$$\begin{gathered} \forall j \in [0, m - 1) \\ IsS_4[2, j] \iff IsS_4[0, j + 1] \\ IsS_4[2, j] \implies IsS_0[3, j] \end{gathered} \tag{7.5}$$

5.R4 *A 3rd species cannot be followed by no note.*

If a quarter note is followed by no note then there would be at least one beat of silence, which is not intraseccally bad in music but is undesirable in counterpoint.

$$\forall \rho \in positions(m - 1) \quad IsS_3[\rho] \implies \neg IsS_0[\rho + 1] \tag{7.6}$$

5.R5 *Only 3rd species and 4th species are used.*

It has already been mentioned but as it stands, florid counterpoint is only composed of the third and fourth species in the solver. The formulation that Fux say that the fifth species is a mixture of the previous ones is confusing. Although species are based on common rules, Fux's examples clearly show a mixture of quarter notes and syncopations. Moreover, the half notes in a florid counterpoint can be generated by the second species as well as by the fourth (if the cost of not having syncopations is low).

In S the species are in the original domain in case future developments lead to adding the first and second species.

$$\forall \rho \in positions(m) \quad \neg IsS_1[\rho] \wedge \neg IsS_2[\rho] \tag{7.7}$$

5.R6 *The first and penultimate measures are linked to the* 4^{th} *species.*

Fux begins all of these counterpoints with an oblique motion created by syncopation and always ends them with a syncopation resolution before the last note. This can result in a first measure and a penultimate measure comprising the sequences $0-0-4-0$ and $4-0-4-0$ respectively (see figure 42). Rule **5.R3** placed above ensures that the syncopations are completed correctly.

$$\begin{gathered} IsS_0[0,0] \wedge IsS_0[1,0] \wedge IsS_4[2,0] \\ IsS_4[0,m-2] \wedge IsS_0[1,m-2] \wedge IsS_4[2,m-2] \end{gathered} \tag{7.8}$$

Figure 7.4: First and penultimate measures in the S array, 5th species.

It is worth noting that the only silence occurs at the beginning of the counterpoint and is defined by the sequence $0-0$. This is the only time this sequence occurs. Another point, with the addition of this constraint, the last note of the counterpoint is necessarily linked to the fourth species, which has no particular impact because this note has the same role in all species, i.e. to be in perfect consonance with the *cantus firmus.*

7.4 Logic Implication of the Species Constraints

Now that the solver knows when a note must be constrained by the rules of a species, it is necessary to represent this concept in the form of constraints.

7.4.1 Generalization of the Species Implications

For this, it is necessary that the previously established rules have the possibility of being activated only if the variables concerned by a rule are variables linked to the species to which the present rule belongs. In other words, a constraint of x^{th} species on a set of variables V must be true only and only if the variables V are bound to notes belonging to this x^{th} species. Unfortunately, this concept cannot be generalized to all the rules because some still apply when only part of the notes concerned is linked to the corresponding species. But an attempt at generalizing this idea can be written as such:

$$\begin{gathered} \forall x \in \{3,4\}, \forall cst_x \in Constraints(x), \forall V \in Variables(cst_x) \\ \left(\bigwedge_{\forall v \in V} IsS_x[v_{pos}] \right) \implies cst_x(V) \\ \text{where } Constraints(x) \text{ is the set of constraints of the species } x, \\ \text{and } Variables(cst_x) \text{ is the set of set of variables concerned by the constraint } cst_x, \\ \text{and } v_{pos} \text{ is the position of the } v \text{ related note in the array } Cp. \end{gathered} \tag{7.9}$$

It will be seen in equation 7.13 in the next section that all the variables concerned by a constraint do not necessarily have to belong to the species in question. From the point of view of programming, each rule had to be re-examined according to its basic operation. This part of the work revealed some architectural concerns that the software was not well enough adapted to handle this new logic, but this will be discussed in section 9.

Let's continue, in the current state of the program, florid counterpoint is considered to use either the third species or the fourth species. This means that a note has only three possible states: 0, 3 or 4. For example, rule **1.H1** states by extension that notes in *thesis* for the third species must be consonant but rule **4.H1** states that they are the notes in *arsis* for the fourth species which must be consonant. The two rules, hitherto distinct in two different species, result now in the fifth species in parallel.

Following the generalization:

$$\begin{aligned} \forall V \in Variables(1.H1_3) \quad & \left(\bigwedge_{\forall v \in V} IsS_3[v_{pos}] \right) \implies 1.H1_3(V) \\ \forall V \in Variables(4.H1_4) \quad & \left(\bigwedge_{\forall v \in V} IsS_4[v_{pos}] \right) \implies 4.H1_4(V) \end{aligned} \tag{7.10}$$

And concretely:

$$\begin{aligned} \forall j \in [0, m) \quad IsS_3[0, j] \implies (H[0, j] \in Cons) \\ \forall j \in [0, m-1) \quad IsS_4[2, j] \implies (H[2, j] \in Cons) \end{aligned} \tag{7.11}$$

It may seem simple but applying this logic to all the constraints of species 3 and 4 is not an easy task with the use of GiL which does not simply allow the addition of an implication on top of a constraint already written. The example above is one of the only cases where this is possible in this way but it must be understood that with GiL, which is only a precarious interface of Gecode, any intermediate step requires a new basic equation with only one operator. Mathematically, the equations would all follow the same notation which would basically just be a copy paste from the previous chapters. The rest of this chapter will therefore focus on the sometimes very specific relationships between species for certain rules that lead to slightly more complex constraints.

7.4.2 Avoiding Multiple Same Final Solutions

One might ask the question: what about notes where $S = 0$? These notes will not show up in the end user interface but the solver still calculates values for these notes. Does this mean that for a single solution on the user side there are a multitude of solutions on the solver side?

No, this is not the case because there is a constraint on the non-displayed notes, aka the non-constrained notes: they must be of the same value as the note of the next beat. In fact, it's the same as putting a fixed note on all the notes that don't appear, but for a branching issue, it's a little more efficient to work like that. The formulation is written as such:

$$\forall \rho \in positions(m-1) \quad IsS_0[\rho] \implies (Cp[\rho] = Cp[\rho + 1]) \tag{7.12}$$

7.5 Formalization of Inter-species Rules into Constraints

Fux, before beginning the lesson of the fifth species, describes variations in syncopations and the introduction of eighth notes, without going into too much detail. Chevalier [22, p.85]

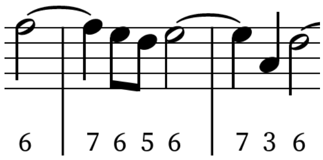


Figure 7.5: Variation of a syncopation with quarter and eighth notes, 5th species.

This kind of variation is used a lot to get more interesting rhythms and melodies. This can be considered as an inter-species rule and requires more attention than the simple example given above (equation 7.11). Figure 7.5 shows two things:

1. In relation to rule **4.P1**[4], the addition of quarter notes between the thesis and the arsis does not change the requirement to have an arsis consonance.

2. If the second eighth note is omitted, the melody does not move, which then implies that eighth notes can be used as mordents when the melodic interval between two beats is zero.

How to formalize these concepts with the new species array system? For the observation 1, it must be understood what is the role of the first quarter note in thesis. Since this is a quarter, shouldn't it be constrained by the third species? No, because this quarter note is part of the syncopation and is actually a 1/3 of the latter[5] played in arsis in the previous measure. This quarter note has no difference with the half note found in the original version of the syncope apart from its duration. So this note must be constrained by the fourth species. In fact, whether the duration of the note in thesis is one beat (quarter note) or two beats (half note) is only determined by whether or not a quarter note takes place in the second beat of the measure. To summarize the constraint that must be imposed: *an arsis note, regardless of its species, must be the consonance just below the thesis note if the latter belongs to the fourth species*. This can be mathematically described by:

$$\begin{gathered} \forall j \in [1, m-1) \\ \neg IsCons[0,j] \land IsS_4[0,j] \implies M^2_{brut}[0,j] \in \{-1,-2\} \land IsCons[2,j] \end{gathered} \tag{7.13}$$

There is indeed a constraint which is applied to the notes in $[2, j]$ whereas the latter do not necessarily belong to the fourth species according to the S array.

For the observation 2, Fux adds that:

> "Furthermore, two eighths may occasionally be used in the next species; that is, on the second and fourth beats of the measure but never on the first and third." Mann [23, p.63]

[4] A dissonant harmony in thesis must resolve in arsis with the next lower consonant harmony.

[5] If a whole note is 1 unit long, then a half note lasts 1/2 unit and a quarter note lasts 1/4 unit. That type of syncopation then lasts 3/4 unit which is equivalent to three quarter notes.

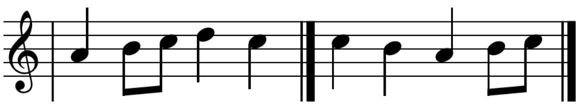


Figure 7.6: Addition of eighth notes in second and fourth beat, 5th species.

One might be disappointed to learn that these rules are not added as constraints in the CSP but several points led to this.

First, even though the solver does not know anything about the second eighth note (the first one being considered as the original quarter), the algorithm that generates the rhythm (see next section) after the solver has run still creates eighth notes. The end user therefore obtains counterpoints with eighth notes.

Second, the second eighth note of the eighths-pair is not bound by any rule. This means that no new solution with eighth notes can be found by the solver except the original solution with a quarter note instead. The eighth note only completes an already existing leap of third or adds a mordent.

Third, the architecture of the program was not designed to handle a whole new note subdivision, especially compared to the almost non-existent interest.

However, the only constraint which changes, or rather which withdraws with this system of eighths is that the melodic interval is not obliged to be zero between the second and third beat and between the fourth and first beat of the next measure. Therefore, rule **2.M2** which stated that two consecutive notes cannot be the same no longer applies at these positions.

7.6 Parsing of the Species Array in Rhythm

Rhythm species parsing occurs after the solver finds a solution. The parser therefore does not deal with Gecode variables but with values. Figure 7.8 is a simplified diagram of the parser. It represents a recursive function that takes as input the entire ordered lists Cp and S. This function outputs the final solution which will be shown to the user. The parser checks what is the next sequence of species and notes to find the corresponding note and associated duration. On the diagram, the notes are kept in the list `N` and the durations of the notes are kept in the list `R`. Once a sequence is found, it is removed from the lists Cp and S to be able to repeat the function again, this until the Cp and S lists are empty. This looks like a classic recursion pattern where the operation is performed on the head of the list and only the tail is kept for the next step. Here it is not necessarily a single element that is processed at a time but one to four elements.

The duration of notes in OpenMusic is represented as a fraction such that one unit represents an entire measure. Therefore, 1 represents the duration of one whole note; $1/2$ that of a half note; $1/4$ that of a quarter note; etc. If the value is negative, then a silence is played instead of a note. Also, in the diagram, the notation `L=[x:]` means that the list `L` is stripped of its first `x` elements. This means that the previously checked sequence occupied the space of `x` beats in total. Finally, for the parser to work correctly, the last value of S is replaced by 1 to signify that it is a whole note.

For example, if the values of the lists Cp and S are the following:

Cp	*72*	*72*	**72**	*72*	**72**	**71**	**71**	**69**	**67**	**69**	**69**	*69*	**69**	*68*	**68**	*69*	**69**
S	*0*	*0*	**4**	*0*	**4**	**3**	**3**	**3**	**3**	**3**	**4**	*0*	**4**	*0*	**4**	*0*	**1**

Table 7.1: Example of Cp and S, 5th species. Only the values in **bold** will be kept in the final solution.

Then the parsed output will be the following:

N		**72**	**71**	69	**71**	**69**	**67**	**69**	71	**69**	**68**	**69**
R	-1/2	**3/4**	**1/8**	1/8	**1/4**	**1/4**	**1/4**	**1/8**	1/8	**1**	**1/2**	**1**

Table 7.2: Parsed output of table 7.1, 5th species.

Note that the sum of the absolute values of `R` will always be equal to m, the number of measures. On the user side, this would appear:

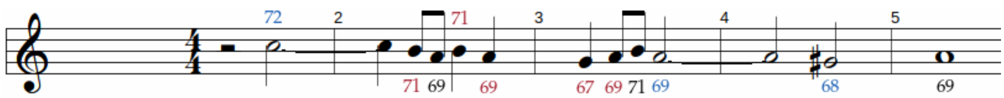


Figure 7.7: Final outcome from table 7.2, 5th species.

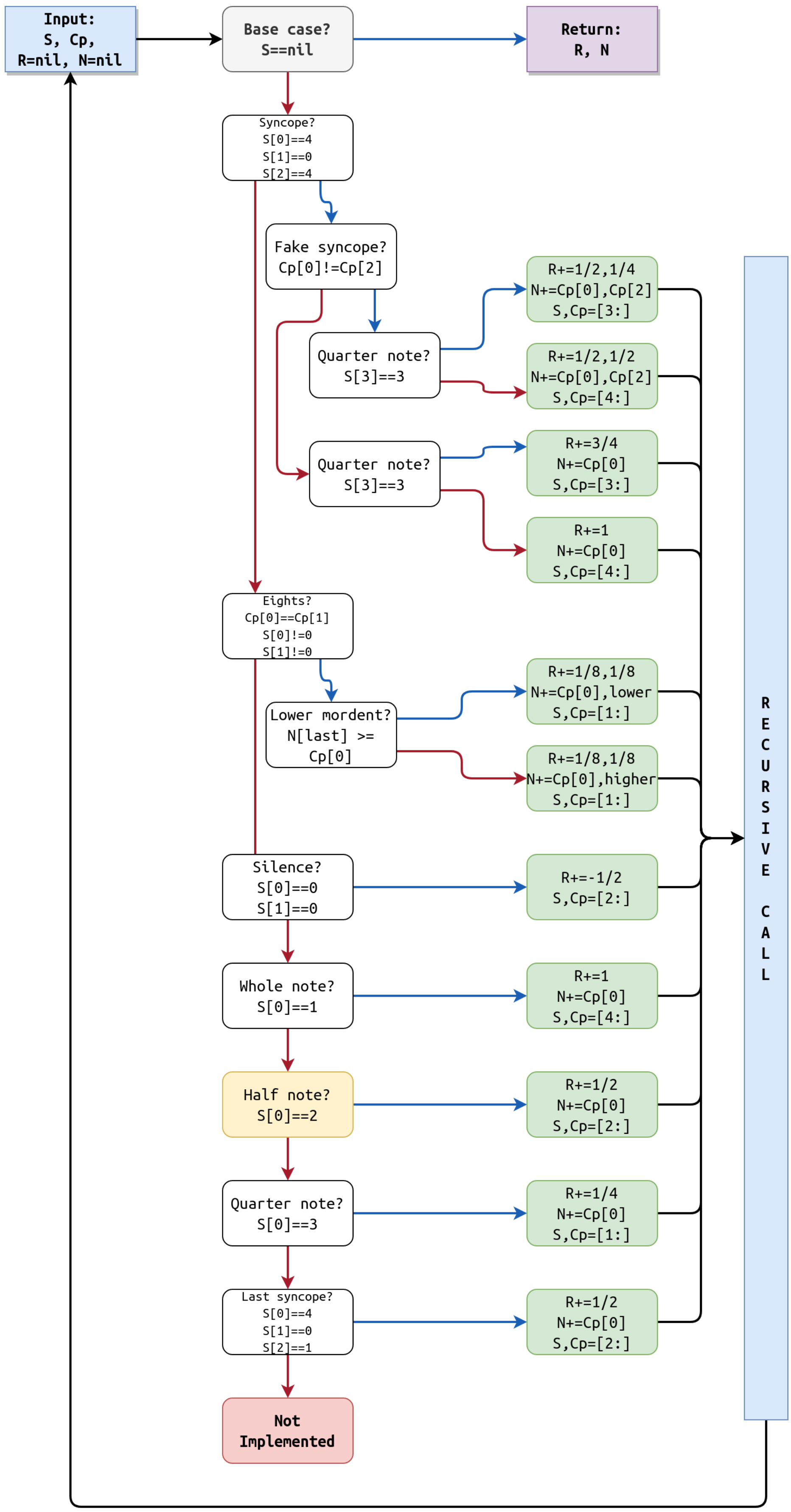


Figure 7.8: Rhythm species parser algorithm diagram, 5th species. A red arrow means the test failed while a blue one means it passed.

Chapter 8

Evaluation and Comparison

This chapter can be seen as the culmination of this dissertation. All the constraints have been described but what about the results? Do the counterpoints found by the solver equal those of Johann Joseph Fux? This is what this chapter will try to answer by comparing the species one by one.

The evaluations of the first four species will be simple analyzes of the differences and common points between the first counterpoint produced by the solver with the default values and the Fux counterpoint presented at the beginning of each species chapter. The analysis of the fifth species will be more advanced by tweaking the solver parameters to obtain more interesting counterpoints. A YouTube video is available **here**[27] to listen to the counterpoints presented in this chapter. The video follows the order of the following sections and includes a description with the time codes of each counterpoint.

Determining what a good counterpoint is is subjective and cultural. The following criticisms are therefore also subjective and cultural. It will be tried to make sociological objectivity and axiological neutrality[1]. It is thus good to note that these last are given by a man of Belgian culture appreciating Western music. Most people would say that Fux's counterpoints "look very baroque". It is therefore hoped that the counterpoints of the solver, presented below, will also be baroque. Moreover, the first four species are complicated to judge because with the absence of rhythm, an interesting melody will remain monotonous.

Let's not forget that the main goal is to observe if constraint programming can be useful in the field of music. Finally, these tests are performed with a version of the solver still under development (dated May 17, 2023). Some default values may have changed in the meantime during updates.

8.1 Evaluation of the First Species

The two counterpoints (see figure 8.1[2]) are globally very similar. A few differences are notable: the solver uses a fifth in 1st measure and does not use a sixth leap from the 5th to the 6th measure. This makes sense because the sixth leap has a cost of 2. Moreover this leap is surprising on the part of Fux because it is not melodically very interesting. Between the 9th and 10th measures, a fifth and an oblique motion are used. They both have a cost of 1. It would be the same to not have a fifth, but a sixth and therefore have a direct motion between the 9th and 10th measure. It would make the end of the song more moving and interesting. Another point is that Fux uses five direct motions (motions supposed to be avoided) while the solver only uses one. This first example shows what the solver is capable of. It respects the rules well and never surprises because it is not aware of it. This point will be discussed further later.

[1]Axiological neutrality is a methodological posture proposed by the sociologist Max Weber. This consists of the researcher becoming aware of his own values during his scientific work, in order to reduce as much as possible the biases that his own value judgments could cause. [53]

[2]The solver solutions come from OpenMusic and have been stretched to better see the score lines.

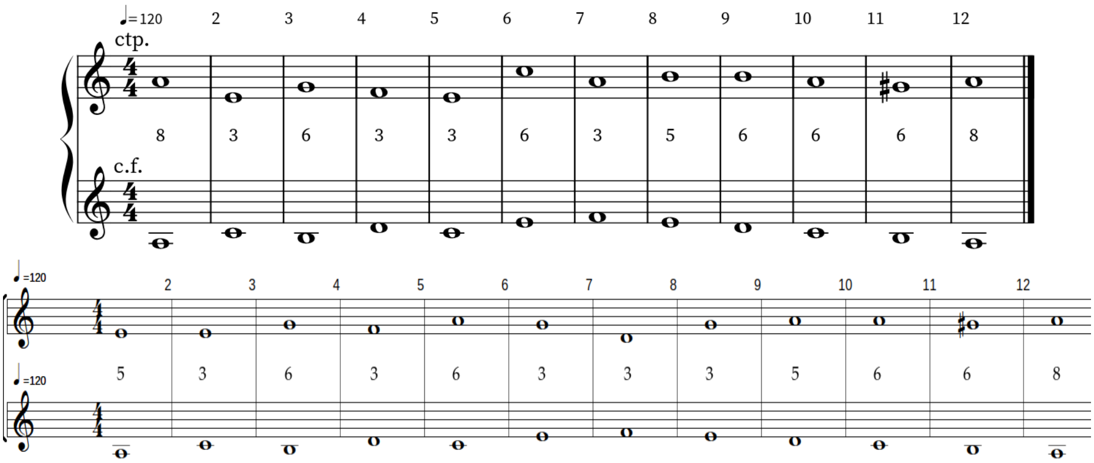


Figure 8.1: 1^{st} species ctp. of Fux (above) vs. ctp. of the solver [0.132 s] (below).

8.2 Evaluation of the Second Species

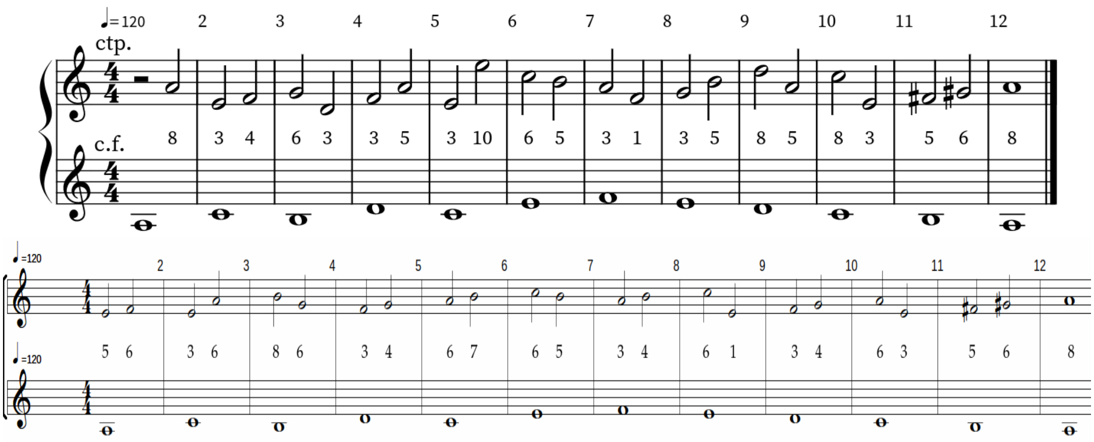


Figure 8.2: 2^{nd} species ctp. of Fux (above) vs. ctp. of the solver [26.849 s] (below).

For the general feeling, the counterpoint of the solver is relatively of the same quality as that of Fux. Besides that, the solver's solution has a four-note motif from the arsis in the 8^{th} measure to the thesis in the 10^{th} measure ($E \rightarrow F \rightarrow G \rightarrow A$). This motif is repeated immediately raising the F and the G by a semitone. It sounds both strange and interesting but one can doubt that Fux would appreciate this melody.

A surprising point is the use that Fux makes of the big leaps between the notes in thesis and those in arsis. For example, he makes a fourth leap in the 3^{rd} measure. According to rule **2.P1**[3], the resulting motion is perceived from the note in arsis, i.e. the motion from the 3^{rd} to the 4^{th} measure is considered direct (cost of 2) instead of contrary (no cost). This is typically the kind of behavior that does not occur with the solver.

Finally, one can notice that the search time for the answer is much higher than the previous one. This is a problem that particularly affects the second species and sometimes the fifth. This seems to come from rule **2.P1** discussed just before. Indeed, the best solutions of the solver (in terms of costs) often use large leaps to have more

[3] Reminder: If the melodic interval of the counterpoint between the thesis and the arsis is larger than a third, then the motion is perceived based on the arsis note.

contrary motions. It goes against stepwise melodies and therefore takes more time. As proof, if the cost of the motions is not taken into account, the solution is found in 0.2 seconds. Alternatively, a trade-off can be made by first adding branching from small values to the motions costs. Therefore, small costs for motions are calculated before other costs. The first solution found then deviates from the lowest possible cost but is found in 8 seconds. This is a fairly common optimization problem when the overall cost minimization is composed of inversely proportional costs.

8.3 Evaluation of the Third Species

Figure 8.3: 3rd species ctp. of Fux (above) vs. ctp. of the solver [1.789 s] (below).

The solver's counterpoint is musically quite poor. Generally speaking, it is monotonous and rambling. Compared to that of Fux, it does not sound really baroque. The big negative point that emerges is the permanent use of stepwise melodic intervals. It is true that Fux is quite mysterious about the rules that make up a good melody and that he uses a lot of one-step intervals. However, "a lot" does not mean "all". This is a very important notion that will be developed later: adding to the solver this notion of compromise, of surprise, of "a little bit of that, a little bit of this", etc. Typically, a way to force a minimum of melodic skips has been added to the solver to counter this problem. Another way to solve that is to put no cost to the melodic intervals of third for example.

Also, the solver's counterpoint contains a redundant melody ($A \rightarrow G \rightarrow A \rightarrow B$) which isn't bad in itself but seems to be randomly repeated and unsatisfactory. Obviously, the solver has no notion concerning the repetition of a pattern. This is also a major point to improve so that the solver can generate more human melodies. This solver really lacks an adjustable notion of monotony.

A more detailed evaluation of the third species can be read in the article of Sprockeels et al. [54]. It contributed to the improvement of this tool on the melodic level.

8.4 Evaluation of the Fourth Species

This example strongly highlights a defect of the solver: a poor melody. Indeed, the counterpoint is supposed to be composed of several melodies which, *independently*, sound melodious and which, *together*, sound harmonious. This horizontal vision of music is transcribed only through the fact that the counterpoint is generated from a

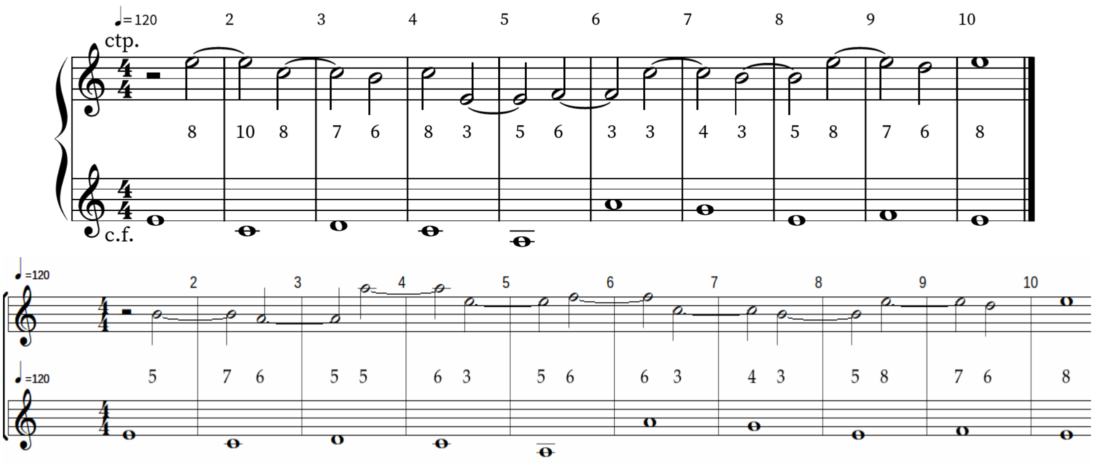


Figure 8.4: 4th species ctp. of Fux (above) vs. ctp. of the solver [0.012 s] (below).

counterpoint. But with Fux, no rule defines what the counterpoint should be as a consistent whole. In fact, his rules could be considered the "micro rules" of counterpoint. It would therefore be necessary for the solver to have "macro rules" defining the very structure of the counterpoint in its entirety.

In the 5th and 6th measure of Fux's counterpoint, the crossing between the two voices creates, for the time of three half notes ($F \rightarrow A \rightarrow C$), a rising melody by skips of third. This intertwining brings out an F major chord giving that nostalgic feeling to the song. On the side of the solver, this opportunity is missed. But actually, the notes are "identical"[4] from the second half of the 4th measure. The only two real differences between those counterpoints are that the solver starts on a fifth and that it prefers an octave leap to the interruption of syncopations. This last point also shows that Fux exaggerates when he explains that syncope should be used "wherever possible"[23, p.89].

Although the generated counterpoint is average, it can allow a more or less experienced composer to find a good counterpoint by shifting a few notes by one octave. It's not perfect, but for a musician who likes to experiment, the solver gives him a good basis instantly that he can then exploit.

8.5 Evaluation of the Fifth Species

For this species, the analysis will be more advanced. First, the counterpoints will only be compared and secondly, a more compositional approach will be put forward. In section 8.5.1, the solver counterpoints are the first results obtained with the default values. In section 8.5.2, the solver will be used more intelligently to obtain a more interesting solution.

8.5.1 Comparison

Whether it is the lower or upper counterpoint, those of Fux are clearly more baroque and are more melodious in general. Solvers' counterpoints aren't bad, but they're far from interesting. In figure 8.5, what Fux does in the 4th and 5th measure is the strong point of the work. The 4th measure has a D three times, which provides a pleasant rest before the repeat. He can afford this repetition because the D is the tonic of the

[4] In terms of the diatonic scale.

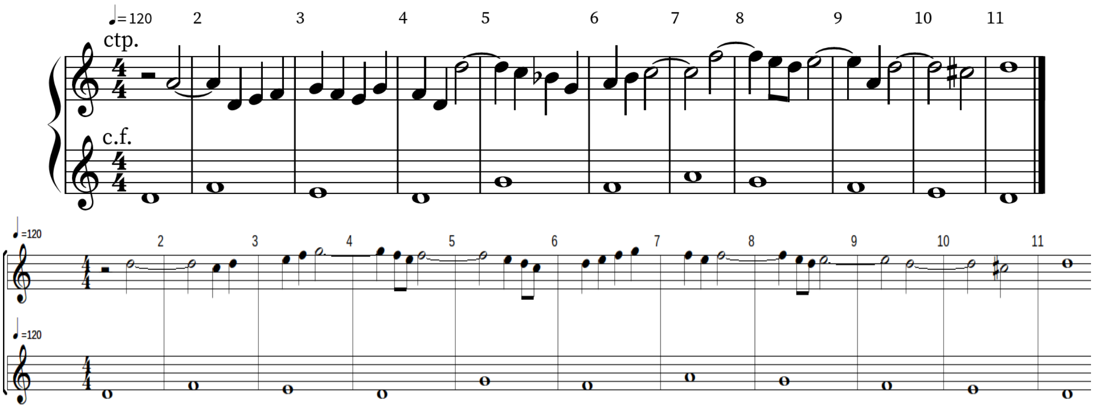


Figure 8.5: 5^{th} species ctp. of Fux (above) vs. ctp. of the solver [0.174 s] (below).

piece. The solver does not have this notion of rest and tension related to the underlying chord.

Also, the $B\flat$ in the 5^{th} measure adds a more nostalgic touch by suggesting a G minor chord. This $B\flat$ is not repeated in the next measure, which is rather original. Again, it's these kinds of little details that make Fux's counterpoints sound better than solver ones. The problem is the same as with the third species, i.e. the melodies are too "stepwise". For information, the counterpoints in the lower part have been added in the appendix in figure B.1 and the criticisms are generally the same.

One topic that hasn't been covered so far is cost comparisons. Indeed, if we force the solver with the same notes as the Fux counterpoint, it is possible to know what its total cost is. This can give a good idea of how well Fux applies its own rules and whether the costs assigned by the solver are consistent in determining what is or is not a good counterpoint.

In this case, the solver's solution costs 14 while that of Fux costs 29. It makes sense that the solver finds solutions with a lower cost since that is the goal of its heuristic, unlike Fux. The cost discrepancy comes mainly from the common use of skips and leaps by Fux. This already costs 9 where the solver has none. This represents almost a third of the total cost. In fact, this way of optimizing costs is not entirely consistent with Fux's music. On the other hand, the melody should still be mainly stepwise. Maybe there is an alternative?

8.5.2 Refinement

A point which was not specified in the previous section but which is important is the branching of the species array S. Indeed, the rhythm of the species is the same for figure 8.5(below) and figure B.1(below). It's not really a problem but the solver first randomly[5] determines which species are going to be used before it starts determining the associated notes. Indeed, it is much harder for the solver to first find an inexpensive solution and then determine if a rhythm can be associated with it. However, it is expensive but not impossible. This further minimizes the cost.

Another point discussed above was the possibility of having more diversified solutions at the level of melodic intervals. Three options have therefore been added to the user interface.

- *Irreverence* artificially increases the minimum cost of the solution. This has two

[5] The randomization is controlled and is done from a seed. Currently, the same seed is always used and there is no way to change it from the UI.

purposes: to prevent over-respecting solutions and/or to reduce the search time because the solver starts cost minimization with a higher lower bound.

- *Minimum percentage of skips* forces the solver to use larger melodic intervals.
- *Force joint contrary melody after skip* activates a rule[6] obliging a step melodic interval in the opposite direction after a skip.

By using these options (see figure 8.6) and lowering the costs associated with the melodic intervals of thirds, fourths, and fifths by one notch, a more interesting solution can be generated.

Figure 8.6: *Irreverence* and *Minimum percentage of skips* used for solution 8.7.

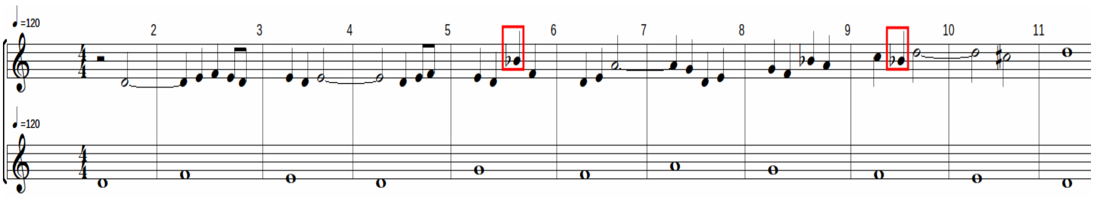


Figure 8.7: 5th species refined counterpoint of the solver [2 min 58 s].

This solution took nearly 3 minutes to be found, which is not huge but not negligible for a composer. Note that the notes boxed in red were changed to $B\flat$[7] to try the solver in a more realistic context[8]. With a few manipulations and tweaking, a good counterpoint is obtained in a few clicks and minutes. The solver shows that using it as a support tool can be very inspiring.

8.6 Experimentation with the Fifth Species

On our side, as a "*cantus firmus*", a more contemporary bass of 17 measures including chromatisms was tested. This example is presented at the end of the YouTube video[27] (at 6:02) and shows the ability of the solver to adapt to "*cantus firmus*" which are not at all classic. To be precise, the solver worked separately twice on this bassline with different preferences. In total, it generates a piece of 33 measures, in just over 3 minutes. The scores are available in the appendix in figure B.2.

We found that the result was stunning. This is clearly a sufficient starting point for at least one accompaniment in a piece of music. Indeed, it shows that even though the solver had been designed around music theory dating back to the Baroque era, it was still able to generate good melodies outside of its primary use.

Obviously, the music has several instruments giving more texture to the piece but all the instrumentation was chosen based on the melody generated by the solver. In the end, this is very good news because this solver is only a first step towards more complex and expert solvers. This demonstrates that it is quite possible in the future to use this kind of solver for more recent and freer music.

[6]Coming from a work of Gallon and Bitsch [42].
[7]These notes have only been transposed by one or two semitones to stay close to the original solution.
[8]Where a composer allows himself to change the few notes that bother him.

Chapter 9

Future Improvements

Given the vast field of computer-aided composition, several points could not be covered in this paper. This thesis is part of a large-scale project and several developers will continue this work which is gradually taking shape. This chapter will therefore cover the few points that need to be improved, as well as a few suggestions to ensure that the project progresses as smoothly as possible.

9.1 Software Architecture

A brief explanation of the project's architecture is available in appendix D to better understand how the software works overall. The Lisp code in this thesis does not have a good architecture for scalability. Indeed, the lack of Lisp skills and an iterative approach with short deadlines has led to an architecture containing "code smells"[1]. For example, object-oriented programming is a good paradigm for developing this project, but its use was not really emphasized.

Currently, constraints are added to a species via a long function that dispatches the constraints, rather than via class inheritance. Ideally, object-oriented inheritance should be used to represent the different variable arrays and species. All variable arrays (H, M, P, etc.) have something in common, whether in terms of their size relative to the *cantus firmus*, or in terms of the way certain rules are applied. A relatively abstract class should represent this type of array to enable these commonalities to be brought together.

The same applies to species that share common rules and should have been represented in a class system of their own. It would be logical for species to be children of the first species. Unfortunately, the scope of this work does not allow for a complete overhaul of the architecture. Moreover, in the near future, the entire code may have to be redone in C++ for reasons of performance, features, maintainability, and so on. Also, GiL has reached its limits, both in terms of ease of programming and in terms of possibilities. The Lisp language is not designed for writing mathematics, since each operation requires a different function call. Code readability can become complicated because these calls are all represented by parentheses. At the same time, it is not possible with GiL to combine basic mathematical operations to form a larger one. One has to break down each complex operation into simple intermediate basic operations a bit like writing assembly, which is undesirable for larger projects. Not to mention that branch-and-bound, heuristics, and multithreading seem complicated to implement in GiL.

Gecode is already a parser implemented in C++, so we strongly advise against using and maintaining GiL in future projects. Constraints should all be written in C++ using the features and facilities that have been implemented in Gecode.

[1] A code smell is a characteristic of a bad code that indicates a certain type of problem[55]. In this case, the code contains some bloaters and change preventers[56].

9.2 Solver Performances

So far, few optimizations have been implemented to reduce the search time of the solver. Performance was not a particularly important point until the fifth species which requires more resources. Most of the time, the solutions are found quickly but in some cases, the solutions can take several minutes, or even never be found in a reasonable time. Indeed, some extreme cases lead to inefficient branching which only finds solutions in infinite time. This is due to several points.

First, the branching is not very dynamic and therefore does not adapt much to the parameters chosen by the user. It is just different with respect to the species. Also, a minimal cost to the solution is provided to prevent the solver from looking for solutions with too low costs that cannot exist. But this remains rudimentary and is not sufficient to find solutions with certain parameters.

Second, for minimization problems, Gecode uses a specific class where the space cost is kept in a variable to be able to minimize it efficiently. Gecode has optimized its algorithm for this kind of problem and does not use simple naive branching. GiL, a priori, does not allow the use of this feature, which undoubtedly considerably slows down the search for a solution. Also, the solver's solutions don't have to be the best but just "good enough".

Third, the current branching is very naive. For example, for the third and fifth species, a branching is done on the cost of the melodic intervals to start finding solutions with no cost on this level. By default, this means that the solver searches for stepwise solutions which, indeed, makes finding solutions much easier. But in fact, it would be better to have a function for this part of the heuristic to first find solutions with mainly joint intervals but also some disjoint intervals. This third point is related to both the performance and the quality of the solutions.

9.3 Solution Quality

During the evaluation, it was shown that several notions on the global architecture of the melody were missing from the formalization. Whether through constraints, heuristic functions, or branch-and-bound, these notions must be represented to find more human solutions. The human ear likes to be able to predict notes but also needs to be surprised from time to time so as not to get bored. This is exactly the problem the solver is struggling to handle. Everything is a question of balance which should be represented by the direction the solver takes when looking for solutions.

As explained above, this can be introduced by a more complex heuristic, capable of looking for solutions including certain skips at places that seem coherent. It can also be introduced by a heuristic including certain patterns in the solution to try to either repeat them or avoid them. It is also possible to find other formalizations of counterpoint giving "macro rules" capable of governing the progression of counterpoint as a whole. Some of these rules can be detected via more general works or via counterpoint statistical analyzes using certain algorithms or certain machine learning models. Several possibilities are offered to the next developers of this project.

Lately, an interesting feature would be that composers can impose a rhythm and certain notes so that the user experience is more complete. This would make it possible to create variations to the melodies and to use the tool as a real component in the creation of a complete work where counterpoint would only be part of it.

Conclusion

In conclusion, this thesis has made significant progress towards the development of a constraint programming based tool for creating music. However, it is important to note that the work presented here is still a work in progress, with several areas that require further exploration and refinement.

One of the key findings of this research is the recognition that a comprehensive formalization of musical rules is crucial for CP to be a relevant approach. The formalization of musical rules using discrete mathematics and constraints provides a solid foundation for generating musically correct solutions. However, it is essential to acknowledge that the process of formalizing all the intricate nuances of music is a challenging task. The use of more precise works could be useful to formalize the counterpoint even better.

The analysis of the generated counterpoint compositions based on Fux's rules has highlighted the need for additional constraints on melodic development, particularly in terms of long-range melodic relationships. While the tool successfully creates harmonically interdependent and melodically independent counterpoints, incorporating constraints that generate more interesting melodies would be a valuable direction. Also, from a technical point of view, the software architecture, performance, and quality of the solutions must be more taken into consideration in the future.

In addition, the successful experimentation outcome signifies that while the current solver represents an initial step, it holds great potential for more complex and advanced solvers in the future. These findings provide optimistic prospects for using similar solvers outside the domain of counterpoint. Indeed, the approach presented in this thesis can be extended to more complex musical styles. CP has the potential to be a powerful paradigm in computer-aided composition for a wide range of musical genres. The application of specific rules and constraints for different styles will open up new possibilities for composers and expand the creative potential of the tool.

In summary, while this thesis has laid a foundation for the development of a constraint programming based composition tool, there is still work to be done. Further research and development are needed to refine the formalization process, incorporate additional constraints on melodic development, and explore the application of CP in more complex musical styles. With continued efforts and advancements in these areas, we hope that constraint programming has the potential to revolutionize computer-aided composition and empower composers with new tools for musical creativity and expression.

Bibliography

[1] IRCAM website Authors. *L'Ircam*. IRCAM. May 30, 2023. URL: https://www.ircam.fr/lircam.

[2] Gecode website Authors. *Generic Constraint Development Environment*. Gecode. June 1, 2023. URL: https://www.gecode.org/index.html.

[3] OpenMusic website Authors. *OpenMusic*. GitHub. June 1, 2023. URL: https://openmusic-project.github.io.

[4] GiL Contributors. *GiL*. GitHub. May 3, 2023. URL: https://github.com/sprockeelsd/GiL.

[5] Christine Payne. *MuseNet*. OpenAI. Apr. 25, 2019. URL: https://openai.com/blog/musenet.

[6] Cheng-Zhi Anna Huang, Ian Simon, and Monica Dinculescu. *Music Transformer: Generating Music with Long-Term Structure*. Magenta. Sept. 16, 2019. URL: https://magenta.tensorflow.org/music-transformer.

[7] Lucas N. Ferreira and Jim Whitehead. *Learning to Generate Music With Sentiment*. 2021. arXiv: 2103.06125 [cs.LG].

[8] Johann Joseph Fux. *Gradus ad Parnassum*. Latin. Ed. by Johann Peter van Ghelen. 1966 New York Broude Bros reprint. Vienna, 1725. URL: http://vmirror.imslp.org/files/imglnks/usimg/2/22/IMSLP286170-PMLP187246-gradusadparnassu00fuxj_0.pdf.

[9] Klaus-Jürgen Sachs and Carl Dahlhaus. *Counterpoint*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.06690. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000006690.

[10] Steven G. Laitz and. *The Complete Musician. An Integrated Approach to Tonal Theory, Analysis, and Listening*. Ed. by Oxford University Press. 2007.

[11] Bill Schottstaedt. *Automatic Species Counterpoint*. Research Report STAN-M-19. System Development Foundation, CCRMA, Departement of Music, Stanford University, 1984. URL: https://ccrma.stanford.edu/files/papers/stanm19.pdf.

[12] Wikipedia Contributors. *Expert system*. Wikipedia. 2023-04-01. URL: https://en.wikipedia.org/wiki/Expert_system.

[13] Russell Ovans and Rod Davison. *An Interactive Constraint-Based Expert Assistant for Music Composition*. Research Report. Expert Systems Lab, Centre for Systems Science, Simon Fraser University, 1992. URL: https://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=36F6D0C404D11E0ABF5D566E1E9294D2?doi=10.1.1.53.1060&rep=rep1&type=pdf.

[14] Francois Pachet and Pierre Roy. "Musical Harmonization with Constraints: A Survey". In: *Constraints* (2001). URL: https://www.francoispachet.fr/wp-content/uploads/2021/01/pachet-01-Musical_Harmonization_with_Constraints.pdf.

[15] Örjan Sandred. "Constraint-Solving Systems in Music Creation". In: *Handbook of Artificial Intelligence for Music*. Springer International Publishing, 2021. URL: https://www.springerprofessional.de/constraint-solving-systems-in-music-creation/19323922.

[16] Jean-Pierre Briot and François Pachet. "Deep learning for music generation: challenges and directions". In: *Neural Computing and Applications* (2020). URL: https://doi.org/10.1007/s00521-018-3813-6.

[17] Cheng-Zhi Anna Huang et al. *Music Transformer*. 2018. arXiv: 1809.04281 [cs.LG].

[18] Cheng-Zhi Anna Huang et al. *Counterpoint by Convolution*. 2019. arXiv: 1903.07227 [cs.LG].

[19] Baptiste Lapière. "Computer-aided musical composition Constraint programming and music". Prom. by Peter Van Roy. MA thesis. Ecole polytechnique de Louvain, Université catholique de Louvain, 2020.

[20] Damien Sprockeels. "Melodizer: A Constraint Programming Tool For Computer-aided Musical Composition". Prom. by Peter Van Roy. MA thesis. Ecole polytechnique de Louvain, Université catholique de Louvain, 2021.

[21] Clément Chardon, Amaury Diels, and Federico Gobbi. "Melodizer 2.0: A Constraint Programming Tool For Computer-aided Musical Composition". Prom. by Peter Van Roy. MA thesis. Ecole polytechnique de Louvain, Université catholique de Louvain, 2022.

[22] Simonne Chevalier. *Gradus ad Parnassum. Johann Joseph Fux*. French. Ed. by Gabriel Foucou. Trans. Latin by Simmone Chevalier. 2019. ISBN: 978-2-9556093-6-1.

[23] Alfred Mann. *The Study of Counterpoint. From Johann Joseph Fux's Gradus ad Parnassum*. English. Ed. and trans. Latin by Alfred Mann. Revised Edition. 500 Fifth Avenue, New York, N.Y. 10110: W.W. Norton & Company, 1971. ISBN: 0-393-00277-2. URL: http://www.opus28.co.uk/Fux_Gradus.pdf.

[24] Christian Shulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode*. Gecode. May 28, 2019. URL: https://www.gecode.org/doc-latest/MPG.pdf.

[25] Coralie Diatkine. *OpenMusic Documentation*. IRCAM. June 21, 2011. URL: https://support.ircam.fr/docs/om/om6-manual/co/OM-Documentation.html.

[26] Thibault Wafflard. *First Species Counterpoint - A (aeolian) mode. From the Gradus ad Parnassum of Johann Joseph Fux*. Noteflight. 2023. URL: https://www.noteflight.com/scores/view/a60f5776648f8042aa5e49c7320a6b6b839471be.

[27] Thibault Wafflard. *Evaluation of a Constraint Programming Based Tool designed for Counterpoint*. Thibault Wafflard, Youtube. May 28, 2023. URL: https://youtu.be/9yB4OGr4Cgk.

[28] Thibault Wafflard. *Fifth Species Counterpoint - D (dorian) mode. From the Gradus ad Parnassum of Johann Joseph Fux*. Noteflight. 2023. URL: https://www.noteflight.com/scores/view/f75c61263ee5a50fa0d6941b8803a2b79aaed759.

[29] Wikipedia Contributors. *MIDI*. Wikipedia. Jan. 3, 2023. URL: https://en.wikipedia.org/wiki/MIDI.

[30] Grove Music Online Contributors. *Step*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.26686. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000026686.

[31] Wikipedia Contributors. *Steps and skips*. Wikipedia. 2022-08-20. URL: https://en.wikipedia.org/wiki/Steps_and_skips.

[32] Grove Music Online Contributors. *Mordent*. 2001. DOI: 10.1093/gmo/9781561592630.article.48831. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000048831.

[33] William Drabkin. *Degree*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.07408. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000007408.

[34] Greer Garden and Robert Donington. *Diminution*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.42071. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000042071.

[35] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Ed. by Elsevier. First Edition. 2006. URL: https://www.dcs.gla.ac.uk/~pat/cpM/papers/CP_Handbook-20060315-final.pdf.

[36] Grove Music Online Contributors. *Leading note*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.16179. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000016179.

[37] Dave Wave. *What is a Tritone? Tritone Explained in 2 Minutes (Music Theory)*. Dave Wave, Youtube. July 16, 2018. URL: https://youtu.be/JJIO-Jr0E8o.

[38] William Drabkin. *Tritone*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.28403. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000028403.

[39] Giovanni Battista Martini. *Esemplare a sia saggio fondamentale pratico di contrappunto sopra il canto fermo*. Bologne, 1774.

[40] Jean-Louis Fabre. *Le contrepoint, les règles mélodiques et les règles harmoniques*. French. Gradus ad Parnassum, Youtube. Feb. 27, 2020. URL: https://youtu.be/O5gbaT8sv-U?t=71.

[41] Gradus ad Parnassum website Authors. *Le Professeur*. French. 2022. URL: https://www.gradusadparnassum.fr/accueil_professeur.php.

[42] Noël Gallon and Marcel Bitsch. *Traité De Contrepoint*. French. Ed. by Durand et Cie. Paris, 1964. URL: https://www.scribd.com/document/410592892/TRAITE-DE-CONTREPOINT-Noel-Gallon-Marcel-Bitsch-pdf.

[43] Marcel Dupré. *Cours de Contrepoint*. French. Ed. by Alphonse Leduc. Paris, 1957. URL: https://www.scribd.com/doc/47338178/10063033-Cours-de-Contrepoint-Marcel-Dupre.

[44] Wikipedia Contributors. *Seventh chord*. Wikipedia. 2022-10-19. URL: https://en.wikipedia.org/wiki/Seventh_chord.

[45] Thibault Wafflard. *Second Species Counterpoint - A (aeolian) mode. From the Gradus ad Parnassum of Johann Joseph Fux*. Noteflight. 2023. URL: https://www.noteflight.com/scores/view/932a33c1d4e0c55f2a934706c4ed2f83cc28885d.

[46] William Drabkin. *Minor (i)*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.18743. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000018743.

[47] Wikipedia Contributors. *Octave*. Wikipedia. 2023-01-07. URL: https://en.wikipedia.org/wiki/Octave.

[48] Thibault Wafflard. *Third Species Counterpoint - F (lydian) mode. From the Gradus ad Parnassum of Johann Joseph Fux*. Noteflight. 2023. URL: https://www.noteflight.com/scores/view/0c58287f0fab7274cc266df98b627f743ed1ee45.

[49] Pierre Denis. *Traité De Composition Musicale. Fait par le célèbre Fux*. French. Ed. by Diod Bijoutier and Garnier & Cadet. Trans. Latin by Pierre Denis. Paris, 1773. URL: http://vmirror.imslp.org/files/imglnks/usimg/b/b2/IMSLP231222-PMLP187246-fux_traite_de_composition_1773.pdf.

[50] Unidentified Translator. *Pratical Rules for Learning Composition. Translated from a Work intitled Gradus ad Parnassum written originally in Latin by John Joseph Feux*. English. Ed. by Welcker. London, ca. 1750. URL: http://vmirror.imslp.org/files/imglnks/usimg/3/31/IMSLP370587-PMLP187246-practicalrulesfo00fuxj.pdf.

[51] Thibault Wafflard. *Fourth Species Counterpoint - E (phrygian) mode. From the Gradus ad Parnassum of Johann Joseph Fux*. Noteflight. 2023. URL: https://www.noteflight.com/scores/view/146b72d15525d0e7f98aff9b63c6a99185e9c911.

[52] William Drabkin. *Octave (i)*. Oxford University Press. 2001. DOI: 10.1093/gmo/9781561592630.article.50054. URL: https://www.oxfordmusiconline.com/grovemusic/view/10.1093/gmo/9781561592630.001.0001/omo-9781561592630-e-0000050054.

[53] Julien Freund. "I. La neutralité axiologique". French. In: *Études sur Max Weber*. Ed. by Librairie Droz. Genève, 1990, pp. 11–69. ISBN: 9782600041263. URL: https://www.cairn.info/etudes-sur-max-weber--9782600041263-p-11.htm.

[54] Damien Sprockeels et al. "A constraint formalization of Fux's counterpoint". In: *Actes des Journées d'Informatique Musicale 2023*. Journées d'Informatique Musicale 2023 (May 23–26, 2023). CICM, Laboratoire Musidanse, Université Paris 8, MSH Paris Nord. 2023. URL: https://jim2023.sciencesconf.org/data/pages/3_2_SPROCKEELS_ET_AL.pdf.

[55] Wikipedia Contributors. *Code smell*. Wikipedia. 2023-02-12. URL: https://en.wikipedia.org/wiki/Code_smell.

[56] Refactoring.Guru website Authors. *Code Smells*. Refactoring.Guru. May 30, 2023. URL: https://refactoring.guru/fr/refactoring/smells.

Appendix A

Transcriptions

> "Ça ce n'est pas bien, j'ai trois fois sol, même deux fois je m'en prive. Alors bon, exceptionnellement je peux permettre de temps en temps d'avoir deux fois la même note mais c'est vrai que dans les traités tels qu'on les utilise, ceux de par exemple: Marcel Bitsch, Marcel Dupré, les traités du XIXème siècle, on évite, enfin on proscrit même la répétition de la note. Bon et bien ça c'est une règle de bon sens en fait. Ce n'est pas une règle imposée comme ça de manière arbitraire. C'est que le contrepoint doit être une ligne en perpétuel mouvement [...]. Attention, chez Fux il le fait, donc c'est intéressant de voir que lui se permet ce genre de choses." Fabre [Jean-Louis Fabre's opinion on the repetition of the same note in counterpoint. 40, 1min 11]

Transcription A.1: French transcription of the video *Le contrepoint, les règles mélodiques et les règles harmoniques* for rule **1.P2**.

Which can be translated as:

> This is not good, I have three times G, even twice I do not use it. So, exceptionally, I can allow from time to time to have the same note twice, but it is true that in the treatises as we use them, those of for example: Marcel Bitsch, Marcel Dupré, the treatises of the XIXth century, we avoid, well we even proscribe the repetition of the note. Well, this is a rule of common sense in fact. It is not a rule imposed arbitrarily. It is that the counterpoint must be a line in perpetual movement [...]. Mind you, Fux does this, so it's interesting to see that he allows himself this kind of thing.

Transcription A.2: English translation of the above quotation A.1.

> "[...] s'il arrive que cinq noires se suivent par degrés conjoints, soit en montant soit en descendant, la première doit être consonante, la deuxième peut être dissonante, la troisième à nouveau nécessairement consonante, la quatrième pourra être dissonante si la cinquième est une consonance;"

Transcription A.3: Original text from Chevalier [22, p.73] for rule **3.H1**.

> "Tertia Contrapuncti Species est quatuor semiminimarum contra unam semibrevem Compositio. Ubi principiò animadvertendum est, quòd, si quinque semiminimas vei ascendendo, vel descendendo **continuò gradatim** se sequi contingat, prima Consonans esse debeat, secunda dissonans esse possit. Tertia denuo Consonans sit, necesse est. Quarta dissonans esse poterit, **si** quinta Consonantia fuerit;"

Transcription A.4: Original text from Fux [8, p.63-64] for rule **3.H1**.

Appendix B

Additional Material

Range	-1	0	1	2	3	4	5	6	7	8	9
C	0	12	24	36	48	60	72	84	96	108	120
$C\sharp$ / $D\flat$	1	13	25	37	49	61	73	85	97	109	121
D	2	14	26	38	50	62	74	86	98	110	122
$D\sharp$ / $E\flat$	3	15	27	39	51	63	75	87	99	111	123
E	4	16	28	40	52	64	76	88	100	112	124
F	5	17	29	41	53	65	77	89	101	113	125
$F\sharp$ / $G\flat$	6	18	30	42	54	66	78	90	102	114	126
G	7	19	31	43	55	67	79	91	103	115	127
$G\sharp$ / $A\flat$	8	20	32	44	56	68	80	92	104	116	-
A	9	21	33	45	57	69	81	93	105	117	-
$A\sharp$ / $B\flat$	10	22	34	46	58	70	82	94	106	118	-
B	11	23	35	47	59	71	83	95	107	119	-

Table B.1: MIDI note values.

Figure B.1: 5$^{\text{th}}$ species upper ctp. of Fux (above) vs. upper ctp. of the solver [2.690 s] (below).

Figure B.2: Solver-generated 5th species "ctp." with a chromatic "*cantus firmus*".

Appendix C

User Guide

This user guide provides a overview of FuxCP, covering its installation process, usage within OpenMusic, and a description of the costs displayed in the interface. While FuxCP is designed to be compatible with all platforms, it relies on GiL, which currently works only on MacOS and Linux. Unfortunately, GiL does not support Windows due to compatibility issues between the 32-bit Lisp license used by OpenMusic and the 64-bit Gecode Windows version. Although it is technically possible to obtain a 32-bit version of Gecode for Windows, it is not recommended.

C.1 Installing FuxCP

C.1.1 Prerequisites

To use FuxCP, it is necessary to download and install the following tools:

- Gecode on `https://www.gecode.org/download.html`
- OpenMusic on `https://openmusic-project.github.io/openmusic/`

And download the following libraries:

- GiL on `https://github.com/sprockeelsd/GiLv2.0`
- FuxCP : `https://github.com/sprockeelsd/Melodizer`

On the last github, other tools such as Melodizer and Melodizer2.0 are available. In the context of this user guide, only the FuxCP folder will be necessary.

C.1.2 Loading FuxCP in OpenMusic

To use the previous libraries, OpenMusic must be launched. Upon opening any workspace, locate the toolbar at the top of the interface. Click on the "Windows" button, highlighted in figure C.1, and select "Library" from the dropdown menu. This action will unveil a new window. In the toolbar of this window, choose "File" and then "Add remote library." Navigate through your file system to find the path where the previously downloaded FuxCP and Gil libraries are stored. Once located, the libraries should appear under the "libraries" folder in the "Library" window, as depicted in figure C.2. Right-click on "fuxcp" and select "Load Library". If no errors occur, the setup is complete.

However, if an error arises, it may be a linking issue with the Gecode library. For MacOS users, a script can be used from the `c++` folder of the gil library. Edit the path to Gecode inside the script to match your system's configuration. Linux users should add the Gecode library to the `LD_LIBRARY_PATH` variable. Go to the `/etc/ld.so.conf.d` folder and create a new `.conf` file if one does not already exist. In this file, paste the full path to the Gecode library, save it, and run `sudo ldconfig` to update the system with the new library. Don't forget to restart OpenMusic and don't stop believing. Following these steps should ensure the proper utilization of FuxCP.

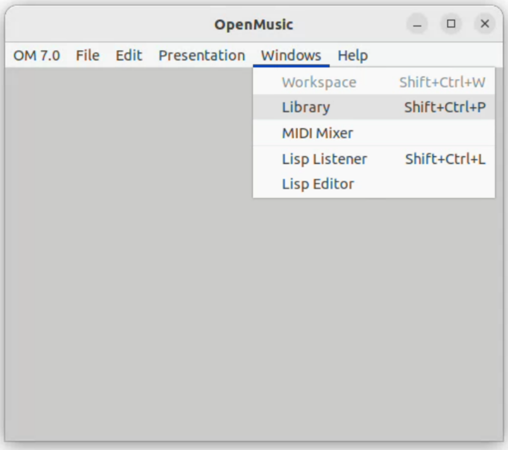


Figure C.1: Opening the "Library" window in OpenMusic.

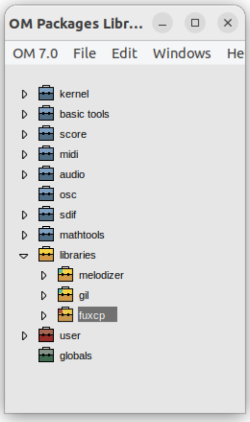


Figure C.2: Loading the "fuxcp" library in OpenMusic.

C.2 Using FuxCP in OpenMusic

It is straightforward to use FuxCP in OpenMusic. There is a single block comprising the entire graphical interface of the tool. This block or class is called `cp-params`. To load it, it is possible to type `fuxcp::cp-params` in a new patch entry; or load the block of the class by loading "cp-params" from the drop-down menu by right-clicking in the patch $(Classes \rightarrow Libraries \rightarrow FuxCP \rightarrow Solver \rightarrow CP-PARAMS)$.

Once this block has appeared, all you have to do is bind an OM voice object, representing the *cantus firmus*, to the second argument of `cp-params` as shown in figure C.3. Don't forget to block the input voice object and evaluate `cp-params` so it can detect the new input. Now `cp-params` can be blocked too. From now on, you could directly use the interface and generate counterpoints using the tool. If you want to retrieve the voice object containing the counterpoint generated by the tool, just bind the third argument on the output side to a voice object. Once bound, it is then possible to evaluate the voice object so that it updates.

But how to use the interface? Just double-click on the block to make it appear. The interface is sorted from left to right, so that the preferences are separated into three different categories: "Preferences for Melodic Intervals of…", "General Preferences",

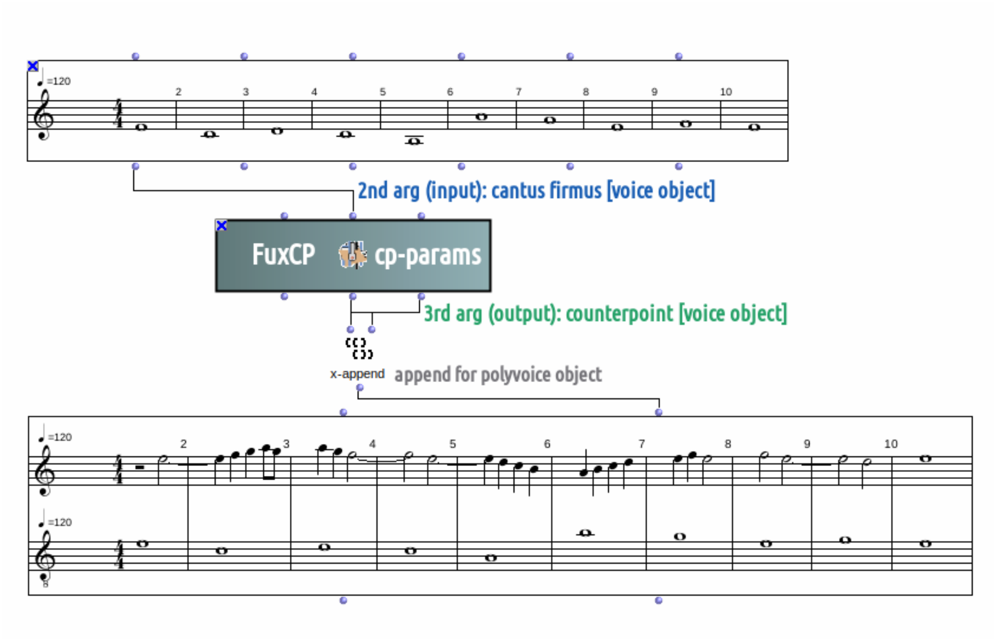


Figure C.3: View of a patch using `fuxcp::cp-params` in OpenMusic.

"Species Specific Preferences", "Solver Configuration", and in the lower right corner, "Solver Launcher" (see figure C.4). Once the preferences have been chosen, the default ones representing the style of Fux, you must save the parameters ("Save Config") in order to then be able to launch the search for a solution ("Next Solution"). This search can take a fraction of a second just as it can take tens of minutes, or even hours if the parameters chosen make the search difficult. If a search takes too long, it is always possible to stop it by clicking on "Stop". You can then either change the preferences in a way (often at the level of the costs of the melodic intervals), or increase the "Irreverence" to obtain potentially less "good" but faster solutions. The description of the parameters is available in the next section.

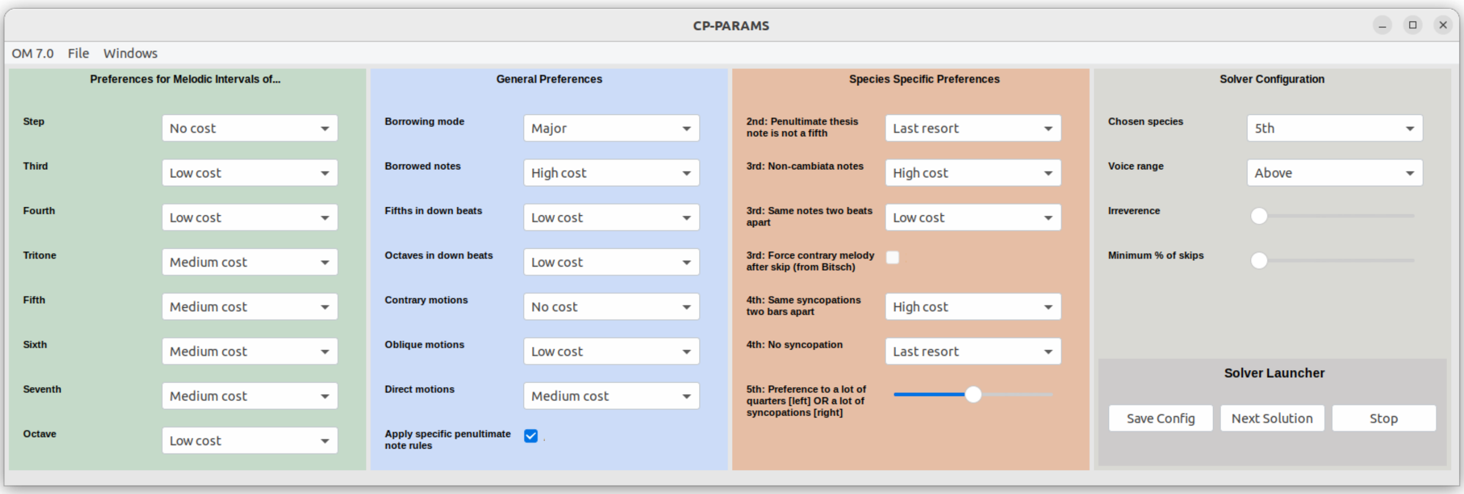


Figure C.4: User interface of the `fuxcp::cp-params` class in OpenMusic.

C.3 Interface Parameters Description

Table C.1 describes all the parameters available in the interface. A low cost represents a high preference while a high cost represents a low preference.

Name	Description	Default value
Step	Preference for melodic intervals of one step or less.	No cost
Third	Preference for melodic third skips.	Low cost
Fourth	Preference for melodic fourth leaps.	Low cost
Tritone	Preference for melodic tritone leaps.	Forbidden
Fifth	Preference for melodic fifth leaps.	Medium cost
Sixth	Preference for melodic sixth leaps.	Medium cost
Seventh	Preference for melodic seventh leaps.	Medium cost
Octave	Preference for melodic octave leaps.	Low cost
Borrowing mode	Type of scale from which notes can be borrowed to generate counterpoint. The first note of the *cantus firmus* determines the tonic of this scale. Applies everywhere except the penultimate bar.	Major
Borrowed notes	Preference for borrowed notes outside the diatonic scale. These notes are defined by the "Borrowing mode" parameter.	High cost
Fifths in down beats	Preference to have harmonic fifths on the first beat of a bar.	Low cost
Octaves in down beats	Preference to have harmonic octaves on the first beat of a bar.	Low cost
Contrary motions	Preference to have, between two bars, one voice rising while the other is falling.	No cost
Oblique motions	Preference to have, between two bars, one static voice while the other is moving.	Low cost
Direct motions	Preference to have, between two bars, the two voices going in the same direction.	Medium cost
Apply specific penultimate note rules	Force all rules on the notes of the penultimate measure. This mainly refers to the penultimate note that must harmonically be either a major sixth or a minor third depending on whether the counterpoint is above or below.	Checked
2nd: Penultimate thesis note is not a fifth	Preference for the first note of the penultimate bar to be something other than a harmonic fifth	Last resort
3rd: Non-cambiata notes	Preference for the second quarter note of a bar to be a consonance already surrounded by two consonances.	High cost
3rd: Same notes two beats apart	Preference to have the same quarter notes two beats apart. A high cost allows to avoid a certain monotony.	Low cost
3rd: Force joint contrary melody after skip	Force that a melodic skip or leap is followed by a melodic step in the opposite direction.	Unchecked
4th: Same syncopations two bars apart	Preference to have the same half notes two bars apart. A high cost allows to avoid a certain monotony.	High cost
4th: No syncopation	Preference to have distinct half notes instead of syncopations.	Last resort
5th: Preferences to a lot of quarters or a lot of syncopations	Determines the minimum percentage of quarter notes (to the left) and syncopations (to the right) in the fifth species. Pushing the slider all the way to one side is not recommended.	<center>
Chosen species	Determines the type of counterpoint that the tool will generate. From whole notes to syncopations, passing through quarter notes. The fifth species uses the rules and preferences of the previous species.	5th
Voice range	Determines around which pitch the counterpoint will be generated depending on the pitch of the first note of the *cantus firmus*.	Above
Irreverence	Artificially increases the minimum cost of the solution to obtain counterpoints that are less respectful of the established preferences. Can also be used to get solutions faster.	0
Minimum % of skips	Determines, depending on the counterpoint size, the percentage of melodic intervals larger than one step.	0%
Save Config	Saves all established preferences and allows you to start a new search for this configuration later.	-
Next Solution	Starts or continues searching for the previously saved configuration. Displays a new window with the solution when it is found. Displays an error message if no other solution can be found.	-
Stop	Pause the search. It may take up to 5 seconds.	-

Table C.1: Description of the parameters of `fuxcp::cp-params`.

Appendix D

Software Architecture

This appendix summarizes the architecture of the software. First, from the point of view of the role of FuxCP as a tool and second, from the point of view of the organization of the code in FuxCP.

As shown in figure D.1, FuxCP is an OpenMusic library that uses GiL to communicate constraints with Gecode. The solver itself therefore runs well in Gecode directly. At the level of the distribution of the files (see figure D.2), all the functions that break the constraints have been placed in a single and same file. The different species, which represent a set of rules, call these functions such that the constraints set reflect the rules of these species. This architecture is not terrible and should rely on object-oriented inheritance. Apart from that, the interface calls the main CSP creation and search functions via the `fuxcp-main.lisp` file. The latter chooses what to do, in particular according to the type of counterpoint chosen.

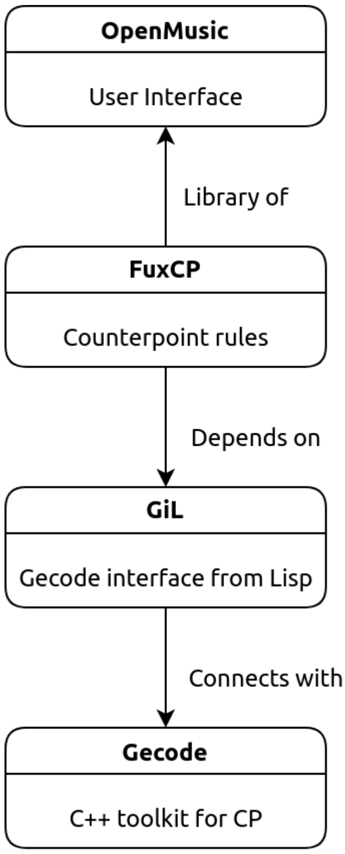


Figure D.1: Macro architecture, overview of the links between FuxCP and the other tools.

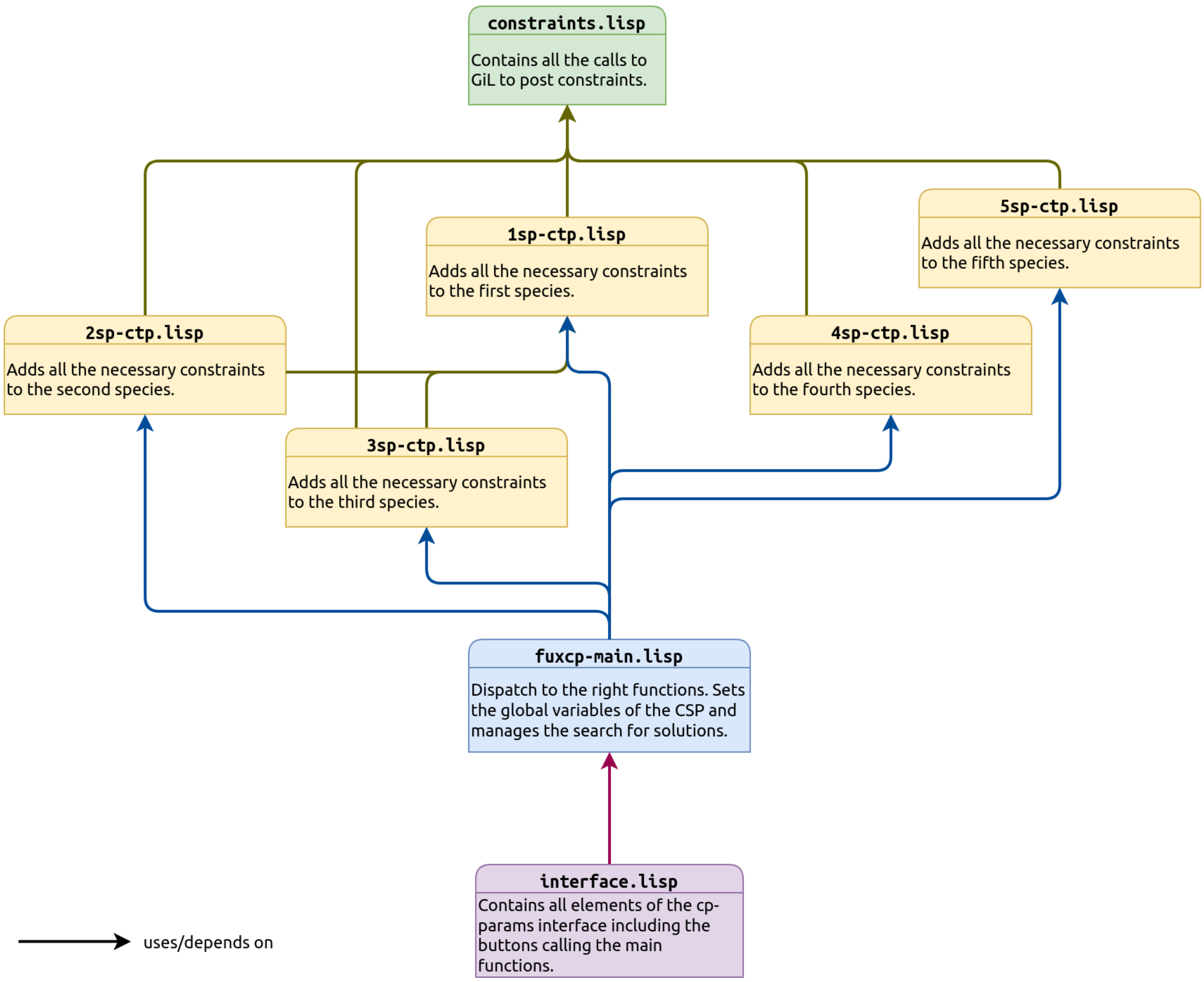


Figure D.2: Micro architecture, overview of the links between the files.

Appendix E

Source Code

E.1 FuxCP.lisp

```lisp
(in-package :om)

(defvar *fuxcp-sources-dir* nil)
(setf *fuxcp-sources-dir* (make-pathname :directory (append (pathname-directory *load-pathname*) '("sources"))))


(mapc 'compile&load (list
    (make-pathname :directory (append (pathname-directory *load-pathname*) (list "sources")) :name "package" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "utils" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "constraints" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "1sp-ctp" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "2sp-ctp" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "3sp-ctp" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "4sp-ctp" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "5sp-ctp" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "fuxcp-main" :type "lisp")
    (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "interface" :type "lisp")
))


(fill-library '(
    ("Solver" nil (fuxcp::cp-params) nil)
))

(print "FuxCP Loaded")
```

E.2 package.lisp

```lisp
(in-package :om)

(defvar *FuxCP-path* (make-pathname  :directory (append (pathname-directory *load-pathname*) (list "FuxCP"))))

(require-library "GIL")

(defpackage :fuxcp
(:use "COMMON-LISP" "OM" "CL-USER"))
```

E.3 interface.lisp

```lisp
(in-package :fuxcp)

; Author: Thibault Wafflard
; Date: June 3, 2023
; This file contains all the cp-params interface.
; That is to say the interface blocks, as well as the global variables updated via the interface.

;;;====================
;;;= cp-params OBJECT =
;;;====================

(print "Loading cp-params object...")

(om::defclass! cp-params ()
;attributes
(
    ; ---------- Input cantus firmus ----------
    (cf-voice :accessor cf-voice :initarg :cf-voice :initform nil :documentation "")
    ; ---------- Melodic parameters ----------
    (m-step-cost-param :accessor m-step-cost-param :initform "No cost" :type string :documentation "")
    (m-third-cost-param :accessor m-third-cost-param :initform "Low cost" :type string :documentation "")
    (m-fourth-cost-param :accessor m-fourth-cost-param :initform "Low cost" :type string :documentation "")
    (m-tritone-cost-param :accessor m-tritone-cost-param :initform "Forbidden" :type string :documentation "")
    (m-fifth-cost-param :accessor m-fifth-cost-param :initform "Medium cost" :type string :documentation "")
    (m-sixth-cost-param :accessor m-sixth-cost-param :initform "Medium cost" :type string :documentation "")
    (m-seventh-cost-param :accessor m-seventh-cost-param :initform "Medium cost" :type string :documentation "")
    (m-octave-cost-param :accessor m-octave-cost-param :initform "Low cost" :type string :documentation "")
    ; ---------- Global parameters (species 1) ----------
    (borrow-mode-param :accessor borrow-mode-param :initform "Major" :type string :documentation "")
    (borrow-cost-param :accessor borrow-cost-param :initform "High cost" :type string :documentation "")
    (h-fifth-cost-param :accessor h-fifth-cost-param :initform "Low cost" :type string :documentation "")
    (h-octave-cost-param :accessor h-octave-cost-param :initform "Low cost" :type string :documentation "")
    (con-motion-cost-param :accessor con-motion-cost-param :initform "No cost" :type string :documentation "")
    (obl-motion-cost-param :accessor obl-motion-cost-param :initform "Low cost" :type string :documentation "")
    (dir-motion-cost-param :accessor dir-motion-cost-param :initform "Medium cost" :type string :documentation "")
    (penult-rule-check-param :accessor penult-rule-check-param :initform t :type boolean :documentation "")
    ; ---------- Species parameters ----------
    ; Species 2
    (penult-sixth-cost-param :accessor penult-sixth-cost-param :initform "Last resort" :type string :documentation "")
    ; Species 3
    (non-cambiata-cost-param :accessor non-cambiata-cost-param :initform "High cost" :type string :documentation "")
    (two-beats-apart-cost-param :accessor two-beats-apart-cost-param :initform "Low cost" :type string :documentation "")
    (con-m-after-skip-check-param :accessor con-m-after-skip-check-param :initform nil :type boolean :documentation "")
    ; Species 4
```

```
    (two-bars-apart-cost-param :accessor two-bars-apart-cost-param :initform "High cost" :type
        string :documentation "")
    (no-syncopation-cost-param :accessor no-syncopation-cost-param :initform "Last resort" :type
         string :documentation "")
    ; Species 5
    (pref-species-slider-param :accessor pref-species-slider-param :initform 50 :type integer :
        documentation "")
    ; ---------- Solver parameters ----------
    (species-param :accessor species-param :initform "5th" :type string :documentation "")
    (voice-type-param :accessor voice-type-param :initform "Above" :type string :documentation "
        ")
    (irreverence-slider-param :accessor irreverence-slider-param :initform 0 :type integer :
        documentation "")
    (min-skips-slider-param :accessor min-skips-slider-param :initform 0 :type integer :
        documentation "")
    ; ---------- Output & Stop ----------
    (current-csp :accessor current-csp :initform nil :documentation "")
    (result-voice :accessor result-voice :initarg :result-voice :initform nil :documentation "")
)
    (:icon 225)
    (:documentation "This class implements FuxCP.
    FuxCP is a constraints programming based tool aiming to generate counterpoints based on
        cantus firmus.")
)

; the editor for the object
(defclass params-editor (om::editorview) ())

(defmethod om::class-has-editor-p ((self cp-params)) t)
(defmethod om::get-editor-class ((self cp-params)) 'params-editor)

(defmethod om::om-draw-contents ((view params-editor))
    (let* ((object (om::object view)))
        (om::om-with-focused-view 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*)
)

; To access the melodizer object, (om::object self)
(defmethod initialize-instance ((self params-editor) &rest args)
    ;;; do what needs to be done by default
    (call-next-method) ; start the search by default?
    (make-my-interface self)
)

; function to create the tool's interface
(defmethod make-my-interface ((self params-editor))
    (print "Creating interface...")
    ; create the main view of the object
    (make-main-view self)

    (let*
        (
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        ;;; setting the different regions of the tool ;;;
        ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
        (melodic-params-panel (om::om-make-view 'om::om-view
            :size (om::om-make-point 400 450)
            :position (om::om-make-point 5 5)
            :bg-color om::*azulito*)
        )
        (general-params-panel (om::om-make-view 'om::om-view
```

```
            :size (om::om-make-point 400 450)
            :position (om::om-make-point 410 5)
            :bg-color om::*azulote*)
        )
        (species-params-panel (om::om-make-view 'om::om-view
            :size (om::om-make-point 400 450)
            :position (om::om-make-point 815 5)
            :bg-color (om::make-color-255 230 190 165))
        )
        (search-params-panel (om::om-make-view 'om::om-view
            :size (om::om-make-point 400 450)
            :position (om::om-make-point 1220 5)
            :bg-color om::*maq-color*)
        )
        (search-buttons (om::om-make-view 'om::om-view
            :size (om::om-make-point 390 120)
            :position (om::om-make-point 1225 330)
            :bg-color om::*workspace-color*)
        )
        )

        (make-general-params-panel self general-params-panel)
        (make-melodic-params-panel self melodic-params-panel)
        (make-species-params-panel self species-params-panel)
        (make-search-params-panel self search-params-panel)
        (make-search-buttons self search-buttons)
        ; ; add the subviews for the different parts into the main view
        (om::om-add-subviews
            self
            search-buttons
            search-params-panel
            species-params-panel
            general-params-panel
            melodic-params-panel
        )
    )
    ; return the editor
    self
)

(defun make-melodic-params-panel (editor melodic-params-panel)
    (om::om-add-subviews
        melodic-params-panel
        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 90 2)
        (om::om-make-point 220 20)
        "Preferences for Melodic Intervals of..."
        :font om::*om-default-font2b*
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 50)
        (om::om-make-point 150 20)
        "Step"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 50)
        (om::om-make-point 200 20)
        "Step"
        :range (costs-list t)
        :value (m-step-cost-param (om::object editor))
        :di-action #'(lambda (cost)
```

```
            (setf (m-step-cost-param (om::object editor)) (nth (om::om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 100)
        (om::om-make-point 150 20)
        "Third"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 100)
        (om::om-make-point 200 20)
        "Third"
        :range (costs-list)
        :value (m-third-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (m-third-cost-param (om::object editor)) (nth (om::om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 150)
        (om::om-make-point 150 20)
        "Fourth"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 150)
        (om::om-make-point 200 20)
        "Fourth"
        :range (costs-list)
        :value (m-fourth-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (m-fourth-cost-param (om::object editor)) (nth (om::om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 200)
        (om::om-make-point 150 20)
        "Tritone"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 200)
        (om::om-make-point 200 20)
        "Tritone"
        :range (costs-list)
        :value (m-tritone-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (m-tritone-cost-param (om::object editor)) (nth (om::om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )
```

```lisp

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 250)
        (om::om-make-point 150 20)
        "Fifth"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 250)
        (om::om-make-point 200 20)
        "Fifth"
        :range (costs-list)
        :value (m-fifth-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (m-fifth-cost-param (om::object editor)) (nth (om::om-get-selected-item-index
                cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 300)
        (om::om-make-point 150 20)
        "Sixth"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 300)
        (om::om-make-point 200 20)
        "Sixth"
        :range (costs-list)
        :value (m-sixth-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (m-sixth-cost-param (om::object editor)) (nth (om::om-get-selected-item-index
                cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 350)
        (om::om-make-point 150 20)
        "Seventh"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 350)
        (om::om-make-point 200 20)
        "Seventh"
        :range (costs-list)
        :value (m-seventh-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (m-seventh-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 400)
```

```
        (om::om-make-point 150 20)
        "Octave"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 400)
        (om::om-make-point 200 20)
        "Octave"
        :range (costs-list)
        :value (m-octave-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (m-octave-cost-param (om::object editor)) (nth (om::om-get-selected-item-index
                 cost) (om::om-get-item-list cost)))
        )
        )
    )
)

(defun make-general-params-panel (editor general-params-panel)
    (om::om-add-subviews
        general-params-panel
        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 140 2)
        (om::om-make-point 220 20)
        "General Preferences"
        :font om::*om-default-font2b*
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 50)
        (om::om-make-point 150 20)
        "Borrowing mode"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 50)
        (om::om-make-point 200 20)
        "Borrowing mode"
        :range (list "None" "Major" "Minor")
        :value (borrow-mode-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (borrow-mode-param (om::object editor)) (nth (om::om-get-selected-item-index
                cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 100)
        (om::om-make-point 150 20)
        "Borrowed notes"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 100)
        (om::om-make-point 200 20)
        "Borrowed notes"
        :range (costs-list t)
        :value (borrow-cost-param (om::object editor))
```

```lisp
        :di-action #'(lambda (cost)
            (setf (borrow-cost-param (om::object editor)) (nth (om::om-get-selected-item-index
                cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 150)
        (om::om-make-point 150 20)
        "Fifths in down beats"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 150)
        (om::om-make-point 200 20)
        "Fifths in down beats"
        :range (costs-list t)
        :value (h-fifth-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (h-fifth-cost-param (om::object editor)) (nth (om::om-get-selected-item-index
                cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 200)
        (om::om-make-point 150 20)
        "Octaves in down beats"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 200)
        (om::om-make-point 200 20)
        "Octaves in down beats"
        :range (costs-list t)
        :value (h-octave-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (h-octave-cost-param (om::object editor)) (nth (om::om-get-selected-item-index
                 cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 250)
        (om::om-make-point 150 20)
        "Contrary motions"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 250)
        (om::om-make-point 200 20)
        "Contrary motions"
        :range (costs-list t)
        :value (con-motion-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (con-motion-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
```

```
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 300)
        (om::om-make-point 150 20)
        "Oblique motions"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 300)
        (om::om-make-point 200 20)
        "Oblique motions"
        :range (costs-list)
        :value (obl-motion-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (obl-motion-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 350)
        (om::om-make-point 150 20)
        "Direct motions"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 350)
        (om::om-make-point 200 20)
        "Direct motions"
        :range (costs-list t)
        :value (dir-motion-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (dir-motion-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 400)
        (om::om-make-point 150 20)
        "Apply specific penultimate note rules"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::om-check-box
        (om::om-make-point 170 400)
        (om::om-make-point 20 20)
        "Apply specific penultimate note rules"
        ::checked-p (penult-rule-check-param (om::object editor))
        :di-action #'(lambda (c)
            (if (om::om-checked-p c)
                (setf (penult-rule-check-param (om::object editor)) t)
                (setf (penult-rule-check-param (om::object editor)) nil)
            )
        )
        )
    )
)
```

```
(defun make-species-params-panel (editor species-params-panel)
    (om::om-add-subviews
        species-params-panel
        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 130 2)
        (om::om-make-point 220 20)
        "Species Specific Preferences"
        :font om::*om-default-font2b*
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 50)
        (om::om-make-point 150 20)
        "2nd: Penultimate thesis note is not a fifth"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 50)
        (om::om-make-point 200 20)
        "2nd: Penultimate thesis note is not a fifth"
        :range (costs-list t)
        :value (penult-sixth-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (penult-sixth-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 100)
        (om::om-make-point 150 20)
        "3rd: Non-cambiata notes"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 100)
        (om::om-make-point 200 20)
        "3rd: Non-cambiata notes"
        :range (costs-list t)
        :value (non-cambiata-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (non-cambiata-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 150)
        (om::om-make-point 150 20)
        "3rd: Same notes two beats apart"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 150)
        (om::om-make-point 200 20)
        "3rd: Same notes two beats apart"
```

```
        :range (costs-list)
        :value (two-beats-apart-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (two-beats-apart-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 200)
        (om::om-make-point 150 20)
        "3rd: Force joint contrary melody after skip (from Bitsch)"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::om-check-box
        (om::om-make-point 170 200)
        (om::om-make-point 20 20)
        "3rd: Force joint contrary melody after skip (from Bitsch)"
        ::checked-p (con-m-after-skip-check-param (om::object editor))
        :di-action #'(lambda (c)
            (if (om::om-checked-p c)
                (setf (con-m-after-skip-check-param (om::object editor)) t)
                (setf (con-m-after-skip-check-param (om::object editor)) nil)
            )
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 250)
        (om::om-make-point 150 20)
        "4th: Same syncopations two bars apart"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 250)
        (om::om-make-point 200 20)
        "4th: Same syncopations two bars apart"
        :range (costs-list)
        :value (two-bars-apart-cost-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (two-bars-apart-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 300)
        (om::om-make-point 150 20)
        "4th: No syncopation"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 300)
        (om::om-make-point 200 20)
        "4th: No syncopation"
        :range (costs-list t)
        :value (no-syncopation-cost-param (om::object editor))
        :di-action #'(lambda (cost)
```

```
            (setf (no-syncopation-cost-param (om::object editor)) (nth (om::
                om-get-selected-item-index cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 350)
        (om::om-make-point 150 50)
        "5th: Preference to a lot of quarters [left] OR a lot of syncopations [right]"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::om-slider
        (om::om-make-point 170 350)
        (om::om-make-point 200 20)
        "5th: Preference to a lot of quarters [left] OR a lot of syncopations [right]"
        :range '(0 100)
        :increment 1
        :value (pref-species-slider-param (om::object editor))
        :di-action #'(lambda (s)
            (setf (pref-species-slider-param (om::object editor)) (om::om-slider-value s))
        )
        )
    )
)

(defun make-search-params-panel (editor search-params-panel)
    (om::om-add-subviews
        search-params-panel
        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 140 2)
        (om::om-make-point 200 20)
        "Solver Configuration"
        :font om::*om-default-font2b*
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 50)
        (om::om-make-point 150 20)
        "Chosen species"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 50)
        (om::om-make-point 200 20)
        "Chosen species"
        :range (list "1st" "2nd" "3rd" "4th" "5th")
        :value (species-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (species-param (om::object editor)) (nth (om::om-get-selected-item-index cost)
                 (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 100)
        (om::om-make-point 150 20)
        "Voice range"
        :font om::*om-default-font1b*
        )
```

```
        (om::om-make-dialog-item
        'om::pop-up-menu
        (om::om-make-point 170 100)
        (om::om-make-point 200 20)
        "Voice range"
        :range (list "Really far above" "Far above" "Above" "Same range" "Below" "Far below" "
            Really far below")
        :value (voice-type-param (om::object editor))
        :di-action #'(lambda (cost)
            (setf (voice-type-param (om::object editor)) (nth (om::om-get-selected-item-index
                cost) (om::om-get-item-list cost)))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 150)
        (om::om-make-point 150 20)
        "Irreverence"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::om-slider
        (om::om-make-point 170 150)
        (om::om-make-point 200 20)
        "Irreverence"
        :range '(0 40)
        :increment 1
        :value (irreverence-slider-param (om::object editor))
        :di-action #'(lambda (s)
            (setf (irreverence-slider-param (om::object editor)) (om::om-slider-value s))
        )
        )

        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 15 200)
        (om::om-make-point 150 20)
        "Minimum % of skips"
        :font om::*om-default-font1b*
        )

        (om::om-make-dialog-item
        'om::om-slider
        (om::om-make-point 170 200)
        (om::om-make-point 200 20)
        "Minimum % of skips"
        :range '(0 100)
        :increment 1
        :value (min-skips-slider-param (om::object editor))
        :di-action #'(lambda (s)
            (setf (min-skips-slider-param (om::object editor)) (om::om-slider-value s))
        )
        )
    )
)

(defun make-search-buttons (editor search-buttons)
    (om::om-add-subviews
        search-buttons
        (om::om-make-dialog-item
        'om::om-static-text
        (om::om-make-point 140 5)
        (om::om-make-point 150 20)
        "Solver Launcher"
```

```
        :font om::*om-default-font3b*
        )

        (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 10 50) ; position (horizontal, vertical)
        (om::om-make-point 120 20) ; size (horizontal, vertical)
        "Save Config"
        :di-action #'(lambda (b)
            (if (null (cf-voice (om::object editor))); if the problem is not initialized
                (error "No voice has been given to the solver. Please set a cantus firmus into the second input and try again.")
            )
            (set-global-cf-variables
                (cf-voice (om::object editor))
                (convert-to-voice-integer (voice-type-param (om::object editor)))
                (borrow-mode-param (om::object editor))
            )
            (defparameter *params* (make-hash-table))
            ;; set melodic parameters
            (setparam-cost 'm-step-cost (m-step-cost-param (om::object editor)))
            (setparam-cost 'm-third-cost (m-third-cost-param (om::object editor)))
            (setparam-cost 'm-fourth-cost (m-fourth-cost-param (om::object editor)))
            (setparam-cost 'm-tritone-cost (m-tritone-cost-param (om::object editor)))
            (setparam-cost 'm-fifth-cost (m-fifth-cost-param (om::object editor)))
            (setparam-cost 'm-sixth-cost (m-sixth-cost-param (om::object editor)))
            (setparam-cost 'm-seventh-cost (m-seventh-cost-param (om::object editor)))
            (setparam-cost 'm-octave-cost (m-octave-cost-param (om::object editor)))
            ;; set general parameters
            (setparam 'borrow-mode (borrow-mode-param (om::object editor)))
            (setparam-cost 'borrow-cost (borrow-cost-param (om::object editor)))
            (setparam-cost 'h-fifth-cost (h-fifth-cost-param (om::object editor)))
            (setparam-cost 'h-octave-cost (h-octave-cost-param (om::object editor)))
            (setparam-cost 'con-motion-cost (con-motion-cost-param (om::object editor)))
            (setparam-cost 'obl-motion-cost (obl-motion-cost-param (om::object editor)))
            (setparam-cost 'dir-motion-cost (dir-motion-cost-param (om::object editor)))
            (setparam 'penult-rule-check (penult-rule-check-param (om::object editor)))
            ;; set species specific parameters
            (setparam-cost 'penult-sixth-cost (penult-sixth-cost-param (om::object editor)))
            (setparam-cost 'non-cambiata-cost (non-cambiata-cost-param (om::object editor)))
            (setparam-cost 'two-beats-apart-cost (two-beats-apart-cost-param (om::object editor)))
            (setparam 'con-m-after-skip-check (con-m-after-skip-check-param (om::object editor)))
            (setparam-cost 'two-bars-apart-cost (two-bars-apart-cost-param (om::object editor)))
            (setparam-cost 'no-syncopation-cost (no-syncopation-cost-param (om::object editor)))
            (setparam-slider 'pref-species-slider (pref-species-slider-param (om::object editor)))
            ;; set search parameters
            (setparam-slider 'irreverence-slider (irreverence-slider-param (om::object editor)))
            (setparam-slider 'min-skips-slider (min-skips-slider-param (om::object editor)))
            (setf (current-csp (om::object editor)) (fux-cp (convert-to-species-integer (species-param (om::object editor)))))
        )
        )

        (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 135 50) ; position
        (om::om-make-point 120 20) ; size
        "Next Solution"
        :di-action #'(lambda (b)
            (if (typep (current-csp (om::object editor)) 'null); if the problem is not initialized
                (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
            (setparam 'is-stopped nil)
            ;get the next solution
            (mp:process-run-function ; start a new thread for the execution of the next method
                "solver-thread" ; name of the thread, not necessary but useful for debugging
                nil ; process initialization keywords, not needed here
                (lambda () ; function to call
                    (setf
                        (result-voice (om::object editor))
                        (search-next-fux-cp (current-csp (om::object editor)))
                    )
                    (om::openeditorframe ; open a voice window displaying the solution
                        (om::omNG-make-new-instance (result-voice (om::object editor)) "Current solution")
                    )
                )
            )
        )
        )

        (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 260 50) ; position (horizontal, vertical)
        (om::om-make-point 120 20) ; size (horizontal, vertical)
        "Stop"
        :di-action #'(lambda (b)
            (setparam 'is-stopped t)
        )
        )
  )
)

; return the list of available costs for the preferences
; @is-required: if true, "Forbidden" is removed
(defun costs-list (&optional (is-required nil))
    (let (
        (costs (list "No cost" "Low cost" "Medium cost" "High cost" "Last resort" "Cost prop. to length" "Forbidden"))
    )
        (if is-required
            (butlast costs)
            costs
        )
    )
)

; set the value @v in the hash table @h with key @k
(defun seth (h k v)
    (setf (gethash k h) v)
)

; set the value @v in the parameters with key @k
(defun setparam (k v)
    (seth *params* k v)
)

; set the cost-converted value @of v in the parameters with key @k
(defun setparam-cost (k v)
    (setparam k (convert-to-cost-integer v))
)

; set the species-converted value @of v in the parameters with key @k
(defun setparam-species (k v)
    (setparam k (convert-to-species-integer v))
)

```

```
; set the slider-converted value @of v in the parameters with key @k
(defun setparam-slider (k v)
    (setparam k (convert-to-percent v))
)

; convert a cost to an integer
(defun convert-to-cost-integer (param)
    (cond
    ((equal param "No cost") 0)
    ((equal param "Low cost") 1)
    ((equal param "Medium cost") 2)
    ((equal param "High cost") 4)
    ((equal param "Last resort") 8)
    ((equal param "Cost prop. to length") (* 2 *cf-len))
    ((equal param "Forbidden") (* 64 *cf-len))
    )
)

; convert a species to an integer
(defun convert-to-species-integer (param)
    (cond
    ((equal param "1st") 1)
    ((equal param "2nd") 2)
    ((equal param "3rd") 3)
    ((equal param "4th") 4)
    ((equal param "5th") 5)
    )
)

;; convert the string for the voice type to an integer
;; belong to {"Really far above" "Far above" "Above" "Same range" "Below" "Far below" "Really
    far below"}
;; convert to {-3 -2 -1 0 1 2 3}
(defun convert-to-voice-integer (param)
    (cond
    ((equal param "Really far above") 3)
    ((equal param "Far above") 2)
    ((equal param "Above") 1)
    ((equal param "Same range") 0)
    ((equal param "Below") -1)
    ((equal param "Far below") -2)
    ((equal param "Really far below") -3)
    )
)

; convert a slider value to a percentage
(defun convert-to-percent (param)
    (float (/ param 100))
)

; convert a mode to an integer
(defun convert-to-mode-integer (param tone)
    (cond
    ((equal param "Major") (mod tone 12))
    ((equal param "Minor") (mod (+ tone 3) 12))
    ((equal param "None") nil)
    )
)

; define all the global variables
(defun set-global-cf-variables (cantus-firmus voice-type borrow-mode)
    ; Lower bound and upper bound related to the cantus firmus pitch
    (defparameter VOICE_TYPE voice-type)
    (defparameter RANGE_UB (+ 12 (* 6 VOICE_TYPE)))
    (defparameter RANGE_LB (+ -6 (* 6 VOICE_TYPE)))
    (defparameter *prev-sol-check nil)
    (defparameter rythmic+pitches nil)
```

```
    (defparameter rythmic-om nil)
    (defparameter pitches-om nil)
    ; get the tonalite of the cantus firmus
    (defparameter *tonalite-offset (get-tone-offset cantus-firmus))
    ; get the *scale of the cantus firmus
    (defparameter *scale (build-scaleset (get-scale) *tonalite-offset))
    ; *chromatic *scale
    (defparameter *chromatic-scale (build-scaleset (get-scale "chromatic") *tonalite-offset))
    ; get the first note of each chord of the cantus firmus
    (defparameter *cf (mapcar #'first (to-pitch-list (om::chords cantus-firmus))))
    ; get the tempo of the cantus firmus
    (defparameter *cf-tempo (om::tempo cantus-firmus))
    ; get the first note of the cantus firmus ;; just used for the moment
    (defparameter *tone-pitch-cf (first *cf))
    ; get the borrowed scale of the cantus firmus, i.e. some notes borrowed from the natural scale of the tone (useful for modes)
    (setq mode-param (convert-to-mode-integer borrow-mode *tone-pitch-cf))
    (if mode-param
        (defparameter *borrowed-scale (build-scaleset (get-scale "borrowed") mode-param))
        (defparameter *borrowed-scale (list))
    )
    ; get notes that are not in the natural scale of the tone
    (defparameter *off-scale (set-difference *chromatic-scale *scale))
    ; set the pitch range of the counterpoint
    (defparameter *cp-range (range (+ *tone-pitch-cf RANGE_UB) :min (+ *tone-pitch-cf RANGE_LB))) ; arbitrary range
    ; set counterpoint pitch domain
    (defparameter *cp-domain (intersection *cp-range *scale))
    ; penultimate (first *cp) note domain
    (defparameter *chromatic-cp-domain (intersection *cp-range *chromatic-scale))
    ; set counterpoint extended pitch domain
    (defparameter *extended-cp-domain (intersection *cp-range (union *scale *borrowed-scale)))
    ; set the domain of the only barrowed notes
    (defparameter *off-domain (intersection *cp-range *off-scale))
    ; length of the cantus firmus
    (defparameter *cf-len (length *cf))
    ; *cf-last-index is the number of melodic intervals in the cantus firmus
    (defparameter *cf-last-index (- *cf-len 1))
    ; *cf-penult-index is the number of larger (n -> n+2) melodic intervals in the cantus firmus
    (defparameter *cf-penult-index (- *cf-len 2))
    ; COST_UB is the upper bound of the cost function
    (defparameter COST_UB (* *cf-len 20))
)
```

E.4 fuxcp-main.lisp

```
(in-package :fuxcp)

; Author: Thibault Wafflard
; Date: June 3, 2023
; This file contains the functions that:
;   - dispatch to the right species functions
;   - set the global variables of the CSP
;   - manage the search for solutions

(print "Loading fux-cp...")

; get the value at key @k in the hash table @h as a list
(defun geth-dom (h k)
    (list (gethash k h))
)

; get the value at key @k in the parameters table as a list
(defun getparam-val (k)
    (geth-dom *params* k)
```

```
)

; get the value at key @k in the parameters table as a domain
(defun getparam-dom (k)
    (list 0 (getparam k))
)

; get the value at key @k in the parameters table
(defun getparam (k)
    (gethash k *params*)
)

; get if borrow-mode param is allowed
(defun is-borrow-allowed ()
    (not (equal (getparam 'borrow-mode) "None"))
)

; re/define all the variables the CSP needs
(defun set-space-variables ()
    ; THE CSP SPACE
    (defparameter *sp* (gil::new-space))

    ;; CONSTANTS
    ; Number of costs added
    (defparameter *n-cost-added 0)
    ; Motion types
    (defparameter DIRECT 2)
    (defparameter OBLIQUE 1)
    (defparameter CONTRARY 0)

    ;; COSTS
    ;; Melodic costs
    (defparameter *m-step-cost* (gil::add-int-var-dom *sp* (getparam-val 'm-step-cost)))
    (defparameter *m-third-cost* (gil::add-int-var-dom *sp* (getparam-val 'm-third-cost)))
    (defparameter *m-fourth-cost* (gil::add-int-var-dom *sp* (getparam-val 'm-fourth-cost)))
    (defparameter *m-tritone-cost* (gil::add-int-var-dom *sp* (getparam-val 'm-tritone-cost)))
    (defparameter *m-fifth-cost* (gil::add-int-var-dom *sp* (getparam-val 'm-fifth-cost)))
    (defparameter *m-sixth-cost* (gil::add-int-var-dom *sp* (getparam-val 'm-sixth-cost)))
    (defparameter *m-seventh-cost* (gil::add-int-var-dom *sp* (getparam-val 'm-seventh-cost)))
    (defparameter *m-octave-cost* (gil::add-int-var-dom *sp* (getparam-val 'm-octave-cost)))
    ;; General costs
    (defparameter *borrow-cost* (gil::add-int-var-dom *sp* (getparam-val 'borrow-cost)))
    (defparameter *h-fifth-cost* (gil::add-int-var-dom *sp* (getparam-val 'h-fifth-cost)))
    (defparameter *h-octave-cost* (gil::add-int-var-dom *sp* (getparam-val 'h-octave-cost)))
    (defparameter *con-motion-cost* (gil::add-int-var-dom *sp* (getparam-val 'con-motion-cost)))
    (defparameter *obl-motion-cost* (gil::add-int-var-dom *sp* (getparam-val 'obl-motion-cost)))
    (defparameter *dir-motion-cost* (gil::add-int-var-dom *sp* (getparam-val 'dir-motion-cost)))
    ;; Species specific costs
    (defparameter *penult-sixth-cost* (gil::add-int-var-dom *sp* (getparam-val '
        penult-sixth-cost)))
    (defparameter *non-cambiata-cost* (gil::add-int-var-dom *sp* (getparam-val '
        non-cambiata-cost)))
    (defparameter *two-beats-apart-cost* (gil::add-int-var-dom *sp* (getparam-val '
        two-beats-apart-cost)))
    (defparameter *two-bars-apart-cost* (gil::add-int-var-dom *sp* (getparam-val '
        two-bars-apart-cost)))
    (defparameter *no-syncopation-cost* (gil::add-int-var-dom *sp* (getparam-val '
        no-syncopation-cost)))

    ;; Params domains
    (defparameter *motions-domain*
        (remove-duplicates (mapcar (lambda (x) (getparam x))
            (list 'con-motion-cost 'obl-motion-cost 'dir-motion-cost)
        ))
    )

    ; Integer constants (to represent costs or intervals)
```

```lisp
    ; 0 in IntVar
    (defparameter ZERO (gil::add-int-var-dom *sp* (list 0)))
    ; 1 in IntVar
    (defparameter ONE (gil::add-int-var-dom *sp* (list 1)))
    ; 3 in IntVar (minor third)
    (defparameter THREE (gil::add-int-var-dom *sp* (list 3)))
    ; 9 in IntVar (major sixth)
    (defparameter NINE (gil::add-int-var-dom *sp* (list 9)))

    ; Boolean constants
    ; 0 in BoolVar
    (defparameter FALSE (gil::add-bool-var *sp* 0 0))
    ; 1 in BoolVar
    (defparameter TRUE (gil::add-bool-var *sp* 1 1))

    ; Intervals constants
    ; perfect consonances intervals
    (defparameter P_CONS (list 0 7))
    ; imperfect consonances intervals
    (defparameter IMP_CONS (list 3 4 8 9))
    ; all consonances intervals
    (defparameter ALL_CONS (union P_CONS IMP_CONS))
    ; dissonances intervals
    (defparameter DIS (list 1 2 5 6 10 11))
    ; penultimate intervals, i.e. minor third and major sixth
    (defparameter PENULT_CONS (list 3 9))
    ; penultimate thesis intervals, i.e. perfect fifth and sixth
    (defparameter PENULT_THESIS (list 7 8 9))
    ; penultimate 1st quarter note intervals, i.e. minor third, major sixth and octave/unisson
    (defparameter PENULT_1Q (list 0 3 8))
    ; penultimate syncope intervals, i.e. seconds and sevenths
    (defparameter PENULT_SYNCOPE (list 1 2 10 11))

    ; P_CONS in IntVar
    (defparameter P_CONS_VAR (gil::add-int-var-const-array *sp* P_CONS))
    ; IMP_CONS in IntVar
    (defparameter IMP_CONS_VAR (gil::add-int-var-const-array *sp* IMP_CONS))
    ; ALL_CONS in IntVar
    (defparameter ALL_CONS_VAR (gil::add-int-var-const-array *sp* ALL_CONS))
    ; PENULT_CONS in IntVar
    (defparameter PENULT_CONS_VAR (gil::add-int-var-const-array *sp* PENULT_CONS))
    ; PENULT_THESIS in IntVar
    (defparameter PENULT_THESIS_VAR (gil::add-int-var-const-array *sp* PENULT_THESIS))
    ; PENULT_1Q in IntVar
    (defparameter PENULT_1Q_VAR (gil::add-int-var-const-array *sp* PENULT_1Q))
    ; PENULT_SYNCOPE in IntVar
    (defparameter PENULT_SYNCOPE_VAR (gil::add-int-var-const-array *sp* PENULT_SYNCOPE))

    ; *cf-brut-intervals is the list of brut melodic intervals in the cantus firmus
    (setq *cf-brut-m-intervals (gil::add-int-var-array *sp* *cf-last-index -127 127))

    ;; FIRST SPECIES COUNTERPOINT GLOBAL VARIABLES
    (defparameter *cp (list nil nil nil nil))
    (defparameter *h-intervals (list nil nil nil nil))
    (defparameter *m-intervals-brut (list nil nil nil nil))
    (defparameter *m-intervals (list nil nil nil nil))
    (defvar *m2-intervals-brut)
    (defvar *m2-intervals)
    (defvar *cf-brut-m-intervals)
    (defvar *is-p-cons-arr)
    (defparameter *motions (list nil nil nil nil))
    (defparameter *motions-cost (list nil nil nil nil))
    (defvar *is-cf-bass)
    (defparameter *is-cf-bass-arr (list nil nil nil nil))
    (defvar *is-cp-off-key-arr)
    (defvar *N-COST-FACTORS)
    (defvar *cost-factors)
```

```lisp
    (defvar *total-cost)
    (defvar *p-cons-cost)
    (defvar *fifth-cost)
    (defvar *octave-cost)
    (defvar *m-degrees-cost)
    (defvar *m-degrees-type)
    (defvar *off-key-cost)

    ;; SECOND SPECIES COUNTERPOINT GLOBAL VARIABLES
    (defvar *h-intervals-abs)
    (defvar *h-intervals-brut)
    (defparameter *m-succ-intervals (list nil nil nil))
    (defparameter *m-succ-intervals-brut (list nil nil nil))
    (defvar *m2-len)
    (defvar *total-m-len)
    (defvar *m-all-intervals)
    (defvar *m-all-intervals-brut)
    (defvar *real-motions)
    (defvar *real-motions-cost)
    (defvar *is-ta-dim-arr)
    (defvar *is-nbour-arr)
    (defvar *penult-thesis-cost)
    (defvar *total-cp)

    ;; THIRD SPECIES COUNTERPOINT GLOBAL VARIABLES
    (defvar *is-5qn-linked-arr)
    (defvar *total-cp-len)
    (defparameter *is-cons-arr (list nil nil nil nil))
    (defparameter *cons-cost (list nil nil nil nil))
    (defvar *is-not-cambiata-arr)
    (defvar *not-cambiata-cost)
    (defvar *m2-eq-zero-cost)

    ;; FOURTH SPECIES COUNTERPOINT GLOBAL VARIABLES
    (defvar *is-no-syncope-arr)
    (defvar *no-syncope-cost)

    ;; FIFTH SPECIES COUNTERPOINT GLOBAL VARIABLES
    (defvar *species-arr) ; 0: no constraint, 1: first species, 2: second species, 3: third species, 4: fourth species
    (defvar *sp-arr) ; represents *species-arr by position in the measure
    (defparameter *is-nth-species-arr (list nil nil nil nil nil)) ; if *species-arr is n, then *is-nth-species-arr is true
    (defparameter *is-3rd-species-arr (list nil nil nil nil)) ; if *species-arr is 3, then *is-3rd-species-arr is true
    (defparameter *is-4th-species-arr (list nil nil nil nil)) ; if *species-arr is 4, then *is-4th-species-arr is true
    (defvar *is-2nd-or-3rd-species-arr) ; if *species-arr is 2 or 3, then *is-2nd-or-3rd-species-arr is true
    (defvar *m-ta-intervals) ; represents the m-intervals between the thesis note and the arsis note of the same measure
    (defvar *m-ta-intervals-brut) ; same but without the absolute reduction
    (defvar *is-mostly-3rd-arr) ; true if second, third and fourth notes are from the 3rd species
    (defvar *is-constrained-arr) ; represents !(*is-0th-species-arr) i.e. there are species constraints
    (defparameter *is-cst-arr (list nil nil nil nil)) ; represents *is-constrained-arr for all beats of the measure

    ; array representing the brut melodic intervals of the cantus firmus
    (create-cf-brut-m-intervals *cf *cf-brut-m-intervals)
)



;; DISPATCHER FUNCTION
(defun fux-cp (species)
```

```lisp
    "Dispatches the counterpoint generation to the appropriate function according to the species
        ."
    ; re/set global variables
    (set-space-variables)

    (print (list "Choosing species: " species))
    (case species ; [1, 2, 3, 4, 5]
        (1 (progn
            (setq *N-COST-FACTORS 5)
            (fux-cp-1st)
        ))
        (2 (progn
            (setq *N-COST-FACTORS 6)
            (fux-cp-2nd)
        ))
        (3 (progn
            (setq *N-COST-FACTORS 7)
            (fux-cp-3rd)
        ))
        (4 (progn
            (setq *N-COST-FACTORS 6)
            (fux-cp-4th)
        ))
        (5 (progn
            (setq *N-COST-FACTORS 8)
            (fux-cp-5th)
        ))
        (otherwise (error "Species ~A not implemented" species))
    )
)

(defun fux-search-engine (the-cp &optional (species 1))
    (let (se tstop sopts)
        ; TOTAL COST
        (gil::g-sum *sp* *total-cost *cost-factors) ; sum of all the cost factors
        (gil::g-cost *sp* *total-cost) ; set the cost function

        ;; SPECIFY SOLUTION VARIABLES
        (print "Specifying solution variables...")
        (gil::g-specify-sol-variables *sp* the-cp)
        (gil::g-specify-percent-diff *sp* 0)

        ;; BRANCHING
        (print "Branching...")
        (setq var-branch-type gil::INT_VAR_DEGREE_SIZE_MAX)
        (setq val-branch-type gil::INT_VAL_RANGE_MIN)

        ; 5th species specific
        (if (eq species 5) ; otherwise there is no species array
            (gil::g-branch *sp* *species-arr var-branch-type gil::INT_VAL_RND)
        )

        ; 3rd and 5th species specific
        (if (member species (list 3 5))(progn
            (gil::g-branch *sp* *m-degrees-cost var-branch-type val-branch-type)
            (gil::g-branch *sp* *off-key-cost var-branch-type val-branch-type)
        )
        (progn ; else
            (if (eq species 2)(progn
                ; (gil::g-branch *sp* *real-motions-cost var-branch-type val-branch-type)
                ; (gil::g-branch *sp* *m-degrees-cost var-branch-type gil::INT_VAL_SPLIT_MIN)
                ; (gil::g-branch *sp* *off-key-cost var-branch-type val-branch-type)
            ))
        )
        )

        ; 5th species specific
```

```
        (if (and (eq species 5) (>= VOICE_TYPE 0)) ; otherwise there is no species array
        (progn
            (gil::g-branch *sp* *no-syncope-cost var-branch-type val-branch-type)
            (gil::g-branch *sp* *not-cambiata-cost var-branch-type val-branch-type)
        )
        )

        ; branching *total-cost
        (gil::g-branch *sp* *total-cost var-branch-type val-branch-type)
        (if (eq species 2)
            (gil::g-branch *sp* *cost-factors var-branch-type val-branch-type)
        )

        ;; Solution variables branching
        (gil::g-branch *sp* the-cp var-branch-type val-branch-type)

        ; time stop
        (setq tstop (gil::t-stop)); create the time stop object
        (setq timeout 5)
        (gil::time-stop-init tstop (* timeout 1000)); 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)
        (gil::set-time-stop sopts tstop); set the timestop object to stop the search if it takes too long

        ;; SEARCH ENGINE
        (print "Search engine...")
        (setq se (gil::search-engine *sp* (gil::opts sopts) gil::DFS));
        (print se)

        (print "CSP constructed")
        (list se the-cp tstop sopts)
    )
)



; SEARCH-NEXT-SOLUTION
; <l> is a list containing in that order the search engine for the problem, the variables
; this function finds the next solution of the CSP using the search engine given as an argument
(defun search-next-fux-cp (l)
    (print "Searching next solution...")
    (let (
        (se (first l))
        (the-cp (second l))
        (tstop (third l))
        (sopts (fourth l))
        (species (fifth l))
        (check t)
        sol sol-pitches sol-species
        )

        (time (om::while check :do
            ; reset the tstop timer before launching the search
            (gil::time-stop-reset tstop)
            ; try to find a solution
            (time (setq sol (try-find-solution se)))
            (if (null sol)
                ; then check if there are solutions left and if the user wishes to continue searching
                (stopped-or-ended (gil::stopped se) (getparam 'is-stopped))
                ; else we have found a solution so break the loop
                (setf check nil)
            )
```

```
        ))

        ; print the solution from GiL
        (print "Solution: ")
        #| (case species
            (1 (progn
                (print "PRINT 1st species")
                (print (list "(first *m-intervals-brut)" (gil::g-values sol (first *m-intervals-brut))))
                (print (list "*cf-brut-m-intervals     " (gil::g-values sol *cf-brut-m-intervals)))
                (print (list "(first *motions)       " (gil::g-values sol (first *motions))))
                (print (list "(first *h-intervals)     " (gil::g-values sol (first *h-intervals))))
            ))
            (2 (progn
                (print "PRINT 2nd species")
                (print (list "(first *cp)         " (gil::g-values sol (first *cp))))
                (print (list "(third *cp)         " (gil::g-values sol (third *cp))))
                (print (list "(third *h-intervals)" (gil::g-values sol (third *h-intervals))))
                (print (list "*m-all-intervals" (gil::g-values sol *m-all-intervals)))
                (print (list "*real-motions" (gil::g-values sol *real-motions)))
                (print (list "*penult-thesis-cost" (gil::g-values sol *penult-thesis-cost)))
            ))
            (3 (progn
                (print "PRINT 3rd species")
                (print (list "(first *cp) " (gil::g-values sol (first *cp))))
                (print (list "(second *cp)" (gil::g-values sol (second *cp))))
                (print (list "(third *cp) " (gil::g-values sol (third *cp))))
                (print (list "(fourth *cp)" (gil::g-values sol (fourth *cp))))
                (print (list "*extended-cp-domain" *extended-cp-domain))
                (print (list "(first *h-intervals) " (gil::g-values sol (first *h-intervals))))
                (print (list "(second *h-intervals)" (gil::g-values sol (second *h-intervals))))
                (print (list "(third *h-intervals) " (gil::g-values sol (third *h-intervals))))
                (print (list "(fourth *h-intervals)" (gil::g-values sol (fourth *h-intervals))))
                (print (list "*m-all-intervals" (gil::g-values sol *m-all-intervals)))
                ; (print (list "(fourth *m-intervals-brut)" (gil::g-values sol (fourth *m-intervals-brut))))
                ; (print (list "(first *motions) " (gil::g-values sol (first *motions))))
                (print (list "(fourth *motions)" (gil::g-values sol (fourth *motions))))
                (print (list "*not-cambiata-cost " (gil::g-values sol *not-cambiata-cost)))
                (print (list "*m2-eq-zero-cost   " (gil::g-values sol *m2-eq-zero-cost)))
                ; (print (list "(first *cons-cost)  " (gil::g-values sol (first *cons-cost))))
                ; (print (list "(second *cons-cost) " (gil::g-values sol (second *cons-cost))))
                ; (print (list "(third *cons-cost)  " (gil::g-values sol (third *cons-cost))))
                ; (print (list "(fourth *cons-cost) " (gil::g-values sol (fourth *cons-cost))))
            ))
            (4 (progn
                (print "PRINT 4th species")
                (print (list "(first *cp)         " (gil::g-values sol (first *cp))))
                (print (list "(third *cp)         " (gil::g-values sol (third *cp))))
                (print (list "(first *h-intervals)" (gil::g-values sol (first *h-intervals))))
                (print (list "(third *h-intervals)" (gil::g-values sol (third *h-intervals))))
                (print (list "*m-all-intervals         " (gil::g-values sol *m-all-intervals)))
                (print (list "(third *m-intervals)     " (gil::g-values sol (third *m-intervals))))
                (print (list "(first *m-succ-intervals) " (gil::g-values sol (first *m-succ-intervals))))
                (print (list "*no-syncope-cost" (gil::g-values sol *no-syncope-cost)))
            ))
            (5 (progn
                (print "PRINT 5th species")
                (print (list "(first *cp) " (gil::g-values sol (first *cp))))
                (print (list "(second *cp)" (gil::g-values sol (second *cp))))
                (print (list "(third *cp) " (gil::g-values sol (third *cp))))
                (print (list "(fourth *cp)" (gil::g-values sol (fourth *cp))))
                (print (list "(first *h-intervals) " (gil::g-values sol (first *h-intervals))))
```

```
                (print (list "(second *h-intervals)" (gil::g-values sol (second *h-intervals))))
                (print (list "(third *h-intervals) " (gil::g-values sol (third *h-intervals))))
                (print (list "(fourth *h-intervals)" (gil::g-values sol (fourth *h-intervals))))
                (print (list "*m-all-intervals" (gil::g-values sol *m-all-intervals)))
                ; (print (list "(fourth *m-intervals-brut)" (gil::g-values sol (fourth *m-intervals-brut))))
                ; (print (list "(first *motions) " (gil::g-values sol (first *motions))))
                (print (list "(fourth *motions)" (gil::g-values sol (fourth *motions))))
                (print (list "*not-cambiata-cost " (gil::g-values sol *not-cambiata-cost)))
                (print (list "*m2-eq-zero-cost   " (gil::g-values sol *m2-eq-zero-cost)))
                ; (print (list "(first *cons-cost)  " (gil::g-values sol (first *cons-cost))))
                (print (list "(second *cons-cost) " (gil::g-values sol (second *cons-cost))))
                (print (list "(third *cons-cost)  " (gil::g-values sol (third *cons-cost))))
                (print (list "(fourth *cons-cost) " (gil::g-values sol (fourth *cons-cost))))
                (print (list "*species-arr" sol-species))
                (print (list "*sp-arr1" (gil::g-values sol (first *sp-arr))))
                (print (list "*sp-arr2" (gil::g-values sol (second *sp-arr))))
                (print (list "*sp-arr3" (gil::g-values sol (third *sp-arr))))
                (print (list "*sp-arr4" (gil::g-values sol (fourth *sp-arr))))
            ))
        )
        (print (list "*m-degrees-cost    " (gil::g-values sol *m-degrees-cost)))
        (print (list "*m-degrees-type    " (gil::g-values sol *m-degrees-type)))
        (print (list "*off-key-cost     " (gil::g-values sol *off-key-cost)))
        (print (list "*fifth-cost  " (gil::g-values sol *fifth-cost)))
        (print (list "*octave-cost " (gil::g-values sol *octave-cost)))
        (print (list "*cost-factors" (gil::g-values sol *cost-factors)))
        (print (list "### COST ### " (gil::g-values sol *total-cost)))
        (print (list "scale         " *scale))
        (print (list "borrowed-scale" *borrowed-scale))
        (print (list "off-scale     " (reverse *off-scale))) |#
        (setq sol-pitches (gil::g-values sol the-cp)) ; store the values of the solution
        (print sol-pitches)
        (case species
            (4 (progn
                (setq rythmic+pitches (get-basic-rythmic 4 *cf-len sol-pitches)) ; get the rythmic correpsonding to the species
                (setq rythmic-om (first rythmic+pitches))
                (setq pitches-om (second rythmic+pitches))
            ))
            (5 (progn
                (setq sol-species (gil::g-values sol *species-arr)) ; store the values of the solution
                (setq rythmic+pitches (parse-species-to-om-rythmic sol-species sol-pitches))
                (setq rythmic-om (first rythmic+pitches))
                ; (print (list "rythmic-om" rythmic-om))
                (setq pitches-om (second rythmic+pitches))
                ; (print (list "pitches-om" pitches-om))
                (setq check (checksum-sol pitches-om rythmic-om))
                ; (print (list "check" check))
                (if (not (null *prev-sol-check))
                    ; then compare the pitches of the previous solution with the current one
                    ; if they are the same launch a new search
                    (if (member check *prev-sol-check)
                        (progn
                            (search-next-fux-cp l)
                        )
                        (progn
                            (print *prev-sol-check)
                            (setq *prev-sol-check (append *prev-sol-check (list check)))
                        )
                    )
                    ; else register the pitches of the current solution
                    (progn
                        (setq *prev-sol-check (list check))
                    )
                )
```

```
            ))
            (otherwise (progn
                (setq rythmic-om (get-basic-rythmic species *cf-len)) ; get the rythmic
                    correpsonding to the species
                (setq pitches-om sol-pitches)
            ))
        )
        (make-instance 'voice :chords (to-midicent pitches-om) :tree (om::mktree rythmic-om '(4
            4)) :tempo *cf-tempo)
    )
)

; try to find a solution, catch errors from GiL and Gecode and restart the search
(defun try-find-solution (se)
    (handler-case
        (gil::search-next se) ; search the next solution, sol is the space of the solution
        (error (c)
            (print "gil::ERROR")
            (try-find-solution se)
        )
    )
)

; 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)
    (print (list "stopped-se" stopped-se "stop-user" stop-user))
    (if (= stopped-se 0); if the search has not been stopped by the TimeStop object, there is no
         more solutions
        (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.")
    )
)
```

E.5 1sp-ctp.lisp

```
(in-package :fuxcp)

; Author: Thibault Wafflard
; Date: June 3, 2023
; This file contains the function that adds all the necessary constraints to the first species.

;;==========================#
;; FIRST SPECIES            #
;;==========================#
(defun fux-cp-1st (&optional (species 1))
    "Create the CSP for the first species of Fux's counterpoint."

    ;============================================ CREATING GIL ARRAYS
        =============================
    ;; initialize the variables
    (print "Initializing variables...")

    ; add the counterpoint array to the space with the domain *cp-domain
    (setf (first *cp) (gil::add-int-var-array-dom *sp* *cf-len *extended-cp-domain))

    (if (and (eq species 1) (is-borrow-allowed))
        ; then add to the penultimate note more possibilities
        (setf (nth *cf-penult-index (first *cp)) (gil::add-int-var-dom *sp* *chromatic-cp-domain
            ))
    )
    ; creating harmonic intervals array
```

```lisp
    (print "Creating harmonic intervals array...")

    ; array of IntVar representing the absolute intervals % 12 between the cantus firmus and the
         counterpoint
    (setf (first *h-intervals) (gil::add-int-var-array *sp* *cf-len 0 11))
    (create-h-intervals (first *cp) *cf (first *h-intervals))

    ; creating melodic intervals array
    (print "Creating melodic intervals array...")
    ; array of IntVar representing the absolute intervals between two notes in a row of the
        counterpoint
    (setf (first *m-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 12))
    (setf (first *m-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -12 12))
    (create-m-intervals-self (first *cp) (first *m-intervals) (first *m-intervals-brut))

    (if (eq species 1) ; only for the first species
        ; then
        (progn
            ; creating melodic intervals array between the note n and n+2
            (setq *m2-intervals (gil::add-int-var-array *sp* *cf-penult-index 0 12))
            (setq *m2-intervals-brut (gil::add-int-var-array *sp* *cf-penult-index -12 12))
            (create-m2-intervals (first *cp) *m2-intervals *m2-intervals-brut)

            ; creating boolean is counterpoint off key array
            (print "Creating is counterpoint off key array...")
            (setq *is-cp-off-key-arr (gil::add-bool-var-array *sp* *cf-len 0 1))
            (create-is-member-arr (first *cp) *is-cp-off-key-arr *off-domain)
        )
    )

    ; creating perfect consonances boolean array
    (print "Creating perfect consonances boolean array...")
    ; array of BoolVar representing if the interval between the cantus firmus and the
        counterpoint is a perfect consonance
    (setq *is-p-cons-arr (gil::add-bool-var-array *sp* *cf-len 0 1))
    (create-is-p-cons-arr (first *h-intervals) *is-p-cons-arr)

    ; creating order/role of pitch array (if cantus firmus is higher or lower than counterpoint)
    ; 0 for being the bass, 1 for being above
    (print "Creating order of pitch array...")
    (setf (first *is-cf-bass-arr) (gil::add-bool-var-array *sp* *cf-len 0 1))
    (create-is-cf-bass-arr (first *cp) *cf (first *is-cf-bass-arr))

    ; creating motion array
    (print "Creating motion array...")
    (setf (first *motions) (gil::add-int-var-array *sp* *cf-last-index 0 2)) ; 0 = contrary, 1 =
         oblique, 2 = direct/parallel
    (setf (first *motions-cost) (gil::add-int-var-array-dom *sp* *cf-last-index *motions-domain
        *))
    (create-motions (first *m-intervals-brut) *cf-brut-m-intervals (first *motions) (first *
        motions-cost))


    ;============================================ HARMONIC CONSTRAINTS
        ============================
    (print "Posting constraints...")

    ; for all intervals between the cantus firmus and the counterpoint, the interval must be a
        consonance
    (print "Harmonic consonances...")
    (case species
        (1 (add-h-cons-cst *cf-len *cf-penult-index (first *h-intervals)))
        (2 (add-h-cons-cst *cf-len *cf-penult-index (first *h-intervals) PENULT_THESIS_VAR))
        (3 (add-h-cons-cst *cf-len *cf-penult-index (first *h-intervals) PENULT_1Q_VAR))
        (otherwise (error "Species not supported"))
    )

```

```
    ; no unisson between the cantus firmus and the counterpoint unless it is the first note or
        the last note
    (print "No unisson...")
    (add-no-unisson-cst (first *cp) *cf)

    (if (/= species 3)
        ; then
        (progn
        ; must start with a perfect consonance
        (print "Perfect consonance at the beginning...")
        (add-p-cons-start-cst (first *h-intervals))

        ; must end with a perfect consonance
        (print "Perfect consonance at the end...")
        (add-p-cons-end-cst (first *h-intervals))
        )
    )

    ; if penultimate measure, a major sixth or a minor third must be used
    ; depending if the cantus firmus is at the bass or on the top part
    (print "Penultimate measure...")
    (if (eq species 1)
        ; then
        (add-penult-cons-cst (penult (first *is-cf-bass-arr)) (penult (first *h-intervals)))
    )


    ;============================================ MELODIC CONSTRAINTS
        =============================

    ; NOTE: with the degree iii in penultimate *cf measure -> no solution bc there is a *tritone
         between I#(minor third) and V.
    (print "Melodic constraints...")
    (if (eq species 1)
        ; then
        (progn
            ; no more than minor sixth melodic interval
            (print "No more than minor sixth...")
            (add-no-m-jump-cst (first *m-intervals))

            ; no *chromatic motion between three consecutive notes
            (print "No chromatic motion...")
            (add-no-chromatic-m-cst (first *m-intervals-brut) *m2-intervals-brut)

            ;==================================== MOTION CONSTRAINTS
                ============================
            (print "Motion constraints...")

            ; no direct motion to reach a perfect consonance
            (print "No direct motion to reach a perfect consonance...")
            (add-no-direct-move-to-p-cons-cst (first *motions) *is-p-cons-arr)

            ; no battuta kind of motion
            ; i.e. contrary motion to an *octave, lower voice up, higher voice down,
                counterpoint melodic interval < -4
            (print "No battuta kind of motion...")
            (add-no-battuta-cst (first *motions) (first *h-intervals) (first *m-intervals-brut)
                (first *is-cf-bass-arr))
        )
    )


    ;============================================ COST FACTORS
        ====================================
    (print "Cost function...")

    (if (eq species 1)
```

```
        ; then
        (progn
            (setq *m-all-intervals (first *m-intervals))
            (set-cost-factors)
            ; 1, 2) imperfect consonances are preferred to perfect consonances
            (print "Imperfect consonances are preferred to perfect consonances...")
            (add-p-cons-cost-cst)
            ; 3, 4) add off-key cost, m-degrees cost and tritons cost
            (set-general-costs-cst *cf-len)

            ; 5) motion costs
            (add-cost-to-factors (first *motions-cost))
        )
    )


    ; RETURN
    (if (eq species 1)
        ; then create the search engine
        (append (fux-search-engine (first *cp)) (list species))
        ; else
        nil
    )
)
```

E.6 2sp-ctp.lisp

```
(in-package :fuxcp)

; Author: Thibault Wafflard
; Date: June 3, 2023
; This file contains the function that adds all the necessary constraints to the second species.

;;==========================#
;; SECOND SPECIES           #
;;==========================#
;; Note: fux-cp-2nd execute the first species algorithm without some constraints.
;; In this function, all the variable names without the arsis-suffix refers to thesis notes AKA the first species notes.
;; All the variable names with the arsis-suffix refers to arsis notes AKA notes on the upbeat.
(defun fux-cp-2nd (&optional (species 2))
    "Create the CSP for the 2nd species of Fux's counterpoint, with the cantus firmus as input"

    ;; ADD FIRST SPECIES CONSTRAINTS
    (fux-cp-1st 2)

    (print "########## SECOND SPECIES ##########")

    ;======================================== CREATION OF GIL ARRAYS ==========================
    (print "Initializing variables...")
    ; add the arsis counterpoint array (of [*cf-len - 1] length) to the space with the domain *cp-domain
    (setf (third *cp) (gil::add-int-var-array-dom *sp* *cf-last-index *extended-cp-domain))
    ; add to the penultimate note more possibilities
    (if (is-borrow-allowed)
        (setf (nth *cf-penult-index (third *cp)) (gil::add-int-var-dom *sp* *chromatic-cp-domain))
    )

    ; merging cp and cp-arsis into one array
    (setq *total-cp-len (+ *cf-len *cf-last-index))
    (setq *total-cp (gil::add-int-var-array *sp* *total-cp-len 0 127)) ; array of IntVar representing thesis and arsis notes combined
    (merge-cp (list (first *cp) (third *cp)) *total-cp) ; merge the two counterpoint arrays into one
```

```

    ; creating harmonic intervals array
    (print "Creating harmonic intervals array...")
    ; array of IntVar representing the absolute intervals % 12 between the cantus firmus and the counterpoint (arsis notes)
    (setf (third *h-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 11))
    (create-h-intervals (third *cp) (butlast *cf) (third *h-intervals))
    ; array of IntVar representing the absolute intervals (not % 12) and brut (just p - q)
    ; between the cantus firmus and the counterpoint (thesis notes)
    (setq *h-intervals-abs (gil::add-int-var-array *sp* *cf-len 0 127))
    (setq *h-intervals-brut (gil::add-int-var-array *sp* *cf-len -127 127))
    (create-intervals *cf (first *cp) *h-intervals-abs *h-intervals-brut)


    ; creating melodic intervals array
    (print "Creating melodic intervals array...")
    ; array of IntVar representing the melodic intervals between arsis note and next thesis note of the counterpoint
    (setf (third *m-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 12))
    (setf (third *m-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -12 12)) ; same without absolute reduction
    (create-m-intervals-next-meas (third *cp) (first *cp) (third *m-intervals) (third *m-intervals-brut))
    ; array of IntVar representing the melodic intervals between a thesis and an arsis note of the same measure the counterpoint
    (setf (first *m-succ-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 12))
    (setf (first *m-succ-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -12 12))
    (create-m-intervals-in-meas (first *cp) (third *cp) (first *m-succ-intervals) (first *m-succ-intervals-brut))


    ; creating melodic intervals array between the note n and n+2 for the whole counterpoint
    (setq *m2-len (- (* *cf-last-index 2) 1)) ; number of melodic intervals between n and n+2 for thesis and arsis notes combined
    (setq *m2-intervals (gil::add-int-var-array *sp* *m2-len 0 12))
    (setq *m2-intervals-brut (gil::add-int-var-array *sp* *m2-len -12 12))
    (create-m2-intervals *total-cp *m2-intervals *m2-intervals-brut)

    ; creating melodic intervals array between the note n and n+1 for the whole counterpoint
    (setq *total-m-len (* *cf-last-index 2)) ; number of melodic intervals between n and n+1 for thesis and arsis notes combined
    (setq *m-all-intervals (gil::add-int-var-array *sp* *total-m-len 0 12))
    (setq *m-all-intervals-brut (gil::add-int-var-array *sp* *total-m-len -12 12))
    (create-m-intervals-self *total-cp *m-all-intervals *m-all-intervals-brut)

    ; creating motion array
    ; 0 = contrary, 1 = oblique, 2 = direct/parallel
    (print "Creating motion array...")
    (setf (third *motions) (gil::add-int-var-array *sp* *cf-last-index 0 2))
    (setf (third *motions-cost) (gil::add-int-var-array-dom *sp* *cf-last-index *motions-domain*))
    (setq *real-motions (gil::add-int-var-array *sp* *cf-last-index 0 2))
    (setf *real-motions-cost (gil::add-int-var-array-dom *sp* *cf-last-index *motions-domain*))
    (create-motions (third *m-intervals-brut) *cf-brut-m-intervals (third *motions) (third *motions-cost))
    (create-real-motions (first *m-succ-intervals) (first *motions) (third *motions) *real-motions (first *motions-cost) (third *motions-cost) *real-motions-cost)

    ; creating boolean diminution array
    (print "Creating diminution array...")
    ; Note: a diminution is the intermediate note that exists between two notes separated by a jump of a third
    ; i.e. E -> D (dim) -> C
    (setq *is-ta-dim-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (print "DEBUG")
    (print (first *m-succ-intervals))
    (print (first *m-intervals))
```

```
    (print (third *m-intervals))
    (create-is-ta-dim-arr (first *m-succ-intervals) (first *m-intervals) (third *m-intervals) *is-ta-dim-arr)


    ; creating boolean is cantus firmus bass array
    (print "Creating is cantus firmus bass array...")
    ; array of BoolVar representing if the cantus firmus is lower than the arsis counterpoint
    (setf (third *is-cf-bass-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-cf-bass-arr (third *cp) (butlast *cf) (third *is-cf-bass-arr))

    ; creating boolean is cantus firmus neighboring the counterpoint array
    (print "Creating is cantus firmus neighboring array...")
    (setq *is-nbour-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-nbour-arr *h-intervals-abs (first *is-cf-bass-arr) *cf-brut-m-intervals *is-nbour-arr)

    ; creating boolean is counterpoint off key array
    (print "Creating is counterpoint off key array...")
    (setq *is-cp-off-key-arr (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-is-member-arr *total-cp *is-cp-off-key-arr *off-domain)


    ;======================================== HARMONIC CONSTRAINTS =============================
    (print "Posting constraints...")

    (print "Harmonic consonances...")

    ; for all harmonic intervals between the cantus firmus and the arsis notes, the interval must be a consonance
    ; unless the arsis note is a diminution
    (print "No dissonance unless diminution for arsis notes...")
    (add-h-cons-arsis-cst *cf-len *cf-penult-index (third *h-intervals) *is-ta-dim-arr)

    ; Fux does not follow this rule so deactivate ?
    ; no unisson between the cantus firmus and the arsis counterpoint
    ; (print "No unisson at all...")
    ; (add-no-unisson-at-all-cst (third *cp) (butlast *cf))

    ; if penultimate measure, a major sixth or a minor third must be used
    ; depending if the cantus firmus is at the bass or on the top part
    (print "Penultimate measure...")
    ; (gil::g-rel *sp* (fourth (first *h-intervals)) gil::IRT_NQ 7) ; TODO: fix this
    (add-penult-cons-cst (lastone (third *is-cf-bass-arr)) (lastone (third *h-intervals)))


    ;======================================== MELODIC CONSTRAINTS ==============================
    (print "Melodic constraints...")

    ; no more than minor sixth melodic interval between thesis and arsis notes UNLESS:
    ;   - the interval between the cantus firmus and the thesis note <= major third
    ;   - the cantus firmus is getting closer to the thesis note
    (print "No more than minor sixth melodic interval between thesis and arsis notes unless...")
    (add-m-inter-arsis-cst (first *m-succ-intervals) *is-nbour-arr)

    ; Fux does not follow this rule, deactivate ?
    ; (print "No more than minor sixth melodic interval between arsis and thesis notes...")
    ; (add-no-m-jump-cst (third *m-intervals))

    ; no *chromatic motion between three consecutive notes
    (print "No chromatic motion...")
    (add-no-chromatic-m-cst *m-all-intervals-brut *m2-intervals-brut)

    ; no unisson between two consecutive notes
    (print "No unisson between two consecutive notes...")
    (add-no-unisson-at-all-cst *total-cp (rest *total-cp))

```

```
    ;======================================== MOTION CONSTRAINTS =============================
    (print "Motion constraints...")

    ; no direct motion to reach a perfect consonance
    (print "No direct motion to reach a perfect consonance...")
    (add-no-direct-move-to-p-cons-cst *real-motions *is-p-cons-arr)

    ; no battuta kind of motion
    ; i.e. contrary motion to an *octave, lower voice up, higher voice down, counterpoint
        melodic interval < -4
    (print "No battuta kind of motion...")
    (add-no-battuta-cst (third *motions) (first *h-intervals) (third *m-intervals-brut) (third *
        is-cf-bass-arr))



    ;======================================== COST FACTORS ====================================
    (set-cost-factors)
    ; 1, 2) imperfect consonances are preferred to perfect consonances
    (print "Imperfect consonances are preferred to perfect consonances...")
    (add-p-cons-cost-cst)

    ; 3, 4) add off-key cost, m-degrees cost
    (set-general-costs-cst)

    ; 5) contrary motion is preferred
    (add-cost-to-factors *real-motions-cost)

    ; 6) the penultimate thesis note is not a fifth
    (print "Penultimate thesis note is not a fifth...")
    ; *penult-thesis-cost = *cf-len (big cost) if penultimate *h-interval /= 7
    (setq *penult-thesis-cost (gil::add-int-var-dom *sp* (getparam-dom 'penult-sixth-cost)))
    (add-single-cost-cst (penult (first *h-intervals)) gil::IRT_NQ 7 *penult-thesis-cost *
        penult-sixth-cost*)
    (setf (nth *n-cost-added *cost-factors) *penult-thesis-cost)
    (incf *n-cost-added)


    ;======================================== COST FUNCTION ==================================
    (print "Cost function...")

    ; RETURN
    (if (eq species 2)
        ; then create the search engine
        (append (fux-search-engine *total-cp 2) (list species))
        ; else
        nil
    )
)
```

E.7 3sp-ctp.lisp

```
(in-package :fuxcp)

; Author: Thibault Wafflard
; Date: June 3, 2023
; This file contains the function that adds all the necessary constraints to the third species.

;;==========================#
;; THIRD SPECIES            #
;;==========================#
;; Note: fux-cp-3rd execute the first species algorithm without some constraints.
;; In this function, 4 quarter notes by measure are assumed.
(defun fux-cp-3rd (&optional (species 3))
```

```lisp
    "Create the CSP for the 3rd species of Fux's counterpoint, with the cantus firmus as input"
    (print "Creating the CSP for the 3rd species of Fux's counterpoint...")

    ;; ADD FIRST SPECIES CONSTRAINTS
    (fux-cp-1st 3)

    (print "########## THIRD SPECIES ##########")

    ;======================================== CREATION OF GIL ARRAYS =========================
    (print "Initializing variables...")

    (loop for i from 1 to 3 do
        ; add all quarter notes to the space with the domain *cp-domain
        (setf (nth i *cp) (gil::add-int-var-array-dom *sp* *cf-last-index *extended-cp-domain))

        (if (and (eq i 3) (is-borrow-allowed))
            ; then add to the penultimate note more possibilities
            (setf (nth *cf-penult-index (nth i *cp)) (gil::add-int-var-dom *sp* *chromatic-cp-domain))
        )
    )

    (loop for i from 1 to 3 do
        (setq i-1 (- i 1))
        ; creating harmonic intervals array
        ; array of IntVar representing the absolute intervals % 12 between the cantus firmus and the counterpoint
        (setf (nth i *h-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 11))
        (create-h-intervals (nth i *cp) (butlast *cf) (nth i *h-intervals))

        ; array of IntVar representing the absolute intervals between a thesis and an arsis note of the same measure the counterpoint
        (setf (nth i-1 *m-succ-intervals) (gil::add-int-var-array *sp* *cf-last-index 1 12))
        (setf (nth i-1 *m-succ-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -12 12))
        (create-intervals (nth i-1 *cp) (nth i *cp) (nth i-1 *m-succ-intervals) (nth i-1 *m-succ-intervals-brut))
    )

    ; merging cp and cp-arsis into one array
    (print "Mergin cps...")
    (setq *total-cp-len (+ *cf-len (* *cf-last-index 3))) ; total length of the counterpoint array
    (setq *total-cp (gil::add-int-var-array *sp* *total-cp-len 0 127)) ; array of IntVar representing thesis and arsis notes combined
    (merge-cp *cp *total-cp) ; merge the four counterpoint arrays into one

    ; creating melodic intervals array
    (print "Creating melodic intervals array...")
    ; array of IntVar representing the absolute intervals
    ; between the last note of measure m and the first note of measure m+1 of the counterpoint
    (setf (fourth *m-intervals) (gil::add-int-var-array *sp* *cf-last-index 1 12))
    (setf (fourth *m-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -12 12)) ; same without absolute reduction
    (create-m-intervals-next-meas (fourth *cp) (first *cp) (fourth *m-intervals) (fourth *m-intervals-brut))

    ; creating melodic intervals array between the note n and n+2 for the whole counterpoint
    (setq *m2-len (- (* *cf-last-index 4) 1)) ; number of melodic intervals between n and n+2 for the total counterpoint
    (setq *m2-intervals (gil::add-int-var-array *sp* *m2-len 0 12))
    (setq *m2-intervals-brut (gil::add-int-var-array *sp* *m2-len -12 12))
    (create-m2-intervals *total-cp *m2-intervals *m2-intervals-brut)

    ; creating melodic intervals array between the note n and n+1 for the whole counterpoint
    (setq *total-m-len (* *cf-last-index 4)) ; number of melodic intervals between n and n+1 for the total counterpoint
```

```
    (setq *m-all-intervals (gil::add-int-var-array *sp* *total-m-len 0 12))
    (setq *m-all-intervals-brut (gil::add-int-var-array *sp* *total-m-len -12 12))
    (create-m-intervals-self *total-cp *m-all-intervals *m-all-intervals-brut)

    ; creating motion array
    ; 0 = contrary, 1 = oblique, 2 = direct/parallel
    (print "Creating motion array...")
    (setf (fourth *motions) (gil::add-int-var-array *sp* *cf-last-index 0 2))
    (setf (fourth *motions-cost) (gil::add-int-var-array-dom *sp* *cf-last-index *motions-domain
        *))
    (create-motions (fourth *m-intervals-brut) *cf-brut-m-intervals (fourth *motions) (fourth *
        motions-cost))

    ; creating boolean is cantus firmus bass array
    (print "Creating is cantus firmus bass array...")
    ; array of BoolVar representing if the cantus firmus is lower than the arsis counterpoint
    (setf (fourth *is-cf-bass-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-cf-bass-arr (fourth *cp) (butlast *cf) (fourth *is-cf-bass-arr))

    ; creating boolean are five consecutive notes by joint degree array
    (print "Creating are five consecutive notes by joint degree array...")
    ; array of BoolVar representing if the five consecutive notes are by joint degree
    (setq *is-5qn-linked-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-5qn-linked-arr *m-all-intervals *m-all-intervals-brut *is-5qn-linked-arr)

    ; creating boolean diminution array
    (print "Creating diminution array...")
    ; Note: a diminution is the intermediate note that exists between two notes separated by a
        jump of a third
    ; i.e. E -> D (dim) -> C
    (setq *is-ta-dim-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-ta-dim-arr (second *m-succ-intervals) (collect-by-4 *m2-intervals 1 T) (third *
        m-succ-intervals) *is-ta-dim-arr)

    ; creating boolean is consonant array
    (print "Creating is consonant array...")
    (loop for i from 0 to 3 do
        ; array of BoolVar representing if the interval is consonant
        (if (eq i 0)
            (setf (nth i *is-cons-arr) (gil::add-bool-var-array *sp* *cf-len 0 1))
            (setf (nth i *is-cons-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
        )
        (create-is-member-arr (nth i *h-intervals) (nth i *is-cons-arr))
    )

    ; creating boolean is not cambiata array
    (print "Creating is not cambiata array...")
    (setq *is-not-cambiata-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-not-cambiata-arr (second *is-cons-arr) (third *is-cons-arr) (second *
        m-succ-intervals) *is-not-cambiata-arr)

    ; creating boolean is counterpoint off key array
    (print "Creating is counterpoint off key array...")
    (setq *is-cp-off-key-arr (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-is-member-arr *total-cp *is-cp-off-key-arr *off-domain)


    ;======================================== HARMONIC CONSTRAINTS ============================
    (print "Posting constraints...")
    ; must start with a perfect consonance
    (print "Perfect consonance at the beginning...")
    (add-p-cons-start-cst (first *h-intervals))

    ; must end with a perfect consonance
    (print "Perfect consonance at the end...")
    (add-p-cons-end-cst (first *h-intervals))

```

```
    ; if penultimate measure, a major sixth or a minor third must be used
    ; depending if the cantus firmus is at the bass or on the top part
    (print "Penultimate measure...")
    (add-penult-cons-cst (lastone (fourth *is-cf-bass-arr)) (lastone (fourth *h-intervals)))
    ; the third note of the penultimate measure must be below the fourth one.
    (gil::g-rel *sp* (lastone (third *m-succ-intervals-brut)) gil::IRT_GR 1)
    ; the second note and the third note of the penultimate measure must be distant by greater
        than 1 semi-tone from the fourth note
    (gil::g-rel *sp* (penult *m2-intervals) gil::IRT_NQ 1)


    ; five consecutive notes by joint degree implies that the first and the third note are
        consonants
    (print "Five consecutive notes by joint degree...")
    (add-linked-5qn-cst (third *is-cons-arr) *is-5qn-linked-arr)

    ; any dissonant note implies that it is surrounded by consonant notes
    (print "Any dissonant note...")
    (add-h-dis-or-cons-3rd-cst (second *is-cons-arr) (third *is-cons-arr) (fourth *is-cons-arr)
        *is-ta-dim-arr)


    ;======================================== MELODIC CONSTRAINTS ==============================
    (print "Melodic constraints...")

    ; no melodic interval between 9 and 11
    (loop for m in *m-succ-intervals do
        (add-no-m-jump-extend-cst m)
    )
    (add-no-m-jump-extend-cst (fourth *m-intervals))

    ; no *chromatic motion between three consecutive notes
    (print "No chromatic motion...")
    (add-no-chromatic-m-cst *m-all-intervals-brut *m2-intervals-brut)

    ; Marcel's rule: contrary melodic step after skip
    (print "Marcel's rule...")
    (add-contrary-step-after-skip-cst *m-all-intervals *m-all-intervals-brut)

    ;======================================== MOTION CONSTRAINTS ============================
    (print "Motion constraints...")

    ; no direct motion to reach a perfect consonance
    (print "No direct motion to reach a perfect consonance...")
    (add-no-direct-move-to-p-cons-cst (fourth *motions) *is-p-cons-arr)

    ; no battuta kind of motion
    ; i.e. contrary motion to an *octave, lower voice up, higher voice down, counterpoint
        melodic interval < -4
    (print "No battuta kind of motion...")
    (add-no-battuta-cst (fourth *motions) (first *h-intervals) (fourth *m-intervals-brut) (
        fourth *is-cf-bass-arr)) ; TODO

    ;======================================== COST FACTORS ===================================
    (set-cost-factors)
    ; 1, 2) imperfect consonances are preferred to perfect consonances
    (print "Imperfect consonances are preferred to perfect consonances...")
    (add-p-cons-cost-cst)

    ; 3, 4) add off-key cost, m-degrees cost and tritons cost
    (set-general-costs-cst)

    ; 5) contrary motion is preferred
    (add-cost-to-factors (fourth *motions-cost))

    ; 6) cambiata notes are preferred (cons - dis - cons > cons - cons - cons)
    (print "Cambiata notes are preferred...")
```

```lisp
    ; IntVar array representing the cost to have cambiata notes
    (setq *not-cambiata-cost (gil::add-int-var-array-dom *sp* *cf-last-index (getparam-dom '
        non-cambiata-cost)))
    (add-cost-bool-cst *is-not-cambiata-arr *not-cambiata-cost *non-cambiata-cost*)
    (add-cost-to-factors *not-cambiata-cost)

    ; 7) intervals between notes n and n+2 are prefered greater than zero
    (print "Intervals between notes n and n+2 are prefered different than zero...")
    ; IntVar array representing the cost to have intervals between notes n and n+2 equal to zero
    (setq *m2-eq-zero-cost (gil::add-int-var-array-dom *sp* *m2-len (getparam-dom '
        two-beats-apart-cost)))
    (add-cost-cst *m2-intervals gil::IRT_EQ 0 *m2-eq-zero-cost *two-beats-apart-cost*)
    (add-cost-to-factors *m2-eq-zero-cost)


    ;======================================== COST FUNCTION ===================================
    (print "Cost function...")


    ; RETURN
    (if (eq species 3)
        ; then create the search engine
        (append (fux-search-engine *total-cp 3) (list species))
        ; else
        nil
    )
)
```

E.8 4sp-ctp.lisp

```lisp
(in-package :fuxcp)

; Author: Thibault Wafflard
; Date: June 3, 2023
; This file contains the function that adds all the necessary constraints to the fourth species.

;;==========================#
;; FOURTH SPECIES           #
;;==========================#
;; Note: fux-cp-4th execute the first species algorithm without some constraints.
;; In this function, the first notes are in Arsis because of the syncopation.
(defun fux-cp-4th (&optional (species 4))
    "Create the CSP for the 2nd species of Fux's counterpoint, with the cantus firmus as input"

    (print "########## FOURTH SPECIES ##########")

    ;======================================== CREATION OF GIL ARRAYS ==========================
    (print "Initializing variables...")
    ; add the arsis counterpoint array (of [*cf-len - 1] length) to the space with the domain *
        cp-domain
    (setf (third *cp) (gil::add-int-var-array-dom *sp* *cf-last-index *extended-cp-domain))
    (setf (first *cp) (gil::add-int-var-array-dom *sp* *cf-last-index *extended-cp-domain))
    ; add to the penultimate note more possibilities
    (if (is-borrow-allowed)
        (progn
        (setf (nth *cf-penult-index (third *cp)) (gil::add-int-var-dom *sp* *chromatic-cp-domain
            ))
        (setf (nth *cf-penult-index (first *cp)) (gil::add-int-var-dom *sp* *chromatic-cp-domain
            ))
        )
    )

    ; merging cp and cp-arsis into one array
    (setq *total-cp-len (* *cf-last-index 2))
```

```
    (setq *total-cp (gil::add-int-var-array *sp* *total-cp-len 0 127)) ; array of IntVar representing thesis and arsis notes combined
    (merge-cp-same-len (list (third *cp) (first *cp)) *total-cp) ; merge the two counterpoint arrays into one

    ; creating harmonic intervals array
    (print "Creating harmonic intervals array...")
    ; array of IntVar representing the absolute intervals % 12 between the cantus firmus and the counterpoint (arsis notes)
    (setf (third *h-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 11))
    (setf (first *h-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 11))
    (create-h-intervals (third *cp) (butlast *cf) (third *h-intervals))
    (create-h-intervals (first *cp) (rest *cf) (first *h-intervals))


    ; creating melodic intervals array
    (print "Creating melodic intervals array...")
    ; array of IntVar representing the melodic intervals between arsis and next thesis note of the counterpoint
    (setf (third *m-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 8))
    (setf (third *m-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -12 12)) ; same without absolute reduction
    (create-intervals (third *cp) (first *cp) (third *m-intervals) (third *m-intervals-brut))
    ; array of IntVar representing the melodic intervals between a thesis and an arsis note of the same measure the counterpoint
    (setf (first *m-succ-intervals) (gil::add-int-var-array *sp* *cf-penult-index 1 12))
    (setf (first *m-succ-intervals-brut) (gil::add-int-var-array *sp* *cf-penult-index -12 12))
    (create-m-intervals-in-meas (first *cp) (rest (third *cp)) (first *m-succ-intervals) (first *m-succ-intervals-brut))


    ; creating melodic intervals array between the note n and n+2 for the whole counterpoint
    (setq *m2-len (- (* *cf-last-index 2) 2)) ; number of melodic intervals between n and n+2 for thesis and arsis notes combined
    (setq *m2-intervals (gil::add-int-var-array *sp* *m2-len 0 12))
    (setq *m2-intervals-brut (gil::add-int-var-array *sp* *m2-len -12 12))
    (create-m2-intervals *total-cp *m2-intervals *m2-intervals-brut)

    ; creating melodic intervals array between the note n and n+1 for the whole counterpoint
    (setq *total-m-len (- (* *cf-last-index 2) 1)) ; number of melodic intervals between n and n+1 for thesis and arsis notes combined
    (setq *m-all-intervals (gil::add-int-var-array *sp* *total-m-len 0 12))
    (setq *m-all-intervals-brut (gil::add-int-var-array *sp* *total-m-len -12 12))
    (create-m-intervals-self *total-cp *m-all-intervals *m-all-intervals-brut)

    ; creating perfect consonances boolean array
    (print "Creating perfect consonances boolean array...")
    ; array of BoolVar representing if the interval between the cantus firmus and the counterpoint is a perfect consonance
    (setq *is-p-cons-arr (gil::add-bool-var-array *sp* *cf-len 0 1))
    (create-is-p-cons-arr (first *h-intervals) *is-p-cons-arr)

    ; creating boolean is cantus firmus bass array
    (print "Creating is cantus firmus bass array...")
    ; array of BoolVar representing if the cantus firmus is lower than the arsis counterpoint
    (setf (third *is-cf-bass-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (setf (first *is-cf-bass-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-cf-bass-arr (third *cp) (butlast *cf) (third *is-cf-bass-arr))
    (create-is-cf-bass-arr (first *cp) (rest *cf) (first *is-cf-bass-arr))

    ; creating boolean is counterpoint off key array
    (print "Creating is counterpoint off key array...")
    (setq *is-cp-off-key-arr (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-is-member-arr *total-cp *is-cp-off-key-arr *off-domain)

    ; creating boolean is consonant array
    (print "Creating is consonant array...")
```

```lisp
    ; array of BoolVar representing if the interval is consonant
    (setf (first *is-cons-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-member-arr (first *h-intervals) (first *is-cons-arr))

    ; creation boolean is no syncope array
    (print "Creating is no syncope array...")
    ; array of BoolVar representing if the thesis note is note related to the previous one
    (setq *is-no-syncope-arr (gil::add-bool-var-array *sp* *cf-penult-index 0 1))
    (create-is-no-syncope-arr (third *m-intervals) *is-no-syncope-arr)


    ;======================================== HARMONIC CONSTRAINTS ============================
    (print "Posting constraints...")

    ; for all harmonic intervals between the cantus firmus and the thesis notes, the interval must be a consonance
    (print "Harmonic consonances...")
    ; here the penultimate thesis note must be a seventh or a second and the arsis note must be a major sixth or a minor third
    (add-penult-dom-cst (penult (first *h-intervals)) PENULT_SYNCOPE_VAR)
    (add-h-cons-cst *cf-len *cf-penult-index (third *h-intervals))
    (add-no-sync-h-cons (first *h-intervals) *is-no-syncope-arr)

    ; must start with a perfect consonance
    (print "Perfect consonance at the beginning...")
    (add-p-cons-start-cst (third *h-intervals))

    ; must end with a perfect consonance
    (print "Perfect consonance at the end...")
    (add-p-cons-end-cst (first *h-intervals))

    ; no seventh dissonance if the cantus firmus is at the top
    (print "No seventh dissonance if the cantus firmus is at the top...")
    (add-no-seventh-cst (first *h-intervals) (first *is-cf-bass-arr))

    ; if penultimate measure, a major sixth or a minor third must be used
    ; depending if the cantus firmus is at the bass or on the top part
    (print "Penultimate measure...")
    (add-penult-cons-cst (lastone (third *is-cf-bass-arr)) (lastone (third *h-intervals)))


    ;======================================== MELODIC CONSTRAINTS =============================
    (print "Melodic constraints...")

    ; melodic intervals cannot be greater than a minor sixth expect the octave
    (print "No more than minor sixth melodic interval between arsis and thesis notes...")
    (add-no-m-jump-extend-cst (first *m-succ-intervals))

    ; no *chromatic motion between three consecutive notes
    (print "No chromatic motion...")
    (add-no-chromatic-m-cst *m-all-intervals-brut *m2-intervals-brut)


    ;======================================== MOTION CONSTRAINTS ============================
    (print "Motion constraints...")

    ; dissonant notes must be followed by the consonant note below
    (print "Dissonant notes must be followed by the consonant note below...")
    (add-h-dis-imp-cons-below-cst (first *m-succ-intervals-brut) (first *is-cons-arr))

    ; no second dissonance if the cantus firmus is at the bass and a octave/unisson precedes it
    (print "No second dissonance if the cantus firmus is at the bass...")
    (add-no-second-cst (third *h-intervals) (first *h-intervals) (first *is-cf-bass-arr))


    ;======================================== COST FACTORS ===================================
    (print "Cost factors...")
```

```
    (set-cost-factors)
    ; 1, 2) imperfect consonances are preferred to perfect consonances
    (add-p-cons-cost-cst t)

    ; 3, 4) add off-key cost, m-degrees cost and tritons cost
    (set-general-costs-cst)

    ; 5) add no syncopation cost
    (print "No syncopation cost...")
    (setq *no-syncope-cost (gil::add-int-var-array-dom *sp* *cf-penult-index (getparam-dom '
        no-syncopation-cost)))
    (add-cost-cst (butlast (third *m-intervals)) gil::IRT_NQ 0 *no-syncope-cost *
        no-syncopation-cost*)
    (add-cost-to-factors *no-syncope-cost)

    ; 6) add m2-intervals equal to 0 cost
    (print "Monotonia...")
    (setq *m2-eq-zero-cost (gil::add-int-var-array-dom *sp* (- *cf-len 3) (getparam-dom '
        two-bars-apart-cost)))
    (add-cost-multi-cst (third *cp) gil::IRT_EQ (cddr (third *cp)) *m2-eq-zero-cost *
        two-bars-apart-cost*)
    (add-cost-to-factors *m2-eq-zero-cost)

    ;======================================== COST FUNCTION ===================================
    (print "Cost function...")

    ; RETURN
    (if (eq species 4)
        ; then create the search engine
        (append (fux-search-engine *total-cp 4) (list species))
        ; else
        nil
    )
)
```

E.9 5sp-ctp.lisp

```
(in-package :fuxcp)

; Author: Thibault Wafflard
; Date: June 3, 2023
; This file contains the function that adds all the necessary constraints to the fifth species.

;;==========================#
;; FIFTH SPECIES            #
;;==========================#
;; Note: fux-cp-5th execute the first species algorithm without some constraints.
;; In this function, 4 notes by measure are assumed.
(defun fux-cp-5th (&optional (species 5))
    "Create the CSP for the 3rd species of Fux's counterpoint, with the cantus firmus as input"
    (print "Creating the CSP for the 3rd species of Fux's counterpoint...")

    ;; CLEANING PREVIOUS SOLUTIONS
    (setq *prev-sol-check nil)
    (setq rythmic+pitches nil)
    (setq rythmic-om nil)
    (setq pitches-om nil)

    (print "########## FIFTH SPECIES ##########")

    ;======================================== CREATION OF BOOLEAN SPECIES ARRAYS ==============
    (print "Creation of boolean species arrays...")
    ; total length of the counterpoint array
    (setq *total-cp-len (+ *cf-len (* *cf-last-index 3)))
```

```
    ; array representing the species type [0: no constraint, 1: 1st species, 2: 2nd species, 3: 3rd species, 4: 4th species]
    (setq *species-arr (gil::add-int-var-array *sp* *total-cp-len 0 4))
    (create-species-arr *species-arr)
    ; arrays representing if a note is constraint by a species
    (setf (nth 0 *is-nth-species-arr) (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-simple-boolean-arr *species-arr gil::IRT_EQ 0 (nth 0 *is-nth-species-arr))
    (setf (nth 1 *is-nth-species-arr) (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-simple-boolean-arr *species-arr gil::IRT_EQ 1 (nth 1 *is-nth-species-arr))
    (setf (nth 2 *is-nth-species-arr) (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-simple-boolean-arr *species-arr gil::IRT_EQ 2 (nth 2 *is-nth-species-arr))
    (setf (nth 3 *is-nth-species-arr) (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-simple-boolean-arr *species-arr gil::IRT_EQ 3 (nth 3 *is-nth-species-arr))
    (setf (nth 4 *is-nth-species-arr) (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-simple-boolean-arr *species-arr gil::IRT_EQ 4 (nth 4 *is-nth-species-arr))

    ; creating boolean is constrained array
    (print "Creating is constrained array...")
    ; array of BoolVar representing if the interval is constrained
    (setq *is-constrained-arr (collect-not-array (nth 0 *is-nth-species-arr)))


    ;======================================== CREATION OF GIL ARRAYS ==========================
    (print "Initializing variables...")

    (loop for i from 0 to 3 do
        (if (eq i 0)
            (progn
                ; add all quarter notes to the space with the domain *cp-domain
                (setf (nth i *cp) (gil::add-int-var-array-dom *sp* *cf-len *extended-cp-domain))
                ; then add to the penultimate note more possibilities
                (if (is-borrow-allowed)
                    (setf (nth *cf-penult-index (nth i *cp)) (gil::add-int-var-dom *sp* *chromatic-cp-domain))
                )
                ; creating harmonic intervals array
                (print "Creating harmonic intervals array...")
                ; array of IntVar representing the absolute intervals % 12 between the cantus firmus and the counterpoint
                (setf (nth i *h-intervals) (gil::add-int-var-array *sp* *cf-len 0 11))
                (create-h-intervals (nth i *cp) *cf (nth i *h-intervals))
            )
            (progn
                ; same as above but 1 note shorter
                (setf (nth i *cp) (gil::add-int-var-array-dom *sp* *cf-last-index *extended-cp-domain))
                (if (is-borrow-allowed)
                    (setf (nth *cf-penult-index (nth i *cp)) (gil::add-int-var-dom *sp* *chromatic-cp-domain))
                )
                (setf (nth i *h-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 11))
                (create-h-intervals (nth i *cp) (butlast *cf) (nth i *h-intervals))
            )
        )
    )

    (loop for i from 0 to 2 do
        (setq i+1 (+ i 1))
        (setf (nth i *m-succ-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -12 12))
        (if (eq i 1)
            ; then melodic interval could be 0 if there was a dissonant syncope before (see that later)
            (setf (nth i *m-succ-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 12))
            ; else no melodic interval of 0
            (setf (nth i *m-succ-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 12))
        )
```

```
        (create-intervals (nth i *cp) (nth i+1 *cp) (nth i *m-succ-intervals) (nth i *
            m-succ-intervals-brut))
    )


    ; merging all cp arrays into one
    (print "Mergin cps...")
    (setq *total-cp (gil::add-int-var-array *sp* *total-cp-len 0 127)) ; array of IntVar
        representing thesis and arsis notes combined
    (merge-cp *cp *total-cp) ; merge the four counterpoint arrays into one

    ; creating melodic intervals array
    (print "Creating melodic intervals array...")
    ; array of IntVar representing the melodic intervals between arsis and next thesis note of
        the counterpoint
    (setf (third *m-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 16))
    (setf (third *m-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -16 16)) ; same
        without absolute reduction
    (create-m-intervals-next-meas (third *cp) (first *cp) (third *m-intervals) (third *
        m-intervals-brut))
    ; array of IntVar representing the absolute intervals
    ; between the last note of measure m and the first note of measure m+1 of the counterpoint
    (setf (fourth *m-intervals) (gil::add-int-var-array *sp* *cf-last-index 0 12)) ; can be 0 if
         this is replace by 2 eight note
    (setf (fourth *m-intervals-brut) (gil::add-int-var-array *sp* *cf-last-index -12 12)) ; same
         without absolute reduction
    (create-m-intervals-next-meas (fourth *cp) (first *cp) (fourth *m-intervals) (fourth *
        m-intervals-brut))

    ; array of IntVar representing the melodic intervals between the thesis note and the arsis
        note of the same measure
    (setq *m-ta-intervals (gil::add-int-var-array *sp* *cf-last-index 0 16))
    (setq *m-ta-intervals-brut (gil::add-int-var-array *sp* *cf-last-index -16 16)) ; same
        without absolute reduction
    (create-intervals (first *cp) (third *cp) *m-ta-intervals *m-ta-intervals-brut)

    ; creating melodic intervals array between the note n and n+2 for the whole counterpoint
    (setq *m2-len (- (* *cf-last-index 4) 1)) ; number of melodic intervals between n and n+2
        for the total counterpoint
    (setq *m2-intervals (gil::add-int-var-array *sp* *m2-len 0 16))
    (setq *m2-intervals-brut (gil::add-int-var-array *sp* *m2-len -16 16))
    (create-m2-intervals *total-cp *m2-intervals *m2-intervals-brut)

    ; creating melodic intervals array between the note n and n+1 for the whole counterpoint
    (setq *total-m-len (* *cf-last-index 4)) ; number of melodic intervals between n and n+1 for
         the total counterpoint
    (setq *m-all-intervals (gil::add-int-var-array *sp* *total-m-len 0 12))
    (setq *m-all-intervals-brut (gil::add-int-var-array *sp* *total-m-len -12 12))
    (create-m-intervals-self *total-cp *m-all-intervals *m-all-intervals-brut *
        is-constrained-arr)

    ; creating motion array
    ; 0 = contrary, 1 = oblique, 2 = direct/parallel
    (print "Creating motion array...")
    (setf (fourth *motions) (gil::add-int-var-array *sp* *cf-last-index 0 2))
    (setf (fourth *motions-cost) (gil::add-int-var-array-dom *sp* *cf-last-index *motions-domain
        *))
    (create-motions (fourth *m-intervals-brut) *cf-brut-m-intervals (fourth *motions) (fourth *
        motions-cost))

    ; creating boolean is cantus firmus bass array
    (print "Creating is cantus firmus bass array...")
    ; array of BoolVar representing if the cantus firmus is lower than the arsis counterpoint
    (setf (first *is-cf-bass-arr) (gil::add-bool-var-array *sp* *cf-len 0 1))
    (create-is-cf-bass-arr (first *cp) (rest *cf) (first *is-cf-bass-arr)) ; 5th
    (setf (third *is-cf-bass-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-cf-bass-arr (third *cp) (butlast *cf) (third *is-cf-bass-arr)) ; 5th
```

```
    (setf (fourth *is-cf-bass-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-cf-bass-arr (fourth *cp) (butlast *cf) (fourth *is-cf-bass-arr))

    ; creating boolean are five consecutive notes by joint degree array
    (print "Creating are five consecutive notes by joint degree array...")
    ; array of BoolVar representing if the five consecutive notes are by joint degree
    (setq *is-5qn-linked-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-5qn-linked-arr *m-all-intervals *m-all-intervals-brut *is-5qn-linked-arr)
    (setq *is-mostly-3rd-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1)) ; 5th
    (create-is-mostly-3rd-arr (nth 3 *is-nth-species-arr) *is-mostly-3rd-arr)

    ; creating boolean is consonant array + species array
    (print "Creating is consonant array and species array...")
    (loop for i from 0 to 3 do
        ; array of BoolVar representing if the interval is consonant
        (if (eq i 0)
            (progn
                (setf (nth i *is-cons-arr) (gil::add-bool-var-array *sp* *cf-len 0 1))
                (setf (nth i *is-3rd-species-arr) (gil::add-bool-var-array *sp* *cf-len 0 1))
                (setf (nth i *is-4th-species-arr) (gil::add-bool-var-array *sp* *cf-len 0 1))
                (setf (nth i *is-cst-arr) (gil::add-bool-var-array *sp* *cf-len 0 1))
            )
            (progn
                (setf (nth i *is-cons-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
                (setf (nth i *is-3rd-species-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
                (setf (nth i *is-4th-species-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
                (setf (nth i *is-cst-arr) (gil::add-bool-var-array *sp* *cf-last-index 0 1))
            )
        )
        (create-is-member-arr (nth i *h-intervals) (nth i *is-cons-arr))
        (create-by-4 (nth 3 *is-nth-species-arr) (nth i *is-3rd-species-arr) i)
        (create-by-4 (nth 4 *is-nth-species-arr) (nth i *is-4th-species-arr) i)
        (create-by-4 *is-constrained-arr (nth i *is-cst-arr) i)
    )

    ; creating boolean diminution array
    (print "Creating diminution array...")
    ; Note: a diminution is the intermediate note that exists between two notes separated by a jump of a third
    ; i.e. E -> D (dim) -> C
    (setq *is-ta-dim-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-ta-dim-arr (second *m-succ-intervals) (collect-by-4 *m2-intervals 1 T) (third *m-succ-intervals) *is-ta-dim-arr)

    ; creating boolean is not cambiata array
    (print "Creating is not cambiata array...")
    (setq *is-not-cambiata-arr (gil::add-bool-var-array *sp* *cf-last-index 0 1))
    (create-is-not-cambiata-arr (second *is-cons-arr) (third *is-cons-arr) (second *m-succ-intervals) *is-not-cambiata-arr)

    ; creating boolean is counterpoint off key array
    (print "Creating is counterpoint off key array...")
    (setq *is-cp-off-key-arr (gil::add-bool-var-array *sp* *total-cp-len 0 1))
    (create-is-member-arr *total-cp *is-cp-off-key-arr *off-domain)

    ; creating perfect consonances boolean array
    (print "Creating perfect consonances boolean array...")
    ; array of BoolVar representing if the interval between the cantus firmus and the counterpoint is a perfect consonance
    (setq *is-p-cons-arr (gil::add-bool-var-array *sp* *cf-len 0 1))
    (create-is-p-cons-arr (first *h-intervals) *is-p-cons-arr)

    ; creation boolean is no syncope array
    (print "Creating is no syncope array...")
    ; array of BoolVar representing if the thesis note is note related to the previous one
```

```
    (setq *is-no-syncope-arr (gil::add-bool-var-array *sp* *cf-penult-index 0 1))
    (create-is-no-syncope-arr (third *m-intervals) *is-no-syncope-arr)


    ;======================================== HARMONIC CONSTRAINTS ============================
    (print "Posting constraints...")

    ; one possible value for non-constrained notes
    (print "One possible value for non-constrained notes...")
    (add-one-possible-value-cst *total-cp (nth 0 *is-nth-species-arr))

    ; perfect consonances should be used at the start and at the end of the piece
    (print "Perfect consonances at the start and at the end...")
    ; if first note is constrained then it must be a perfect consonance
    (add-p-cons-cst-if (first (first *h-intervals)) (first *is-constrained-arr))
    ; if first note is not constrained then the third note must be a perfect consonance
    (add-p-cons-cst-if (first (third *h-intervals)) (first (nth 0 *is-nth-species-arr)))
    ; no matter what species it is, the last harmonic interval must be a perfect consonance
    (add-p-cons-end-cst (first *h-intervals))

    ; if penultimate measure, a major sixth or a minor third must be used
    ; depending if the cantus firmus is at the bass or on the top part
    (print "Penultimate measure...")
    (add-penult-cons-cst (lastone (fourth *is-cf-bass-arr)) (lastone (fourth *h-intervals))
        (penult (nth 3 *is-nth-species-arr))
    ) ; 3rd species
    ; the third note of the penultimate measure must be below the fourth one. (3rd species)
    (gil::g-rel-reify *sp* (lastone (third *m-succ-intervals-brut)) gil::IRT_GR 1
        (penult (nth 3 *is-nth-species-arr)) gil::RM_IMP
    ) ; 3rd species
    ; the second note and the third note of the penultimate measure must be
    ; distant by greater than 1 semi-tone from the fourth note (3rd species)
    (gil::g-rel-reify *sp* (penult *m2-intervals) gil::IRT_NQ 1
        (nth (total-index *cf-penult-index 1) (nth 3 *is-nth-species-arr)) gil::RM_IMP
    ) ; 3rd species

    ; for the 4th species, the thesis note must be a seventh or a second and the arsis note must be a major sixth or a minor third
    ; major sixth or minor third
    (add-penult-cons-cst (lastone (third *is-cf-bass-arr)) (lastone (third *h-intervals))
        (penult (butlast (nth 4 *is-nth-species-arr)))
    ) ; 4th species
    ; seventh or second
    ; (note: a => !b <=> !(a ^ b)), so here we use the negation of the conjunction
    (gil::g-op *sp* (penult (first *is-4th-species-arr)) gil::BOT_AND (penult (first *is-cons-arr)) 0) ; 4th species

    ; every thesis note should be consonant if it does not belong to the fourth species (or not constrained at all)
    (print "Every thesis note should be consonant...")
    (add-h-cons-cst-if (first *is-cons-arr) (collect-by-4 (nth 1 *is-nth-species-arr))) ; 1st species
    (add-h-cons-cst-if (first *is-cons-arr) (collect-by-4 (nth 2 *is-nth-species-arr))) ; 2nd species
    (add-h-cons-cst-if (first *is-cons-arr) (first *is-3rd-species-arr)) ; 3rd species
    (add-h-cons-cst-if (third *is-cons-arr) (third *is-4th-species-arr)) ; 4th species
    (add-h-cons-cst-if (first *is-cons-arr) (collect-bot-array (rest (first *is-4th-species-arr)) *is-no-syncope-arr)) ; 4th species

    ; five consecutive notes by joint degree implies that the first and the third note are consonants
    (print "Five consecutive notes by joint degree...") ; 3rd species
    (add-linked-5qn-cst (third *is-cons-arr) (collect-bot-array *is-5qn-linked-arr *is-mostly-3rd-arr))

    ; any dissonant note implies that it is surrounded by consonant notes
    (print "Any dissonant note...") ; 3rd species
```

```
    (add-h-dis-or-cons-3rd-cst
        (second *is-cons-arr)
        (collect-t-or-f-array (third *is-cons-arr) (third *is-3rd-species-arr))
        (fourth *is-cons-arr)
        *is-ta-dim-arr
    )

    ; no seventh dissonance if the cantus firmus is at the top
    (print "No seventh dissonance if the cantus firmus is at the top...")
    (add-no-seventh-cst (first *h-intervals) (first *is-cf-bass-arr) (first *is-4th-species-arr)
        ) ; 4th species


    ;======================================== MELODIC CONSTRAINTS =============================
    (print "Melodic constraints...")

    ; no melodic interval between 9 and 11
    (add-no-m-jump-extend-cst *m-all-intervals (collect-bot-array (butlast *is-constrained-arr)
        (rest *is-constrained-arr)))

    ; no unisson between two consecutive notes
    ; exept for in the second part or the fourth part of the measure
    (print "No unisson between two consecutive notes...")
    ; if 1st note and 2nd note exists (it means it belongs to a species)
    (add-no-unisson-at-all-cst
        (first *cp) (second *cp)
        (collect-bot-array (first *is-cst-arr) (second *is-cst-arr))
    ) ; 5th
    (add-no-unisson-at-all-cst
        (third *cp) (fourth *cp)
        (collect-bot-array (third *is-cst-arr) (fourth *is-cst-arr))
    ) ; 5th

    ; melodic intervals between thesis and arsis note from the same measure
    ; can't be greater than a minor sixth expect the octave (just for the fourth species)
    (print "No more than minor sixth melodic interval between arsis and thesis notes...")
    ; only applied if the the second note is not constrained
    (add-no-m-jump-extend-cst *m-ta-intervals (collect-by-4 (nth 0 *is-nth-species-arr) 1)) ; 4
        th species

    ; no same syncopation if 4th species
    (add-no-same-syncopation-cst (first *cp) (third *cp) (collect-bot-array (first *
        is-4th-species-arr) (third *is-cst-arr)))


    ;======================================== MOTION CONSTRAINTS ============================
    (print "Motion constraints...")

    ; no direct motion to reach a perfect consonance
    (print "No direct motion to reach a perfect consonance...")
    (add-no-direct-move-to-p-cons-cst (fourth *motions) (collect-bot-array *is-p-cons-arr (
        fourth *is-3rd-species-arr)) nil) ; 3rd species

    ; no battuta kind of motion
    ; i.e. contrary motion to an *octave, lower voice up, higher voice down, counterpoint
        melodic interval < -4
    (print "No battuta kind of motion...")
    (add-no-battuta-cst
        (fourth *motions) (first *h-intervals) (fourth *m-intervals-brut) (fourth *
            is-cf-bass-arr) (fourth *is-3rd-species-arr)
    ) ; 3rd species

    ; dissonant notes must be followed by the consonant note below
    (print "Dissonant notes must be followed by the consonant note below...")
    (add-h-dis-imp-cons-below-cst *m-ta-intervals-brut (first *is-cons-arr) (first *
        is-4th-species-arr)) ; TODO 4th species

```

```
    ; no second dissonance if the cantus firmus is at the bass and a octave/unisson precedes it
    (print "No second dissonance if the cantus firmus is at the bass...")
    (add-no-second-cst
        (third *h-intervals) (rest (first *h-intervals)) (rest (first *is-cf-bass-arr))
        (rest (first *is-4th-species-arr))
    ) ; TODO 4th species

    ; Marcel's rule
    (add-contrary-step-after-skip-cst *m-all-intervals *m-all-intervals-brut)


    ;======================================== COST FACTORS ====================================
    (set-cost-factors)
    (print "Imperfect consonances are preferred to perfect consonances...")
    (setq *fifth-cost  (gil::add-int-var-array-dom *sp* *cf-len (getparam-dom 'h-fifth-cost))) ; IntVar array representing the cost to have fifths
    (setq *octave-cost (gil::add-int-var-array-dom *sp* *cf-len (getparam-dom 'h-octave-cost))) ; IntVar array representing the cost to have octaves
    (add-cost-cst-if (first *h-intervals) gil::IRT_EQ 7 (first *is-cst-arr) *fifth-cost *h-fifth-cost*) ; *fifth-cost = 1 if *h-interval == 7
    (add-cost-cst-if (first *h-intervals) gil::IRT_EQ 0 (first *is-cst-arr) *octave-cost *h-octave-cost*) ; *octave-cost = 1 if *h-interval == 0
    (add-cost-to-factors *fifth-cost)
    (add-cost-to-factors *octave-cost)

    ; 3, 4) add off-key cost, m-degrees cost and tritons cost
    (set-general-costs-cst *total-cp-len *is-constrained-arr (collect-bot-array (butlast *is-constrained-arr) (rest *is-constrained-arr)))

    ; 5) contrary motion is preferred
    (add-cost-to-factors (fourth *motions))

    ; 6) cambiata notes are preferred (cons - dis - cons > cons - cons - cons)
    (print "Cambiata notes are preferred...")
    ; IntVar array representing the cost to have cambiata notes
    (setq *not-cambiata-cost (gil::add-int-var-array-dom *sp* *cf-last-index (getparam-dom 'non-cambiata-cost)))
    (add-cost-bool-cst-if *is-not-cambiata-arr *is-mostly-3rd-arr *not-cambiata-cost *non-cambiata-cost*)
    (add-cost-to-factors *not-cambiata-cost)

    ; 7) intervals between notes n and n+2 are prefered greater than zero
    (print "Intervals between notes n and n+2 are prefered different than zero...")
    ; IntVar array representing the cost to have intervals between notes n and n+2 equal to zero
    (setq *m2-eq-zero-cost (gil::add-int-var-array-dom *sp* *m2-len (getparam-dom 'two-beats-apart-cost)))
    (add-cost-cst-if
        *m2-intervals gil::IRT_EQ 0
        (collect-bot-array (butlast (butlast *is-constrained-arr)) (rest (rest *is-constrained-arr)))
        *m2-eq-zero-cost *two-beats-apart-cost*
    )
    (add-cost-to-factors *m2-eq-zero-cost)

    ; 8) add no syncopation cost
    (setq *no-syncope-cost (gil::add-int-var-array-dom *sp* *cf-penult-index (getparam-dom 'no-syncopation-cost)))
    (add-cost-cst-if
        (butlast (third *m-intervals)) gil::IRT_NQ 0
        (third *is-4th-species-arr)
        *no-syncope-cost
        *no-syncopation-cost*
    )
    (add-cost-to-factors *no-syncope-cost)


    ;======================================== COST FUNCTION ===================================
```

```
    (print "Cost function...")

    (loop for i from 0 to 3 do
        (setf (nth i *cons-cost) (gil::add-int-var-array *sp* *cf-last-index 0 1)) ; IntVar representing the cost to have a consonance
        (add-cost-bool-cst (nth i *is-cons-arr) (nth i *cons-cost)) ; *cons-cost = 1 if *is-cons-arr == 1
    )


    (print *extended-cp-domain)

    ; RETURN
    (if (eq species 5)
        ; then create the search engine
        ; (append (fux-search-engine *total-cp) (list species))
        (append (fux-search-engine *total-cp 5) '(5))
        ; else
        nil
    )
)
```

E.10 constraints.lisp

```
(in-package :fuxcp)

; Author: Thibault Wafflard
; Date: June 3, 2023
; This file contains all the functions adding constraints to the CSP.
; They are all called from the different species.


;================================================ CP CONSTRAINTS UTILS ============================


; add a single cost regarding if the relation rel-type(tested, cst-val) is true
(defun add-single-cost-cst (tested rel-type cst-val cost &optional (cost-value ONE))
    (let (
        (b (gil::add-bool-var *sp* 0 1)) ; to store the result of the test
    )
        (gil::g-rel-reify *sp* tested rel-type cst-val b) ; test the relation
        (gil::g-ite *sp* b cost-value ZERO cost) ; add the cost if the test is true
    )
)

; add a cost regarding if the relation rel-type(tested-var, cst-val) is true
(defun add-cost-cst (tested-var-arr rel-type cst-val costs &optional (cost-value ONE))
    (loop
        for cost in costs
        for tested in tested-var-arr
        do
            (add-single-cost-cst tested rel-type cst-val cost cost-value)
    )
)

; add a cost regarding if the relation rel-type(tested-var, cst-val) is true
; NOTE: the difference with add-cost-cst is that the cst-val is an array
(defun add-cost-multi-cst (tested-var-arr rel-type cst-val-arr costs &optional (cost-value ONE))
    (loop
        for cost in costs
        for tested in tested-var-arr
        for cst-val in cst-val-arr
        do
            (add-single-cost-cst tested rel-type cst-val cost cost-value)
```

```
    )
)

; add a cost regarding if the relation rel-type(tested-var, cst-val) is true AND is-cst is true
(defun add-cost-cst-if (tested-var-arr rel-type cst-val is-cst-arr costs &optional (cost-value ONE))
    (loop
        for cost in costs
        for tested in tested-var-arr
        for is-cst in is-cst-arr
        do
            (add-single-cost-cst-if tested rel-type cst-val is-cst cost cost-value)
    )
)

(defun add-single-cost-cst-if (tested rel-type cst-val is-cst cost cost-value)
    (let (
        (b (gil::add-bool-var *sp* 0 1)) ; to store the result of the test
        (b-and (gil::add-bool-var *sp* 0 1)) ; b and cst
    )
        (gil::g-rel-reify *sp* tested rel-type cst-val b)
        (gil::g-op *sp* b gil::BOT_AND is-cst b-and) ; b-and = b and cst
        (gil::g-ite *sp* b-and cost-value ZERO cost) ; add the cost if the test is true
    )
)

; add a cost regarding if the booleans are true in bool-arr
(defun add-cost-bool-cst (bool-arr costs &optional (cost-value ONE))
    (loop
        for b in bool-arr
        for cost in costs
        do
            (gil::g-ite *sp* b cost-value ZERO cost)
    )
)

; add a cost regarding if the booleans are true in bool-arr AND if is-cst is true in is-cst-arr
(defun add-cost-bool-cst-if (bool-arr is-cst-arr costs &optional (cost-value ONE))
    (loop
        for b in bool-arr
        for cst in is-cst-arr
        for cost in costs
        do
            (add-single-cost-bool-cst-if b cst cost cost-value)
    )
)

; add a cost regarding if b is true AND if cst is true
(defun add-single-cost-bool-cst-if (b cst cost cost-value)
    (let (
        (b-and (gil::add-bool-var *sp* 0 1)) ; b and cst
    )
        (gil::g-op *sp* b gil::BOT_AND cst b-and) ; b-and = b and cst
        (gil::g-ite *sp* b-and cost-value ZERO cost) ; add the cost if the test is true
    )
)

; add a cost regarding only if b AND cst are true (do not force ZERO if false)
(defun add-single-cost-bool-cst-eqv (b cst cost cost-value)
    (let (
        (b-and (gil::add-bool-var *sp* 0 1)) ; b and cst
    )
        (gil::g-op *sp* b gil::BOT_AND cst b-and) ; b-and = b and cst
        (gil::g-rel-reify *sp* cost gil::IRT_EQ cost-value b-and gil::RM_IMP) ; add the cost if the test is true
    )
)
```

```
; add constraints such that costs =
;   - 0 if m-degree in [0, 1, 2]
;   - 1 if m-degree in [3, 4, 12]
;   - 2 otherwise
; @m-all-intervals: all the melodic intervals of cp in a row
; @m-degrees-cost: the cost of each melodic interval
(defun add-m-degrees-cost-cst (m-all-intervals m-degrees-cost m-degrees-type &optional (
    is-cst-arr nil))
    (loop
    for m in m-all-intervals
    for c in m-degrees-cost
    for d in m-degrees-type
    do
        (let (
            (b-l3 (gil::add-bool-var *sp* 0 1)) ; true if m < 3
            (b-3 (gil::add-bool-var *sp* 0 1)) ; true if m == 3
            (b-4 (gil::add-bool-var *sp* 0 1)) ; true if m == 4
            (b-34 (gil::add-bool-var *sp* 0 1)) ; true if m in [3, 4]
            (b-5 (gil::add-bool-var *sp* 0 1)) ; true if m == 5
            (b-6 (gil::add-bool-var *sp* 0 1)) ; true if m == 6
            (b-7 (gil::add-bool-var *sp* 0 1)) ; true if m == 7
            (b-8 (gil::add-bool-var *sp* 0 1)) ; true if m == 8
            (b-9 (gil::add-bool-var *sp* 0 1)) ; true if m == 9
            (b-89 (gil::add-bool-var *sp* 0 1)) ; true if m in [8, 9]
            (b-10 (gil::add-bool-var *sp* 0 1)) ; true if m == 10
            (b-11 (gil::add-bool-var *sp* 0 1)) ; true if m == 11
            (b-1011 (gil::add-bool-var *sp* 0 1)) ; true if m in [10, 11]
            (b-12 (gil::add-bool-var *sp* 0 1)) ; true if m == 12
        )
            (gil::g-rel-reify *sp* m gil::IRT_LE 3 b-l3) ; m < 3
            (gil::g-rel-reify *sp* m gil::IRT_EQ 3 b-3) ; m = 3
            (gil::g-rel-reify *sp* m gil::IRT_EQ 4 b-4) ; m = 4
            (gil::g-op *sp* b-3 gil::BOT_OR b-4 b-34) ; m in [3, 4]
            (gil::g-rel-reify *sp* m gil::IRT_EQ 5 b-5) ; m = 5
            (gil::g-rel-reify *sp* m gil::IRT_EQ 6 b-6) ; m = 6
            (gil::g-rel-reify *sp* m gil::IRT_EQ 7 b-7) ; m = 7
            (gil::g-rel-reify *sp* m gil::IRT_EQ 8 b-8) ; m = 8
            (gil::g-rel-reify *sp* m gil::IRT_EQ 9 b-9) ; m = 9
            (gil::g-op *sp* b-8 gil::BOT_OR b-9 b-89) ; m in [8, 9]
            (gil::g-rel-reify *sp* m gil::IRT_EQ 10 b-10) ; m = 10
            (gil::g-rel-reify *sp* m gil::IRT_EQ 11 b-11) ; m = 11
            (gil::g-op *sp* b-10 gil::BOT_OR b-11 b-1011) ; m in [10, 11]
            (gil::g-rel-reify *sp* m gil::IRT_EQ 12 b-12) ; m = 12
            ; set costs
            (gil::g-rel-reify *sp* c gil::IRT_EQ *m-step-cost* b-l3 gil::RM_IMP)
            (gil::g-rel-reify *sp* c gil::IRT_EQ *m-third-cost* b-34 gil::RM_IMP)
            (gil::g-rel-reify *sp* c gil::IRT_EQ *m-fourth-cost* b-5 gil::RM_IMP)
            (gil::g-rel-reify *sp* c gil::IRT_EQ *m-tritone-cost* b-6 gil::RM_IMP)
            (gil::g-rel-reify *sp* c gil::IRT_EQ *m-fifth-cost* b-7 gil::RM_IMP)
            (gil::g-rel-reify *sp* c gil::IRT_EQ *m-sixth-cost* b-89 gil::RM_IMP)
            (gil::g-rel-reify *sp* c gil::IRT_EQ *m-seventh-cost* b-1011 gil::RM_IMP)
            (gil::g-rel-reify *sp* c gil::IRT_EQ *m-octave-cost* b-12 gil::RM_IMP)
            ; set types
            (gil::g-rel-reify *sp* d gil::IRT_EQ 2 b-l3 gil::RM_IMP)
            (gil::g-rel-reify *sp* d gil::IRT_EQ 3 b-34 gil::RM_IMP)
            (gil::g-rel-reify *sp* d gil::IRT_EQ 4 b-5 gil::RM_IMP)
            (gil::g-rel-reify *sp* d gil::IRT_EQ 1 b-6 gil::RM_IMP)
            (gil::g-rel-reify *sp* d gil::IRT_EQ 5 b-7 gil::RM_IMP)
            (gil::g-rel-reify *sp* d gil::IRT_EQ 6 b-89 gil::RM_IMP)
            (gil::g-rel-reify *sp* d gil::IRT_EQ 7 b-1011 gil::RM_IMP)
            (gil::g-rel-reify *sp* d gil::IRT_EQ 8 b-12 gil::RM_IMP)
        )
    )
)
```

```lisp
; add cost constraints such that a cost is added when a fifth or an octave is present in the 1st beat
; except for the 4th species where it is the 3rd beat
; @is-sync: true means it is the 4th species
(defun add-p-cons-cost-cst (&optional (is-sync nil))
    (setq *fifth-cost  (gil::add-int-var-array-dom *sp* *cf-penult-index (getparam-dom 'h-fifth-cost))) ; IntVar array representing the cost to have fifths
    (setq *octave-cost (gil::add-int-var-array-dom *sp* *cf-penult-index (getparam-dom 'h-octave-cost))) ; IntVar array representing the cost to have octaves
    (if is-sync
        ; then 4th species
        (add-h-inter-cost-cst (rest (third *h-intervals)))
        ; else
        (add-h-inter-cost-cst (restbutlast (first *h-intervals)))
    )
    (add-cost-to-factors *fifth-cost)
    (add-cost-to-factors *octave-cost)
)

; add cost constraints such that a cost is added when a fifth or an octave is present in @h-intervals
(defun add-h-inter-cost-cst (h-intervals)
        (add-cost-cst h-intervals gil::IRT_EQ 7 *fifth-cost *h-fifth-cost*) ; *fifth-cost = 1 if *h-interval == 7
        (add-cost-cst h-intervals gil::IRT_EQ 0 *octave-cost *h-octave-cost*) ; *octave-cost = 1 if *h-interval == 0
)

; Get the minimum cost possible for a counterpoint depending on the costs of the melodic intervals
; @m-len: number of melodic intervals
(defun get-min-m-cost (m-len)
    ; get the minimum cost for skips
    (setq min-skip-cost (min
        (getparam 'm-third-cost)
        (getparam 'm-fourth-cost)
        (getparam 'm-tritone-cost)
        (getparam 'm-fifth-cost)
        (getparam 'm-sixth-cost)
        (getparam 'm-seventh-cost)
        (getparam 'm-octave-cost)
    ))
    ; get the minimum number of skips
    (setq int-min-skip (ceiling (* (getparam 'min-skips-slider) m-len)))
    ; return the minimum cost
    (+
        (* int-min-skip min-skip-cost)
        (* (- m-len int-min-skip) (min (getparam 'm-step-cost) min-skip-cost))
    )
)

; setup the cost factors with the minimum cost possible
(defun set-cost-factors ()
    (setq m-len (length *m-all-intervals))
    (setq lb-max (max (ceiling (/ *cf-len 4)) (get-min-m-cost m-len)))
    ; (print (list "lb-max: " lb-max))
    (setq lb-i (floor (* (getparam 'irreverence-slider) (* 2 m-len))))
    ; (print (list "lb-i: " lb-i))
    (defparameter COST_LB (+ lb-max lb-i))
    ; (print '("COST_LB: " COST_LB))
    ; IntVar array representing all the cost factors
    (setq *cost-factors (gil::add-int-var-array *sp* *N-COST-FACTORS 0 COST_UB))
    ; IntVar representing the *total *cost
    (setq *total-cost (gil::add-int-var *sp* COST_LB COST_UB))
    (print 'debug123)
)

```

```lisp
; add general costs for most of the species
(defun set-general-costs-cst (&optional (cp-len *total-cp-len) (is-cst-arr1 nil) (is-cst-arr2
    nil))
    (let (
        (m-len (- cp-len 1))
    )
        ; 2) sharps and flats should be used sparingly
        (print "Sharps and flats should be used sparingly...")
        (setq *off-key-cost (gil::add-int-var-array-dom *sp* cp-len (getparam-dom 'borrow-cost))
            ) ; IntVar array representing the cost to have off-key notes
        (if (null is-cst-arr1)
            ; then
            (add-cost-bool-cst *is-cp-off-key-arr *off-key-cost *borrow-cost*)
            ; else
            (add-cost-bool-cst-if *is-cp-off-key-arr is-cst-arr1 *off-key-cost *borrow-cost*)
        )
        ; sum of the cost of the off-key notes
        (add-cost-to-factors *off-key-cost)

        ; 3) melodic intervals should be as small as possible
        (print "Melodic intervals should be as small as possible...")
        ; IntVar array representing the cost to have melodic large intervals
        (setq degrees-cost-domain
            (remove-duplicates (mapcar (lambda (x) (getparam x))
                (list 'm-step-cost 'm-third-cost 'm-fourth-cost 'm-tritone-cost 'm-fifth-cost '
                    m-sixth-cost 'm-seventh-cost 'm-octave-cost)
            ))
        )
        (setq *m-degrees-cost (gil::add-int-var-array-dom *sp* m-len degrees-cost-domain))
        (setq *m-degrees-type (gil::add-int-var-array *sp* m-len 1 8))
        (add-m-degrees-cost-cst *m-all-intervals *m-degrees-cost *m-degrees-type is-cst-arr2)
        (add-cost-to-factors *m-degrees-cost)
        (gil::g-count *sp* *m-degrees-type 2 gil::IRT_LQ (floor (* (- 1 (getparam '
            min-skips-slider)) m-len)))
    )
)

; merge lists intermittently such that the first element of the first list is followed by the
    first element of the second list, etc.
; attention: cp-len is lenght of the first list in cp-list and it should be 1 more than the
    lenght of the other lists
(defun merge-cp (cp-list total-cp)
    (let (
        (cp-len-1 (- (length (first cp-list)) 1))
        (n-list (length cp-list))
    )
        (loop
        for i from 0 below cp-len-1
        do
            (loop for j from 0 below n-list do
                (setf (nth (+ (* i n-list) j) total-cp) (nth i (nth j cp-list)))
            )
        )
        (gil::g-rel *sp* (lastone total-cp) gil::IRT_EQ (lastone (first cp-list)))
    )
)

; merge lists intermittently such that the first element of the first list is followed by the
    first element of the second list, etc.
; attention: lengths should be the same
(defun merge-cp-same-len (cp-list total-cp)
    (let (
        (cp-len (length (first cp-list)))
        (n-list (length cp-list))
    )
        (loop
        for i from 0 below cp-len
```

```
        do
            (loop for j from 0 below n-list do
                (setf (nth (+ (* i n-list) j) total-cp) (nth i (nth j cp-list)))
            )
        )
    )
)

; create the harmonic intervals between @cp and @cf in @h-intervals
(defun create-h-intervals (cp cf h-intervals)
    (loop
        for p in cp
        for q in cf
        for i in h-intervals do
            (inter-eq-cst *sp* p q i) ; add a constraint to *sp* such that i = |p - q| % 12
    )
)

; create the intervals between @line1 and @line2 in @intervals and @brut-intervals
(defun create-intervals (line1 line2 intervals brut-intervals)
    (loop
        for p in line1
        for q in line2
        for i in intervals
        for ib in brut-intervals
        do
            (inter-eq-cst-brut *sp* q p ib i) ; add a constraint to *sp* such that ib = p - q and i = |ib|
    )
)

; create the intervals between @line1 and @line2 in @intervals and @brut-intervals where @is-cst-arr is true
(defun create-intervals-for-cst (line1 line2 intervals brut-intervals is-cst-arr)
    (loop
        for p in line1
        for q in line2
        for i in intervals
        for ib in brut-intervals
        for is-cst in is-cst-arr
        do
            (inter-eq-cst-brut-for-cst *sp* q p ib i is-cst) ; add a constraint to *sp* such that ib = p - q and i = |ib|
    )
)

; create the melodic intervals of @cp in @m-intervals and @m-intervals-brut
; @is-cst-arr is a list of booleans indicating whether the melodic interval is constrained or not
(defun create-m-intervals-self (cp m-intervals m-intervals-brut &optional (is-cst-arr nil))
    (if is-cst-arr
        ; then
        (create-intervals-for-cst (butlast cp) (rest cp) m-intervals m-intervals-brut is-cst-arr)
        ; else
        (create-intervals (butlast cp) (rest cp) m-intervals m-intervals-brut)
    )
)

; create an array of IntVar with the melodic interval between each arsis and its following thesis
(defun create-m-intervals-next-meas (cp-arsis cp m-intervals-arsis m-intervals-arsis-brut)
    (create-intervals cp-arsis (rest cp) m-intervals-arsis m-intervals-arsis-brut)
)

; create the melodic intervals two positions apart of @cp in @m2-intervals and @m2-intervals-brut
```

```lisp
(defun create-m2-intervals (cp m2-intervals m2-intervals-brut)
    (create-intervals (butlast (butlast cp)) (rest (rest cp)) m2-intervals m2-intervals-brut)
)

; create the melodic intervals between the thesis of @cp and the arsis of @cp-arsis in
    @m-intervals and @m-intervals-brut
(defun create-m-intervals-in-meas (cp cp-arsis ta-intervals ta-intervals-brut)
    (create-intervals (butlast cp) cp-arsis ta-intervals ta-intervals-brut)
)

; create the brut melodic intervals of @cf in @cf-brut-m-intervals
(defun create-cf-brut-m-intervals (cf cf-brut-m-intervals)
    (loop
        for p in (butlast cf)
        for q in (rest cf)
        for i in cf-brut-m-intervals do
            (let (
                (ib (inter q p t))
            )
                (gil::g-rel *sp* i gil::IRT_EQ ib)
            )
    )
)

; create the boolean array @is-p-cons-arr indicating if the interval is a perfect consonance or
    not
(defun create-is-p-cons-arr (h-intervals is-p-cons-arr)
    (loop
        for i in h-intervals
        for p in is-p-cons-arr
        do
            (let (
                (b-7 (gil::add-bool-var *sp* 0 1))
                (b-0 (gil::add-bool-var *sp* 0 1))
            )
                (gil::g-rel-reify *sp* i gil::IRT_EQ 7 b-7) ; b-7 = (i == 7) -> the interval is
                    a fifth
                (gil::g-rel-reify *sp* i gil::IRT_EQ 0 b-0) ; b-0 = (i == 0) -> the interval is
                    an octave
                (gil::g-op *sp* b-0 gil::BOT_OR b-7 p) ; p = b-7 || b-0
            )
    )
)

; create the boolean array @is-cf-bass-arr indicating if the cantus firmus is the bass or not
(defun create-is-cf-bass-arr (cp cf is-cf-bass-arr)
    (loop
        for p in cp
        for q in cf
        for b in is-cf-bass-arr
        do
            (gil::g-rel-reify *sp* p gil::IRT_GQ q b) ; b = (p >= q)
    )
)

; create an array of BoolVar such that is-ta-dim-arr is true if the note is a diminution:
; 1 -> inter(thesis, arsis) == 1 or 2 && inter(thesis, thesis + 1) == 3 or 4 && inter(arsis,
    thesis + 1) == 1 or 2
; @m-intervals-ta: the melodic interval between each thesis and its following arsis
; @m-intervals: the melodic interval between each thesis and its following thesis
; @m-intervals-arsis: the melodic interval between each arsis and its following thesis
; @is-ta-dim-arr: the array of BoolVar to fill
(defun create-is-ta-dim-arr (m-intervals-ta m-intervals m-intervals-arsis is-ta-dim-arr)
    (loop
        for mta in m-intervals-ta ; inter(thesis, arsis)
        for mtt in m-intervals ; inter(thesis, thesis + 1)
        for mat in m-intervals-arsis ; inter(arsis, thesis + 1)
```

```
        for b in is-ta-dim-arr ; the BoolVar to create
        do
            (let (
                (btt3 (gil::add-bool-var *sp* 0 1)) ; for mtt == 3
                (btt4 (gil::add-bool-var *sp* 0 1)) ; for mtt == 4
                (bta-second (gil::add-bool-var *sp* 0 1)) ; for mat <= 2
                (btt-third (gil::add-bool-var *sp* 0 1)) ; for mtt == 3 or 4
                (bat-second (gil::add-bool-var *sp* 0 1)) ; for mta <= 2
                (b-and (gil::add-bool-var *sp* 0 1)) ; temporary BoolVar
            )
                (gil::g-rel-reify *sp* mtt gil::IRT_EQ 3 btt3) ; btt3 = (mtt == 3)
                (gil::g-rel-reify *sp* mtt gil::IRT_EQ 4 btt4) ; btt4 = (mtt == 4)
                (gil::g-rel-reify *sp* mta gil::IRT_LQ 2 bta-second) ; bta2 = (mta <= 2)
                (gil::g-rel-reify *sp* mat gil::IRT_LQ 2 bat-second) ; bat1 = (mat <= 2)
                (gil::g-op *sp* btt3 gil::BOT_OR btt4 btt-third) ; btt-third = btt3 || btt4
                (gil::g-op *sp* bta-second gil::BOT_AND btt-third b-and) ; temporay operation
                (gil::g-op *sp* b-and gil::BOT_AND bat-second b) ; b = bta-second && btt-third
                    && bat-second
            )
    )
)

; create an array of BoolVar
; 1 -> inter(cp, cf) <= 4 && cf getting closer to cp
(defun create-is-nbour-arr (h-intervals-abs is-cf-bass-arr cf-brut-m-intervals is-nbour-arr)
    (loop
        for hi in (butlast h-intervals-abs)
        for bass in (butlast is-cf-bass-arr)
        for mi in cf-brut-m-intervals
        for n in is-nbour-arr
        do
            (let (
                (b-hi (gil::add-bool-var *sp* 0 1)) ; for (hi <= 4)
                (b-cfu (gil::add-bool-var *sp* 0 1)) ; for cf going up
                (b-cfgc (gil::add-bool-var *sp* 0 1)) ; for cf getting closer to cp
            )
                (gil::g-rel-reify *sp* hi gil::IRT_LQ 4 b-hi) ; b-hi = (hi <= 4)
                (gil::g-rel-reify *sp* mi gil::IRT_GQ 0 b-cfu) ; b-cfu = (mi >= 0)
                (gil::g-op *sp* bass gil::BOT_EQV b-cfu b-cfgc) ; b-cfgc = (bass == b-cfu)
                (gil::g-op *sp* b-hi gil::BOT_AND b-cfgc n) ; n = b-hi && b-cfgc
            )
    )
)

; TODO: new version below should be used instead of this one
; create an array of BoolVar
; 1 -> 5 quarter notes strictly ups or downs and are linked by joint degrees
; Note: the rule is applied measure by measure
(defun create-is-5qn-linked-arr (m-all-intervals m-all-intervals-brut is-5qn-linked-arr)
    (loop
    for i from 0 to (- (length m-all-intervals) 3)
    for m1 in m-all-intervals
    for m2 in (rest m-all-intervals)
    for m3 in (rest (rest m-all-intervals))
    for m4 in (rest (rest (rest m-all-intervals)))
    for mb1 in m-all-intervals-brut
    for mb2 in (rest m-all-intervals-brut)
    for mb3 in (rest (rest m-all-intervals-brut))
    for mb4 in (rest (rest (rest m-all-intervals-brut)))
    for b in is-5qn-linked-arr
    do
        (if (eq (mod i 4) 0)
            ; then
            (let (
                (b1 (gil::add-bool-var *sp* 0 1)) ; (m1 <= 2)
                (b2 (gil::add-bool-var *sp* 0 1)) ; (m2 <= 2)
                (b3 (gil::add-bool-var *sp* 0 1)) ; (m3 <= 2)
```

```
                (b4 (gil::add-bool-var *sp* 0 1)) ; (m4 <= 2)
                (bb1 (gil::add-bool-var *sp* 0 1)) ; (mb1 > 0)
                (bb2 (gil::add-bool-var *sp* 0 1)) ; (mb2 > 0)
                (bb3 (gil::add-bool-var *sp* 0 1)) ; (mb3 > 0)
                (bb4 (gil::add-bool-var *sp* 0 1)) ; (mb4 > 0)
                (b-and1 (gil::add-bool-var *sp* 0 1)) ; (b1 && b2)
                (b-and2 (gil::add-bool-var *sp* 0 1)) ; (b3 && b4)
                (b-and3 (gil::add-bool-var *sp* 0 1)) ; (b-and1 && b-and2)
                (b-eq1 (gil::add-bool-var *sp* 0 1)) ; (mb1 == mb2)
                (b-eq2 (gil::add-bool-var *sp* 0 1)) ; (mb3 == mb3)
                (b-eq3 (gil::add-bool-var *sp* 0 1)) ; (b-eq1 == b-eq2)
            )
                (gil::g-rel-reify *sp* m1 gil::IRT_LQ 2 b1) ; b1 = (m1 <= 2)
                (gil::g-rel-reify *sp* m2 gil::IRT_LQ 2 b2) ; b2 = (m2 <= 2)
                (gil::g-rel-reify *sp* m3 gil::IRT_LQ 2 b3) ; b3 = (m3 <= 2)
                (gil::g-rel-reify *sp* m4 gil::IRT_LQ 2 b4) ; b4 = (m4 <= 2)
                (gil::g-rel-reify *sp* mb1 gil::IRT_GQ 0 bb1) ; bb1 = (mb1 > 0)
                (gil::g-rel-reify *sp* mb2 gil::IRT_GQ 0 bb2) ; bb2 = (mb2 > 0)
                (gil::g-rel-reify *sp* mb3 gil::IRT_GQ 0 bb3) ; bb3 = (mb3 > 0)
                (gil::g-rel-reify *sp* mb4 gil::IRT_GQ 0 bb4) ; bb4 = (mb4 > 0)
                (gil::g-op *sp* b1 gil::BOT_AND b2 b-and1) ; b-and1 = b1 && b2
                (gil::g-op *sp* b3 gil::BOT_AND b4 b-and2) ; b-and2 = b3 && b4
                (gil::g-op *sp* b-and1 gil::BOT_AND b-and2 b-and3) ; b-and3 = b-and1 && b-and2
                (gil::g-op *sp* bb1 gil::BOT_EQV bb2 b-eq1) ; b-eq1 = (bb1 == bb2)
                (gil::g-op *sp* bb3 gil::BOT_EQV bb4 b-eq2) ; b-eq2 = (bb3 == bb4)
                (gil::g-op *sp* b-eq1 gil::BOT_EQV b-eq2 b-eq3) ; b-eq3 = (b-eq1 == b-eq2)
                (gil::g-op *sp* b-and3 gil::BOT_AND b-eq3 b) ; b = b-and3 && b-eq3
            )
        )
    )
)

; create an array of BoolVar representing if the second note is not cambiata
(defun create-is-not-cambiata-arr (is-cons-arr2 is-cons-arr3 m-intervals is-not-cambiata-arr)
    (loop
    for b2 in is-cons-arr2
    for b3 in is-cons-arr3
    for m in m-intervals
    for b in is-not-cambiata-arr
    do
        (let (
            (b-m (gil::add-bool-var *sp* 0 1)) ; (m <= 2)
            (b-and (gil::add-bool-var *sp* 0 1)) ; (b2 && b3)
        )
            (gil::g-op *sp* b2 gil::BOT_AND b3 b-and) ; b-and = b2 && b3
            (gil::g-rel-reify *sp* m gil::IRT_LQ 2 b-m) ; b-m = (m <= 2)
            (gil::g-op *sp* b-and gil::BOT_AND b-m b) ; b = b-and && b-m
        )
    )
)

; create an array of BoolVar representing if there is no syncopation
(defun create-is-no-syncope-arr (m-intervals is-no-syncope-arr)
    (loop
    for m in (butlast m-intervals)
    for b in is-no-syncope-arr
    do
        (gil::g-rel-reify *sp* m gil::IRT_NQ 0 b)
    )
)

; add constraints such that @b-member is true iff @candidate is a member of @member-list
(defun add-is-member-cst (candidate member-list b-member)
    (let (
        (results (gil::add-int-var-array *sp* (length member-list) 0 1)) ; where candidate == m
        (sum (gil::add-int-var *sp* 0 (length member-list))) ; sum(results)
    )
```

```
        (loop
        for m in member-list
        for r in results
        do
            (let (
                (b1 (gil::add-bool-var *sp* 0 1)) ; b1 = (candidate == m)
            )
                (gil::g-rel-reify *sp* candidate gil::IRT_EQ m b1) ; b1 = (candidate == m)
                (gil::g-ite *sp* b1 ONE ZERO r) ; r = (b1 ? 1 : 0)
            )
        )
        (gil::g-sum *sp* sum results) ; sum = sum(results)
        (gil::g-rel-reify *sp* sum gil::IRT_GR 0 b-member) ; b-member = (sum >= 1)
    )
)

; create an array of BoolVar
; 1 -> the harmonic interval is member of the set (consonances set by default)
(defun create-is-member-arr (h-intervals cons-arr &optional (cons-set ALL_CONS))
    (loop
    for h in h-intervals
    for b in cons-arr
    do
        (add-is-member-cst h cons-set b)
    )
)

; add the constraint such that the harmonies in @h-intervals are consonances expect the penultimate note (specific rule)
; @len: the length of the counterpoint
; @cf-penult-index: the index of penultimate note in the counterpoint
; @h-intervals: the array of harmonic intervals
; @penult-dom-var: the domain of the penultimate note
(defun add-h-cons-cst (len cf-penult-index h-intervals &optional (penult-dom-var PENULT_CONS_VAR))
    (loop for i from 0 below len do
        (setq h-interval (nth i h-intervals))
        (if (eq i cf-penult-index) ; if it is the penultimate note
            ; then add major sixth + minor third by default
            (add-penult-dom-cst h-interval penult-dom-var)
            ; else add all consonances
            (if (not (null h-interval))
                (gil::g-member *sp* ALL_CONS_VAR h-interval)
            )
        )
    )
)

; add the constraint such that the penultimate note belongs to the domain @penult-dom-var
(defun add-penult-dom-cst (h-interval penult-dom-var)
    (if (getparam 'penult-rule-check)
        (gil::g-member *sp* penult-dom-var h-interval)
    )
)

; add the constraint such that is-cst-arr[i] => is-cons-arr[i] is true
; -is-cons-arr: array of BoolVar, 1 -> the harmonic interval is a consonance
; -is-cst-arr: array of BoolVar, 1 -> the note is constrained by a species
(defun add-h-cons-cst-if (is-cons-arr is-cst-arr)
    (loop
    for is-cons in is-cons-arr
    for is-cst in is-cst-arr
    do
        (gil::g-op *sp* is-cst gil::BOT_IMP is-cons 1) ; (is-cst => is-cons) = 1
    )
)
```

```
; add the constraint such that h-intervals[i] belongs to ALL_CONS_VAR is-no-syncope-arr[i] is
    true
; in other words, if there is no syncopation the note cannot be dissonant
(defun add-no-sync-h-cons (h-intervals is-no-syncope-arr)
    (loop
    for h in h-intervals
    for b in is-no-syncope-arr
    do
        (loop for d in DIS do
            (gil::g-rel-reify *sp* h gil::IRT_NQ d b gil::RM_IMP) ; b => (h != d)
        )
    )
)

; for future work: should use not(nth i is-cons-arr) instead of add a constraint for each
    dissonance in DIS
; -len: length of the harmonic array
; -cf-penult-index: index of the penultimate note in the counterpoint
; -h-intervals-arsis: harmonic intervals of the arsis of the counterpoint
; -is-ta-dim-arr: array of BoolVar, 1 -> the note in arsis is a diminution
; -penult-dom-var: domain of the penultimate note
(defun add-h-cons-arsis-cst (len cf-penult-index h-intervals-arsis is-ta-dim-arr &optional (
    penult-dom-var PENULT_CONS_VAR))
    (loop
    for i from 0 below len
    for b in is-ta-dim-arr
    do
        (if (eq i cf-penult-index) ; if it is the penultimate note
            ; then add major sixth + minor third
            (add-penult-dom-cst (nth i h-intervals-arsis) penult-dom-var)
            ; else dissonance implies there is a diminution
            (loop for d in DIS do
                (gil::g-rel-reify *sp* (nth i h-intervals-arsis) gil::IRT_EQ d b gil::RM_PMI)
            )
        )
    )
)

; add the constraint such that (c3 OR (c2 AND c4)) AND (c3 OR dim) is true,
; where : - cn represents if the nth note of the measure is consonant
;         - dim represents if the 3rd note is a diminution
(defun add-h-dis-or-cons-3rd-cst (is-cons-2nd is-cons-3rd is-cons-4th is-dim &optional (
    is-cst-arr nil))
    (loop
    for b-c2nd in is-cons-2nd
    for b-c3rd in is-cons-3rd
    for b-c4th in is-cons-4th
    for b-dim in is-dim
    do
        (let (
            (b-and1 (gil::add-bool-var *sp* 0 1)) ; s.f. b-c2nd AND b-c4th
        )
            (gil::g-op *sp* b-c2nd gil::BOT_AND b-c4th b-and1) ; b-and1 = b-c2nd AND b-c4th
            (gil::g-op *sp* b-c3rd gil::BOT_OR b-dim 1) ; b-and2 = b-c2nd AND b-c4th AND b-dim
        )
    )
)

; add constraints such that
;   any dissonant note implies that it is followed by the next consonant note below
; @m-succ-intervals-brut: list of IntVar, s.f. brut melodic intervals between thesis and arsis
; @is-cons-arr: list of BoolVar, s.f. 1 -> the note is consonant
; @is-cst-arr: list of BoolVar, s.f. 1 -> the note is constrained by a species
(defun add-h-dis-imp-cons-below-cst (m-succ-intervals-brut is-cons-arr &optional (is-cst-arr nil
    ))
    (loop
    for m in m-succ-intervals-brut
```

```
    for b in is-cons-arr
    for i from 0 below (length m-succ-intervals-brut)
    do
        (let (
            (b-not (gil::add-bool-var *sp* 0 1)) ; s.f. !b (dissonance)
            (is-cst (true-if-null is-cst-arr i)) ; s.f. is-cst = 1 -> the note is constrained by a species
            (b-and (gil::add-bool-var *sp* 0 1)) ; s.f. b-not && is-cst
        )
            (gil::g-op *sp* b gil::BOT_EQV FALSE b-not) ; b-not = !b (dissonance)
            (gil::g-op *sp* b-not gil::BOT_AND is-cst b-and) ; b-and = b-not && is-cst
            (gil::g-rel-reify *sp* m gil::IRT_LE 0 b-and gil::RM_IMP) ; b-and => m < 0
            (gil::g-rel-reify *sp* m gil::IRT_GQ -2 b-and gil::RM_IMP) ; b-and => m >= -2
        )
    )
)

; add constraints such that if a melodic interval is greater than one step (2)
; then the next melodic interval should be one step and in the opposite direction
(defun add-contrary-step-after-skip-cst (m-all-intervals m-all-intervals-brut)
    (if (not (getparam 'con-m-after-skip-check))
        (return-from add-contrary-step-after-skip-cst)
    )
    (loop
    for m in m-all-intervals
    for m+1 in (rest m-all-intervals)
    for mb in m-all-intervals-brut
    for mb+1 in (rest m-all-intervals-brut)
    do
        (let (
            (b-skip (gil::add-bool-var *sp* 0 1)) ; m > 2
            (b-mb-up (gil::add-bool-var *sp* 0 1)) ; mb > 0
            (b-mb+1-down (gil::add-bool-var *sp* 0 1)) ; mb+1 < 0
            (b-contrary (gil::add-bool-var *sp* 0 1)) ; b-mb-up <=> b-mb+1-down
        )
            (gil::g-rel-reify *sp* m gil::IRT_GR 2 b-skip) ; b-skip := m > 2
            (gil::g-rel-reify *sp* mb gil::IRT_GR 0 b-mb-up) ; b-mb-up := mb > 0
            (gil::g-rel-reify *sp* mb+1 gil::IRT_LE 0 b-mb+1-down) ; b-mb+1-down := mb+1 < 0
            (gil::g-op *sp* b-mb-up gil::BOT_EQV b-mb+1-down b-contrary) ; b-contrary := b-mb-up <=> b-mb+1-down
            (gil::g-rel-reify *sp* m+1 gil::IRT_LQ 2 b-skip gil::RM_IMP) ; b-skip => m+1 <= 2
            (gil::g-op *sp* b-skip gil::BOT_IMP b-contrary 1) ; b-skip => b-contrary
        )
    )
)

; is-5qn-linked-arr implies that is-cons-arr1 (supposed to always be true) and is-cons-arr3 are true
(defun add-linked-5qn-cst (is-cons-arr3 is-5qn-linked-arr)
    (loop
    ; for b1 in is-cons-arr1
    for b3 in is-cons-arr3
    for b in is-5qn-linked-arr
    do
        (gil::g-op *sp* b gil::BOT_IMP b3 1) ; b => b3
    )
)

; add the constraint such that there cp is never equal to cf
(defun add-no-unisson-at-all-cst (cp cf &optional (is-cst-arr nil))
    (loop
        for p in cp
        for q in cf
        for i from 0 below (length cp)
        do
            (rel-reify-if p gil::IRT_NQ q (nth i is-cst-arr))
    )
```

```
)

; add the constraint such that there is no unisson unless it is the first or last note
(defun add-no-unisson-cst (cp cf)
    (add-no-unisson-at-all-cst (restbutlast cp) (restbutlast cf))
)

; add the constraint such that the first harmonic interval is a perfect consonance
(defun add-p-cons-start-cst (h-intervals)
    (gil::g-member *sp* P_CONS_VAR (first h-intervals))
)

; add the constraint such that the last harmonic interval is a perfect consonance
(defun add-p-cons-end-cst (h-intervals)
    (gil::g-member *sp* P_CONS_VAR (lastone h-intervals))
)

; add the constraint such that the first and last harmonic interval are 0 if cp is at the bass
;   not(is-cf-bass[0, 0]) => h-interval[0, 0] = 0
;   not(is-cf-bass[-1, -1]) => h-interval[-1, -1] = 0
; @h-interval: the harmonic interval array
; @is-cf-bass-arr: boolean variables indicating if cf is at the bass
(defun add-tonic-tuned-cst (h-interval is-cf-bass-arr)
    (let (
        (bf-not (gil::add-bool-var *sp* 0 1)) ; for !(first is-cf-bass-arr)
        (bl-not (gil::add-bool-var *sp* 0 1)) ; for !(lastone is-cf-bass-arr)
    )
        (gil::g-op *sp* (first is-cf-bass-arr) gil::BOT_EQV FALSE bf-not) ; bf-not = !(first is-cf-bass-arr)
        (gil::g-op *sp* (lastone is-cf-bass-arr) gil::BOT_EQV FALSE bl-not) ; bl-not = !(lastone is-cf-bass-arr)
        (gil::g-rel-reify *sp* (first h-interval) gil::IRT_EQ 0 bf-not gil::RM_IMP) ; bf-not => h-interval[0, 0] = 0
        (gil::g-rel-reify *sp* (lastone h-interval) gil::IRT_EQ 0 bl-not gil::RM_IMP) ; bl-not => h-interval[-1, -1] = 0
    )
)

; add the constraint such that the harmonic interval is a perfect consonance if it is constrained by a species
(defun add-p-cons-cst-if (h-inter is-cst)
    (let (
        (b-fifth (gil::add-bool-var *sp* 0 1)) ; b-fifth = h-inter is a fifth
        (b-octave (gil::add-bool-var *sp* 0 1)) ; b-octave = h-inter is an octave
        (b-p-cons (gil::add-bool-var *sp* 0 1)) ; b-p-cons = h-inter is a perfect consonance
    )
        (gil::g-rel-reify *sp* h-inter gil::IRT_EQ 7 b-fifth) ; b-fifth = h-inter is a fifth
        (gil::g-rel-reify *sp* h-inter gil::IRT_EQ 0 b-octave) ; b-octave = h-inter is an octave
        (gil::g-op *sp* b-fifth gil::BOT_OR b-octave b-p-cons) ; b-p-cons = b-fifth or b-octave
        (gil::g-op *sp* is-cst gil::BOT_IMP b-p-cons 1) ; is-cst => b-p-cons
    )
)

(defun add-penult-cons-cst (b-bass h-interval &optional (and-cond nil))
    (if (getparam 'penult-rule-check)
        (if (null and-cond)
            (gil::g-ite *sp* b-bass NINE THREE h-interval)
            (and-ite b-bass NINE THREE h-interval and-cond)
        )
    )
)

; add a constraint such that there is no seventh harmonic interval if cf is at the top
(defun add-no-seventh-cst (h-intervals is-cf-bass-arr &optional (is-cst-arr nil))
    (loop
    for h in h-intervals
    for b in is-cf-bass-arr
```

```lisp
    for i from 0 below (length h-intervals)
    do
        (let (
            (b-not (gil::add-bool-var *sp* 0 1)) ; b-not = !b
            (is-cst (nth i is-cst-arr)) ; is-cst = is-cst-arr[i]
            (b-and (gil::add-bool-var *sp* 0 1)) ; b-and = b-not and is-cst
        )
            (gil::g-op *sp* b gil::BOT_EQV FALSE b-not) ; b-not = !b
            (if (null is-cst)
                (gil::g-op *sp* b-not gil::BOT_AND TRUE b-and) ; b-and = b-not
                (gil::g-op *sp* b-not gil::BOT_AND is-cst b-and) ; b-and = b-not and is-cst
            )
            (gil::g-rel-reify *sp* h gil::IRT_NQ 10 b-and gil::RM_IMP) ; b-and => h != 10
            (gil::g-rel-reify *sp* h gil::IRT_NQ 11 b-and gil::RM_IMP) ; b-and => h != 11
        )
    )
)

; add a constraint such that there is no second harmonic interval if:
;   - cf is at the bass AND
;   - octave/unisson harmonic interval precedes it
(defun add-no-second-cst (h-intervals-arsis h-intervals-thesis is-cf-bass-arr &optional (is-cst-arr nil))
    (loop
    for ia in h-intervals-arsis
    for it in h-intervals-thesis
    for b in is-cf-bass-arr
    for i from 0 below (length h-intervals-arsis)
    do
        (let (
            (b-uni (gil::add-bool-var *sp* 0 1)) ; b-uni = (ia == 0)
            (b-and (gil::add-bool-var *sp* 0 1)) ; b-and = b AND b-uni
            (is-cst (true-if-null is-cst-arr i)) ; is-cst = is-cst-arr[i] or TRUE
            (b-and-cst (gil::add-bool-var *sp* 0 1)) ; b-and-cst = b-and AND is-cst
        )
            (gil::g-rel-reify *sp* ia gil::IRT_EQ 0 b-uni) ; b-uni = (ia == 0)
            (gil::g-op *sp* b gil::BOT_AND b-uni b-and) ; b-and = b AND b-uni
            (gil::g-op *sp* b-and gil::BOT_AND is-cst b-and-cst) ; b-and-cst = b-and AND is-cst
            (gil::g-rel-reify *sp* it gil::IRT_NQ 1 b-and-cst gil::RM_IMP)
            (gil::g-rel-reify *sp* it gil::IRT_NQ 2 b-and-cst gil::RM_IMP)
        )
    )
)

; add a constraint such that there is no melodic interval greater than @jump (8, minor 6th by default)
(defun add-no-m-jump-cst (m-intervals &optional (jump 8))
    (gil::g-rel *sp* m-intervals gil::IRT_LQ jump)
)

; add a constraint such that m-intervals does not belong to [9, 10, 11]
(defun add-no-m-jump-extend-cst (m-intervals &optional (is-cst-arr nil))
    (if (null is-cst-arr)
        ; then
        (progn
        (gil::g-rel *sp* m-intervals gil::IRT_NQ 9)
        (gil::g-rel *sp* m-intervals gil::IRT_NQ 10)
        (gil::g-rel *sp* m-intervals gil::IRT_NQ 11)
        )
        ; else
        (progn
        (loop
            for m in m-intervals
            for b in is-cst-arr
            do
                (gil::g-rel-reify *sp* m gil::IRT_NQ 9 b gil::RM_IMP)
                (gil::g-rel-reify *sp* m gil::IRT_NQ 10 b gil::RM_IMP)
```

```lisp
                (gil::g-rel-reify *sp* m gil::IRT_NQ 11 b gil::RM_IMP)
        )
        )
    )
)

; add melodic interval constraints such that:
;   - minor sixth intervals and octave intervals implies that is-nbour is true
;   - no seventh intervals
(defun add-m-inter-arsis-cst (m-intervals-ta is-nbour-arr)
    (loop
        for m in m-intervals-ta
        for n in is-nbour-arr
        do
            (let (
                (b-maj-six (gil::add-bool-var *sp* 0 1)) ; for (m = 9)
                (b-min-sev (gil::add-bool-var *sp* 0 1)) ; for (m == 10)
                (b-maj-sev (gil::add-bool-var *sp* 0 1)) ; for (m == 11)
                (b-or (gil::add-bool-var *sp* 0 1)) ; temporary variable for (b-min-sev or
                    b-maj-sev)
            )
                (gil::g-rel-reify *sp* m gil::IRT_EQ 12 n gil::RM_PMI) ; m == 12 implies n is
                    true
                (gil::g-rel-reify *sp* m gil::IRT_EQ 9 b-maj-six) ; b-maj-six = (m == 9)
                (gil::g-rel-reify *sp* m gil::IRT_EQ 10 b-min-sev) ; b-min-sev = (m == 10)
                (gil::g-rel-reify *sp* m gil::IRT_EQ 11 b-maj-sev) ; b-maj-sev = (m == 11)
                (gil::g-op *sp* b-min-sev gil::BOT_OR b-maj-sev b-or) ; b-or = (b-min-sev or
                    b-maj-sev)
                (gil::g-op *sp* b-or gil::BOT_OR b-maj-six 0) ; not (b-min-sev || b-maj-sev)
            )
    )
)

; add melodic interval constraints such that there is no chromatic interval:
;   - no m1 == 1 and m2 == 2 OR
;   - no m1 == -1 and m2 == -2
(defun add-no-chromatic-m-cst (m-intervals-brut m2-intervals-brut)
    (loop
        for m1 in (rest m-intervals-brut)
        for m2 in m2-intervals-brut do
        (let (
            (b1 (gil::add-bool-var *sp* 0 1)) ; for (m1 == 1)
            (b2 (gil::add-bool-var *sp* 0 1)) ; for (m2 == 2)
            (b3 (gil::add-bool-var *sp* 0 1)) ; for (m1 == -1)
            (b4 (gil::add-bool-var *sp* 0 1)) ; for (m2 == -2)
        )
            (gil::g-rel-reify *sp* m1 gil::IRT_EQ 1 b1) ; b1 = (m1 == 1)
            (gil::g-rel-reify *sp* m2 gil::IRT_EQ 2 b2) ; b2 = (m2 == 2)
            (gil::g-op *sp* b1 gil::BOT_AND b2 0) ; not(b1 and b2)
            (gil::g-rel-reify *sp* m1 gil::IRT_EQ -1 b3) ; b3 = (m1 == -1)
            (gil::g-rel-reify *sp* m2 gil::IRT_EQ -2 b4) ; b4 = (m2 == -2)
            (gil::g-op *sp* b3 gil::BOT_AND b4 0) ; not(b3 and b4)
        )
    )
)

; add melodic interval constraints such that there is no chromatic interval:
;   - no m1 == 1 and m2 == 1 OR
;   - no m1 == -1 and m2 == -1
; @m-intervals-brut: list of all the melodic intervals
(defun add-no-chromatic-allm-cst (m-intervals-brut)
    (loop
        for m1 in m-intervals-brut
        for m2 in (rest m-intervals-brut) do
        (let (
            (b1 (gil::add-bool-var *sp* 0 1)) ; for (m1 == 1)
            (b2 (gil::add-bool-var *sp* 0 1)) ; for (m2 == 1)
```

```
            (b3 (gil::add-bool-var *sp* 0 1)) ; for (m1 == -1)
            (b4 (gil::add-bool-var *sp* 0 1)) ; for (m2 == -1)
        )
            (gil::g-rel-reify *sp* m1 gil::IRT_EQ 1 b1) ; b1 = (m1 == 1)
            (gil::g-rel-reify *sp* m2 gil::IRT_EQ 1 b2) ; b2 = (m2 == 1)
            (gil::g-op *sp* b1 gil::BOT_AND b2 0) ; not(b1 and b2)
            (gil::g-rel-reify *sp* m1 gil::IRT_EQ -1 b3) ; b3 = (m1 == -1)
            (gil::g-rel-reify *sp* m2 gil::IRT_EQ -1 b4) ; b4 = (m2 == -1)
            (gil::g-op *sp* b3 gil::BOT_AND b4 0) ; not(b3 and b4)
        )
    )
)

(defun create-motions (m-intervals-brut cf-brut-m-intervals motions costs)
    (loop
        for p in m-intervals-brut
        for q in cf-brut-m-intervals
        for m in motions
        for c in costs
        do
            (let (
                ; boolean variables
                (b-pu (gil::add-bool-var *sp* 0 1)) ; boolean p up
                (b-qu (gil::add-bool-var *sp* 0 1)) ; boolean q up
                (b-ps (gil::add-bool-var *sp* 0 1)) ; boolean p stays
                (b-qs (gil::add-bool-var *sp* 0 1)) ; boolean q stays
                (b-pd (gil::add-bool-var *sp* 0 1)) ; boolean p down
                (b-qd (gil::add-bool-var *sp* 0 1)) ; boolean q down
                ; direct motion
                (b-both-up (gil::add-bool-var *sp* 0 1)) ; boolean both up
                (b-both-stays (gil::add-bool-var *sp* 0 1)) ; boolean both stays
                (b-both-down (gil::add-bool-var *sp* 0 1)) ; boolean both down
                (dm-or1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
                (dm-or2 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
                ; oblique motion
                (b-pu-qs (gil::add-bool-var *sp* 0 1)) ; boolean p up and q stays
                (b-pd-qs (gil::add-bool-var *sp* 0 1)) ; boolean p down and q stays
                (b-ps-qu (gil::add-bool-var *sp* 0 1)) ; boolean p stays and q up
                (b-ps-qd (gil::add-bool-var *sp* 0 1)) ; boolean p stays and q down
                (om-or1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
                (om-or2 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
                (om-or3 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
                ; contrary motion
                (b-pu-qd (gil::add-bool-var *sp* 0 1)) ; boolean p up and q down
                (b-pd-qu (gil::add-bool-var *sp* 0 1)) ; boolean p down and q up
                (cm-or1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
            )
                (gil::g-rel-reify *sp* p gil::IRT_LE 0 b-pd) ; b-pd = (p < 0)
                (gil::g-rel-reify *sp* p gil::IRT_EQ 0 b-ps) ; b-ps = (p == 0)
                (gil::g-rel-reify *sp* p gil::IRT_GR 0 b-pu) ; b-pu = (p > 0)
                (gil::g-rel-reify *sp* q gil::IRT_LE 0 b-qd) ; b-qd = (q < 0)
                (gil::g-rel-reify *sp* q gil::IRT_EQ 0 b-qs) ; b-qs = (q == 0)
                (gil::g-rel-reify *sp* q gil::IRT_GR 0 b-qu) ; b-qu = (q > 0)
                ; direct motion
                (gil::g-op *sp* b-pu gil::BOT_AND b-qu b-both-up) ; b-both-up = (b-pu and b-qu)
                (gil::g-op *sp* b-ps gil::BOT_AND b-qs b-both-stays) ; b-both-stays = (b-ps and b-qs)
                (gil::g-op *sp* b-pd gil::BOT_AND b-qd b-both-down) ; b-both-down = (b-pd and b-qd)
                (gil::g-op *sp* b-both-up gil::BOT_OR b-both-stays dm-or1) ; dm-or1 = (b-both-up or b-both-stays)
                (gil::g-op *sp* dm-or1 gil::BOT_OR b-both-down dm-or2) ; dm-or2 = (dm-or1 or b-both-down)
                (gil::g-rel-reify *sp* m gil::IRT_EQ DIRECT dm-or2) ; m = 1 if dm-or2
                (gil::g-rel-reify *sp* c gil::IRT_EQ *dir-motion-cost* dm-or2) ; add the cost of direct motion
                ; oblique motion
```

```
                (gil::g-op *sp* b-pu gil::BOT_AND b-qs b-pu-qs) ; b-pu-qs = (b-pu and b-qs)
                (gil::g-op *sp* b-pd gil::BOT_AND b-qs b-pd-qs) ; b-pd-qs = (b-pd and b-qs)
                (gil::g-op *sp* b-ps gil::BOT_AND b-qu b-ps-qu) ; b-ps-qu = (b-ps and b-qu)
                (gil::g-op *sp* b-ps gil::BOT_AND b-qd b-ps-qd) ; b-ps-qd = (b-ps and b-qd)
                (gil::g-op *sp* b-pu-qs gil::BOT_OR b-pd-qs om-or1) ; om-or1 = (b-pu-qs or
                    b-pd-qs)
                (gil::g-op *sp* om-or1 gil::BOT_OR b-ps-qu om-or2) ; om-or2 = (om-or1 or b-ps-qu
                    )
                (gil::g-op *sp* om-or2 gil::BOT_OR b-ps-qd om-or3) ; om-or3 = (om-or2 or b-ps-qd
                    )
                (gil::g-rel-reify *sp* m gil::IRT_EQ OBLIQUE om-or3) ; m = 0 if om-or3
                (gil::g-rel-reify *sp* c gil::IRT_EQ *obl-motion-cost* om-or3) ; add the cost of
                     oblique motion
                ; contrary motion
                (gil::g-op *sp* b-pu gil::BOT_AND b-qd b-pu-qd) ; b-pu-qd = (b-pu and b-qd)
                (gil::g-op *sp* b-pd gil::BOT_AND b-qu b-pd-qu) ; b-pd-qu = (b-pd and b-qu)
                (gil::g-op *sp* b-pu-qd gil::BOT_OR b-pd-qu cm-or1) ; cm-or1 = (b-pu-qd or
                    b-pd-qu)
                (gil::g-rel-reify *sp* m gil::IRT_EQ CONTRARY cm-or1) ; m = -1 if cm-or1
                (gil::g-rel-reify *sp* c gil::IRT_EQ *con-motion-cost* cm-or1) ; add the cost of
                     contrary motion
            )
    )
)

; create the motion list variable as it is perceived by the human ear,
; i.e. if the interval between the thesis and the arsis note is greater than a third,
; then the motion is perceived from the arsis note and not from the thesis note
; @m-intervals-ta: melodic intervals between the thesis and the arsis note
; @motions: motions perceived from the thesis note
; @motions-arsis: motions perceived from the arsis note
; @real-motions: motions perceived by the human ear
(defun create-real-motions (m-intervals-ta motions motions-arsis real-motions motions-costs
    motions-arsis-costs real-motions-costs)
    (loop
        for tai in m-intervals-ta
        for t-move in motions
        for a-move in motions-arsis
        for r-move in real-motions
        for t-c in motions-costs
        for a-c in motions-arsis-costs
        for r-c in real-motions-costs
        do
            (let (
                (b (gil::add-bool-var *sp* 0 1)) ; for (tai > 4)
            )
                (gil::g-rel-reify *sp* tai gil::IRT_GR 4 b) ; b = (tai > 4)
                (gil::g-ite *sp* b a-move t-move r-move) ; r-move = (b ? a-move : t-move)
                (gil::g-ite *sp* b a-c t-c r-c) ; r-c = (b ? a-c : t-c)
            )
    )
)

; add the constraint such that there is no perfect consonance in thesis that is reached by
    direct motion
(defun add-no-direct-move-to-p-cons-cst (motions is-p-cons-arr &optional (r t))
    (loop
        for m in motions
        for b in (rest-if is-p-cons-arr r)
        do
            (gil::g-rel-reify *sp* m gil::IRT_NQ DIRECT b gil::RM_IMP)
    )
)

; return the rest of the list if the boolean is true, else return the list
(defun rest-if (l b)
    (if b
```

```
        (rest l)
        l
    )
)

; TODO pass to new version function below
; add the constraint such that there is no battuta kind of motion, i.e.:
;   - contrary motion
;   - skip in the upper voice
;   - lead to an octave
(defun add-no-battuta-cst (motions h-intervals m-intervals-brut is-cf-bass-arr &optional (
    is-cst-arr nil))
    (loop
    for move in motions
    for hi in (rest h-intervals)
    for mi in m-intervals-brut
    for b in (butlast is-cf-bass-arr)
    for i from 0 below *cf-last-index
    do
        (let (
            (is-cm (gil::add-bool-var *sp* 0 1)) ; is contrary motion
            (is-oct (gil::add-bool-var *sp* 0 1)) ; is moving to octave
            (is-cp-down (gil::add-bool-var *sp* 0 1)) ; is counterpoint going down
            (b-and1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
            (b-and2 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
            (b-and3 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
        )
            (gil::g-rel-reify *sp* move gil::IRT_EQ CONTRARY is-cm) ; is-cm = (m == -1)
            (gil::g-rel-reify *sp* hi gil::IRT_EQ 0 is-oct) ; is-oct = (hi == 0)
            (gil::g-rel-reify *sp* mi gil::IRT_LE -4 is-cp-down) ; is-cp-down = (mi < -4)
            (gil::g-op *sp* is-cm gil::BOT_AND is-oct b-and1) ; b-and1 = (is-cm and is-oct)
            (gil::g-op *sp* b-and1 gil::BOT_AND is-cp-down b-and2) ; b-and2 = (b-and1 and
                is-cp-down)
            (if (null is-cst-arr)
                ; then constraint is always added
                (gil::g-op *sp* b-and2 gil::BOT_AND b 0) ; (is-cm and is-oct and is-cp-down and
                    b) = FALSE
                ; else constraint is added only if the current note is constrained
                (progn
                    (gil::g-op *sp* b-and2 gil::BOT_AND b b-and3) ; b-and3 = (b-and2 and b)
                    ; is-cst => (b-and3 == 0) can be written as not (is-cst and b-and3)
                    (gil::g-op *sp* (nth i is-cst-arr) gil::BOT_AND b-and3 0)
                )
            )
        )
    )
)

; TEST new version
; add the constraint such that there is no battuta kind of motion, i.e.:
;   - contrary motion
;   - skip in the upper voice
;   - lead to an octave
(defun add-no-battuta-bis-cst (motions h-intervals m-intervals-brut cf-brut-m-intervals
    is-cf-bass-arr &optional (is-cst-arr nil))
    (loop
    for move in motions
    for hi in (rest h-intervals)
    for mi in m-intervals-brut
    for cf-mi in cf-brut-m-intervals
    for b in (butlast is-cf-bass-arr)
    for i from 0 below *cf-last-index
    do
        (let (
            (is-cm (gil::add-bool-var *sp* 0 1)) ; is contrary motion
            (is-oct (gil::add-bool-var *sp* 0 1)) ; is moving to octave
```

```
            (is-cp-down (gil::add-bool-var *sp* 0 1)) ; is counterpoint going down more than 4 semi-tones
            (is-cf-down (gil::add-bool-var *sp* 0 1)) ; is cantus firmus going down more than 4 semi-tones
            (b-not (gil::add-bool-var *sp* 0 1)) ; !b = cantus firmus is not the bass
            (b-and1 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
            (b-and2 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
            (b-and3 (gil::add-bool-var *sp* 0 1)) ; temporary boolean
        )
            (gil::g-rel-reify *sp* move gil::IRT_EQ CONTRARY is-cm) ; is-cm = (m == 0)
            (gil::g-rel-reify *sp* hi gil::IRT_EQ 0 is-oct) ; is-oct = (hi == 0)
            (gil::g-rel-reify *sp* mi gil::IRT_LE -4 is-cp-down) ; is-cp-down = (mi < -4)
            (gil::g-rel-reify *sp* cf-mi gil::IRT_LE -4 is-cf-down) ; is-cf-down = (cf-mi < -4)
            (gil::g-op *sp* b gil::BOT_EQV FALSE b-not) ; b-not = !b
            (gil::g-op *sp* is-cm gil::BOT_AND is-oct b-and1) ; b-and1 = (is-cm and is-oct)
            (gil::g-op *sp* b gil::BOT_AND is-cp-down b-and2) ; b-and2 = (b-and1 and is-cp-down)
            (gil::g-op *sp* b-not gil::BOT_AND is-cf-down b-and3) ; b-and3 = (b-not and is-cf-down)

            (if (null is-cst-arr)
                ; then constraint is always added
                (progn
                    ; first case: (is-cm and is-oct and b and is-cp-down) = FALSE
                    (gil::g-op *sp* b-and1 gil::BOT_AND b-and2 0)
                    ; second case: (is-cm and is-oct and b-not and is-cf-down) = FALSE
                    (gil::g-op *sp* b-and1 gil::BOT_AND b-and3 0)
                )
                ; else constraint is added only if the current note is constrained
                (progn (let (
                    (b-and4 (gil::add-bool-var *sp* 0 1)) ; first case
                    (b-and5 (gil::add-bool-var *sp* 0 1)) ; second case
                )
                    (gil::g-op *sp* b-and1 gil::BOT_AND b-and2 b-and4) ; first case: b-and4 = (b-and1 and b-and2)
                    (gil::g-op *sp* b-and1 gil::BOT_AND b-and3 b-and5) ; second case: b-and5 = (b-and1 and b-and3)
                    ; is-cst => (b-and == 0) can be written as not (is-cst and b-and)
                    (gil::g-op *sp* (nth i is-cst-arr) gil::BOT_AND b-and4 0) ; first case
                    (gil::g-op *sp* (nth i is-cst-arr) gil::BOT_AND b-and5 0) ; second case
                ))
            )
        )
    )
)

;; 5th species methods
; add the constraint such that the selected notes are the same as the midi-selected notes
(defun add-selected-notes-cst (selected midi-selected cp)
    (print "Adding selected notes constraint")
    (print selected)
    (print midi-selected)
    (loop
    for i in selected
    for ms in midi-selected
    do
        (setq i+1 (+ i 1))
        (gil::g-rel *sp* (nth i cp) gil::IRT_EQ (first ms))
        (gil::g-rel *sp* (nth i+1 cp) gil::IRT_EQ (second ms))
    )
)

; add constraints such that the boolean array is true if the simple constraint is respected
(defun create-simple-boolean-arr (candidate-arr rel-type cst b-arr)
    (loop
        for c in candidate-arr
        for b in b-arr
        do
```

```
            (gil::g-rel-reify *sp* c rel-type cst b)
    )
)

; do the gil::g-ite constraint but only if and-cond is true
(defun and-ite (test then else var and-cond)
    (let (
        (b-and-then (gil::add-bool-var *sp* 0 1)) ; b-and-then = test and and-cond
        (test-not (gil::add-bool-var *sp* 0 1)) ; test-not = !test
        (b-and-else (gil::add-bool-var *sp* 0 1)) ; b-and-else = !test and and-cond
    )
        (gil::g-op *sp* test gil::BOT_AND and-cond b-and-then) ; b-and-then = test and and-cond
        (gil::g-op *sp* test gil::BOT_EQV FALSE test-not) ; test-not = !test
        (gil::g-op *sp* test-not gil::BOT_AND and-cond b-and-else) ; b-and-else = !test and and-cond
        (gil::g-rel-reify *sp* var gil::IRT_EQ then b-and-then gil::RM_IMP) ; b-and-then => var = then
        (gil::g-rel-reify *sp* var gil::IRT_EQ else b-and-else gil::RM_IMP) ; b-and-else => var = else
    )
)

; merge the boolean arrays with the and operator
(defun bot-merge-array (b-arr1 b-arr2 b-collect-arr &optional (bot gil::BOT_AND))
    (loop
    for b1 in b-arr1
    for b2 in b-arr2
    for b in b-collect-arr
    do
        (gil::g-op *sp* b1 bot b2 b)
    )
)

; merge the boolean arrays with the or operator and just return it
(defun collect-bot-array (b-arr1 b-arr2 &optional (bot gil::BOT_AND))
    (let (
        (b-collect-arr (gil::add-bool-var-array *sp* (length b-arr1) 0 1))
    )
        (loop
        for b1 in b-arr1
        for b2 in b-arr2
        for b in b-collect-arr
        do
            (gil::g-op *sp* b1 bot b2 b)
        )
        b-collect-arr
    )
)


(defun collect-t-or-f-array (yes-arr no-arr)
    (collect-bot-array
                yes-arr
                (collect-not-array no-arr)
                gil::BOT_OR
    )
)

(defun collect-not-array (arr)
    (collect-bot-array arr (gil::add-bool-var-array *sp* (length arr) 0 0) gil::BOT_EQV)
)

; do the gil::g-rel-reify constraint but use the condition that (b AND and-cond) is true
(defun bot-reify (var rel-type cst b and-cond &optional (bot gil::BOT_AND) (mode gil::RM_EQV))
    (let (
        (b-and (gil::add-bool-var *sp* 0 1)) ; b-and = b and and-cond
    )
```

```
        (gil::g-op *sp* b bot and-cond b-and) ; b-and = b and and-cond
        (gil::g-rel-reify *sp* var rel-type cst b-and mode) ; b-and == var rel-type cst
    )
)

; return the index of a note as all the notes are in a row,
; i.e. return the total index of the note at the given measure at the given beat assuming that
    we are in 4 4 time
; the index is 0-based, same for measure and beat
(defun total-index (measure beat)
    (+ (* measure 4) beat)
)

; is-mostly-3rd is true if second, third and fourth notes are from 3rd species
; note that is-mostly-3rd-arr have a length 4 times shorter than is-3rd-species-arr
(defun create-is-mostly-3rd-arr (is-3rd-species-arr is-mostly-3rd-arr)
    (loop
    for meas from 0 below (length is-mostly-3rd-arr)
    do
        (let (
            (b-23 (gil::add-bool-var *sp* 0 1)) ; b-23 = is-3rd-species-arr[meas][1] AND
                is-3rd-species-arr[meas][2]
        )
            ; b-23
            (gil::g-op *sp* (nth (total-index meas 1) is-3rd-species-arr) gil::BOT_AND (nth (
                total-index meas 2) is-3rd-species-arr) b-23)
            ; b-23 and "b-4" are stocked in is-mostly-3rd-arr[meas]
            (gil::g-op *sp* b-23 gil::BOT_AND (nth (total-index meas 3) is-3rd-species-arr) (nth
                 meas is-mostly-3rd-arr))
        )
    )
)

; collect elements all the 4 elements of the array, i.e. n, n+4, n+8, n+12, etc.
; note: n is the offset
(defun collect-by-4 (arr &optional (offset 0) (b nil) (up-bound 4))
    (setq len (if (eq offset 0) *cf-len *cf-last-index))
    (if (null b)
        ; then make a boolean array
        (setq ret (gil::add-bool-var-array *sp* len 0 1))
        ; else make a integer array
        (setq ret (gil::add-int-var-array *sp* len 0 up-bound))
    )
    (loop
    for i from offset below (length arr) by 4
    for j from 0 below len
    do
        (gil::g-rel *sp* (nth i arr) gil::IRT_EQ (nth j ret))
    )
    ret
)

; create an array for one beat from the entire array
(defun create-by-4 (arr-from arr-to &optional (offset 0))
    (loop
    for i from offset below (length arr-from) by 4
    for j in arr-to
    do
        (gil::g-rel *sp* (nth i arr-from) gil::IRT_EQ j)
    )
)

; add a reify constraint if @b is not nil, else add a rel constraint
(defun rel-reify-if (var rel-type cst &optional (b nil) (rm gil::RM_IMP))
    (if (null b)
        (gil::g-rel *sp* var rel-type cst)
        (gil::g-rel-reify *sp* var rel-type cst b rm)
```

```
    )
)

; return BoolVar true if nil element
(defun true-if-null (arr i)
    (if (null arr)
        ; then
        TRUE
        ; else
        (nth i arr)
    )
)

; add the constraint such that if sp3 is 4th species, then sp4 is 0 and the next sp1 is 4th
    species
; and vice versa (cannot have 4th species in first position without 4th species in third
    position)
; - sp-arr3: array of IntVar for species at the third position
; - sp-arr4: array of IntVar for species at the fourth position
; - sp-arr1: array of IntVar for species at the first position
(defun add-4th-rythmic-cst (sp-arr3 sp-arr4 sp-arr1)
    (loop
    for sp3 in sp-arr3
    for sp4 in sp-arr4
    for sp1 in (rest sp-arr1)
    do
        (let (
            (b-34 (gil::add-bool-var *sp* 0 1)) ; b-34 = sp3 == 4th species
            (b-14 (gil::add-bool-var *sp* 0 1)) ; b-14 = sp1 == 4th species
        )
            (gil::g-rel-reify *sp* sp3 gil::IRT_EQ 4 b-34) ; b-34 = sp3 == 4th species
            (gil::g-rel-reify *sp* sp1 gil::IRT_EQ 4 b-14) ; b-14 = sp1 == 4th species
            (gil::g-rel-reify *sp* sp4 gil::IRT_EQ 0 b-34 gil::RM_IMP) ; b-34 => sp4 == 0
            (gil::g-op *sp* b-34 gil::BOT_EQV b-14 1) ; b-34 <=> b-14
        )
    )
)

; add the constraint such that if n belongs to @species, then n+m have to exist (not 0)
; by default, the constraint is added for the third species
; - species-arr: array of IntVar for species
; - spec: species to check
; - offset: offset to check
(defun add-no-silence-cst (species-arr &key (spec 3) (offset 1))
    (loop
    for n in species-arr
    for n+m in (nthcdr offset species-arr)
    do
        (let (
            (b (gil::add-bool-var *sp* 0 1)) ; b = (n == species)
        )
            (gil::g-rel-reify *sp* n gil::IRT_EQ spec b) ; b = (n == spec)
            (gil::g-rel-reify *sp* n+m gil::IRT_NQ 0 b gil::RM_IMP) ; b => (n+m != 0)
        )
    )
)

; add the constraint such that there is maximum 2 consecutive measures without 4th species
(defun add-min-syncope-cst (third-sp-arr)
    (loop
    for sp1 in (nthcdr 1 third-sp-arr)
    for sp2 in (nthcdr 2 third-sp-arr)
    for sp3 in (nthcdr 3 third-sp-arr)
    do
        (let (
            (b1-not-4 (gil::add-bool-var *sp* 0 1)) ; b1-not-4 = sp1 != 4
            (b2-not-4 (gil::add-bool-var *sp* 0 1)) ; b2-not-4 = sp2 != 4
```

```lisp
      (b-and (gil::add-bool-var *sp* 0 1)) ; b-and = b1-not-4 && b2-not-4
    )
      (gil::g-rel-reify *sp* sp1 gil::IRT_NQ 4 b1-not-4) ; b1-not-4 = sp1 != 4
      (gil::g-rel-reify *sp* sp2 gil::IRT_NQ 4 b2-not-4) ; b2-not-4 = sp2 != 4
      (gil::g-op *sp* b1-not-4 gil::BOT_AND b2-not-4 b-and) ; b-and = b1-not-4 && b2-not-4
      (gil::g-rel-reify *sp* sp3 gil::IRT_EQ 4 b-and gil::RM_IMP) ; b-and => sp3 == 4
    )
  )
)

; add all constraints to create a rythmic and select what species to use
; mandatory rules are:
; - 4th species is only used in third and first position
; - 4th species in third position is followed by a 0 (no note/constraint) and then a 4th species
; - no 3rd species followed by 0
; classic rules are:
; - first and penultimate measure are 4th species
; - only 3rd and 4th species are used
; - 3rd species should represent at least 1/3 of the notes
; - 4th species should represent at least 1/4 of the notes
(defun create-species-arr (species-arr &key (min-3rd-pc (* (- 1 (getparam 'pref-species-slider)) 0.66)) (min-4th-pc (* (getparam 'pref-species-slider) 0.5)))
  (print "Create species array...")
  (let* (
    (count-3rd (gil::add-int-var-array *sp* *total-cp-len 0 1))
    (count-4th (gil::add-int-var-array *sp* *total-cp-len 0 1))
    (n-3rd-int (floor (* *total-cp-len min-3rd-pc))) ; minimum number of 3rd species
    (n-4th-int (floor (* *total-cp-len min-4th-pc))) ; minimum number of 4th species
    (sum-3rd (gil::add-int-var *sp* n-3rd-int *total-cp-len)) ; set the bounds of sum-3rd
    (sum-4th (gil::add-int-var *sp* n-4th-int *total-cp-len)) ; set the bounds of sum-4th
  )
    (setq *sp-arr (list
      (collect-by-4 species-arr 0 t)
      (collect-by-4 species-arr 1 t)
      (collect-by-4 species-arr 2 t)
      (collect-by-4 species-arr 3 t)
    ))

    (print "Counting 3rd and 4th species...")
    ; count the number of 3rd and 4th species
    (add-cost-cst species-arr gil::IRT_EQ 3 count-3rd)
    (add-cost-cst species-arr gil::IRT_EQ 4 count-4th)
    ; sum the number of 3rd and 4th species
    (gil::g-sum *sp* sum-3rd count-3rd)
    (gil::g-sum *sp* sum-4th count-4th)

    ; 4th species is only used in third and first position
    (gil::g-rel *sp* (second *sp-arr) gil::IRT_NQ 4) ; second position not 4th species
    (gil::g-rel *sp* (fourth *sp-arr) gil::IRT_NQ 4) ; fourth position not 4th species

    ; 4th species in third position is followed by a 0 (no note/constraint) and then a 4th species
    (add-4th-rythmic-cst (third *sp-arr) (fourth *sp-arr) (first *sp-arr))

    ; only 3rd and 4th species are used
    (gil::g-rel *sp* species-arr gil::IRT_NQ 1) ; not 1st species
    (gil::g-rel *sp* species-arr gil::IRT_NQ 2) ; not 2nd species

    ; first and penultimate measure are 4th species
    ; first measure = [0 0 4 0]
    (gil::g-rel *sp* (first (first *sp-arr)) gil::IRT_EQ 0) ; first note is silent
    (gil::g-rel *sp* (first (second *sp-arr)) gil::IRT_EQ 0) ; second note is silent
    (gil::g-rel *sp* (first (third *sp-arr)) gil::IRT_EQ 4) ; third note is 4th species
    ; penultimate measure = [4 0 4 0]
    (gil::g-rel *sp* (penult (first *sp-arr)) gil::IRT_EQ 4) ; first note is 4th species
    (gil::g-rel *sp* (lastone (second *sp-arr)) gil::IRT_EQ 0) ; second note does not exist
    (gil::g-rel *sp* (lastone (third *sp-arr)) gil::IRT_EQ 4) ; third note is 4th species
```

```
        ; no silence after 3rd species notes
        (add-no-silence-cst species-arr)

        ; no silence after 4th species notes in n+4 position
        (add-no-silence-cst species-arr :spec 4 :offset 4)

        ; maximum two consecutive measures without 4th species
        (add-min-syncope-cst (third *sp-arr))
    )
)

; add constraints such that the non-constrained notes have only one possible value
(defun add-one-possible-value-cst (cp is-not-cst-arr)
    (loop
    for p in cp
    for p+1 in (nthcdr 1 cp)
    for b-not-cst in is-not-cst-arr
    do
        (gil::g-rel-reify *sp* p gil::IRT_EQ p+1 b-not-cst gil::RM_IMP) ; TODO the value of the
            note
    )
)

; add constraints such that consecutives syncopations cannot be the same
; depending on @is-syncope-arr which is true if the note is a syncopation
(defun add-no-same-syncopation-cst (cp-thesis cp-arsis is-syncope-arr)
    (loop
    for th in (rest cp-thesis)
    for ar in (rest cp-arsis)
    for b in (rest is-syncope-arr)
    do
        (gil::g-rel-reify *sp* th gil::IRT_NQ ar b gil::RM_IMP)
    )
)

; find the next @type note in the borrowed scale,
; if there is no note in the range then return the note of the other @type
; - note: integer for the current note
; - type: atom [lower | higher] for the type of note to find
; note: this function has noting to do with GECODE
(defun find-next-note (note type)
    (let (
        ; first sort the scale corresponding to the type
        (sorted-scale (if (eq type 'lower)
            (sort *extended-cp-domain #'>)
            (sort *extended-cp-domain #'<)
        ))
    )
        (if (eq type 'lower)
            ; then we search the first note in the sorted scale that is lower than the current
                note
            (progn
                (loop for n in sorted-scale do
                    (if (< n note) (return-from find-next-note n))
                )
                ; no note so we return the penultimate element of the sorted scale
                (penult sorted-scale)
            )
            ; else we search the first note in the sorted scale that is higher than the current
                note
            (progn
                (loop for n in sorted-scale do
                    (if (> n note) (return-from find-next-note n))
                )
                ; no note so we return the penultimate element of the sorted scale
                (penult sorted-scale)
```

```
            )
        )
    )
)

; parse the species array to get the corresponding rythmic pattern for open music
; - species-arr: array of integer for species (returned by the next-solution algorithm)
; - cp-arr: array of integer for counterpoint notes (returned by the next-solution algorithm)
; note: this function has noting to do with GECODE
(defun parse-species-to-om-rythmic (species-arr cp-arr)
    ; replace the last element of the species array by 1
    (setf (first (last species-arr)) 1)
    (build-rythmic-pattern species-arr cp-arr)
)

; build the rythmic pattern for open music from the species array
; - species-arr: array of integer for species
; - cp-arr: array of integer for counterpoint notes
; - rythmic-arr: array of integer for the rythmic (supposed to be nil and then filled by the recursive function)
; - notes-arr: array of interger for notes (supposed to be nil and then filled by the recursive function)
; - b-debug: boolean to print debug info
; note: this function has noting to do with GECODE
(defun build-rythmic-pattern (species-arr cp-arr &optional (rythmic-arr nil) (notes-arr nil) (b-debug nil))
    ; print debug info
    (if b-debug
        (progn
        (print "Current species and notes:")
        (print species-arr)
        (print cp-arr)
        (print "Current answer:")
        (print rythmic-arr)
        (print notes-arr)
        )
    )
    ; base case
    (if (null species-arr)
        ; then return the rythmic pattern
        (list rythmic-arr notes-arr)
    )

    (let (
        (sn (first species-arr)) ; current species
        (sn+1 (second species-arr)) ; next species
        (sn+2 (third species-arr)) ; next next species
        (sn+3 (fourth species-arr)) ; next next next species
        (cn (first cp-arr)) ; current counterpoint note
        (cn+1 (second cp-arr)) ; next counterpoint note
        (cn+2 (third cp-arr)) ; next next counterpoint note
        (cn+3 (fourth cp-arr)) ; next next next counterpoint note
    )
        ; replace all nil by -1 for the species
        (if (null sn) (setf sn -1))
        (if (null sn+1) (setf sn+1 -1))
        (if (null sn+2) (setf sn+2 -1))
        (if (null sn+3) (setf sn+3 -1))
        ; replace all nil by -1 for the counterpoint
        (if (null cn) (setf cn -1))
        (if (null cn+1) (setf cn+1 -1))
        (if (null cn+2) (setf cn+2 -1))
        (if (null cn+3) (setf cn+3 -1))

        (if b-debug
            (progn
            (print (format nil "sn: ~a, sn+1: ~a, sn+2: ~a, sn+3: ~a" sn sn+1 sn+2 sn+3))
```

```
            (print (format nil "cn: ~a, cn+1: ~a, cn+2: ~a, cn+3: ~a" cn cn+1 cn+2 cn+3))
            )
        )

        (cond
            ; 1 if it is the last note [1 -1 ...]
            ((and (eq sn 1) (eq sn+1 -1))
                (list (append rythmic-arr (list 1)) (append notes-arr (list cn)))
            )

            ; if [4 0 4 ...] -> which syncope ?
            ((and (eq sn 4) (eq sn+1 0) (eq sn+2 4))
            (if (/= cn cn+2) ; syncopation but different notes ?
                ; then same as half note
                (if (eq sn+3 3)
                    ; then 1/2 + 1/4 if [4 0 4 3] (syncopation catch up by a quarter note)
                    (build-rythmic-pattern
                        (nthcdr 3 species-arr)
                        (nthcdr 3 cp-arr)
                        (append rythmic-arr (list 1/2 1/4))
                        (append notes-arr (list cn cn+2))
                    )
                    ; else 1/2 + 1/2 if [4 0 4 0] (basic syncopation)
                    (build-rythmic-pattern
                        (nthcdr 4 species-arr)
                        (nthcdr 4 cp-arr)
                        (append rythmic-arr (list 1/2 1/2))
                        (append notes-arr (list cn cn+2))
                    )
                )
                ; else same as full note syncopated
                (if (eq sn+3 3)
                    ; then 3/4 if [4 0 4 3] (syncopation catch up by a quarter note)
                    (build-rythmic-pattern
                        (nthcdr 3 species-arr)
                        (nthcdr 3 cp-arr)
                        (append rythmic-arr (list 3/4))
                        (append notes-arr (list cn))
                    )
                    ; else 1 if [4 0 4 0] (basic syncopation)
                    (build-rythmic-pattern
                        (nthcdr 4 species-arr)
                        (nthcdr 4 cp-arr)
                        (append rythmic-arr (list 1))
                        (append notes-arr (list cn))
                    )
                )
            ))

            ; 1/8 note (croche) if cn == cn+1 AND [!0 (3 or 4) ...]
            ((and (eq cn cn+1) (/= sn 0) (or (eq sn+1 3) (eq sn+1 4)))
                (if (>= (lastone notes-arr) cn)
                    ; then eighth note with the next lower note
                    (build-rythmic-pattern
                        (nthcdr 1 species-arr)
                        (nthcdr 1 cp-arr)
                        (append rythmic-arr (list 1/8 1/8))
                        (append notes-arr (list cn (find-next-note cn 'lower)))
                    )
                    ; else eighth note with the next higher note
                    (build-rythmic-pattern
                        (nthcdr 1 species-arr)
                        (nthcdr 1 cp-arr)
                        (append rythmic-arr (list 1/8 1/8))
                        (append notes-arr (list cn (find-next-note cn 'higher)))
                    )
                )
```

```
            )

            ; silence if [0 0 ...]
            ((and (eq sn 0) (eq sn+1 0))
                (build-rythmic-pattern
                    (nthcdr 2 species-arr)
                    (nthcdr 2 cp-arr)
                    (append rythmic-arr (list -1/2))
                    notes-arr
                )
            )

            ; 1 if [1 0 0 0] (full note)
            ((and (eq sn 1) (eq sn+1 0) (eq sn+2 0) (eq sn+3 0))
                (build-rythmic-pattern
                    (nthcdr 4 species-arr)
                    (nthcdr 4 cp-arr)
                    (append rythmic-arr (list 1))
                    (append notes-arr (list cn))
                )
            )

            ; 1/2 if [2 0 ...] (half note)
            ((and (eq sn 2) (eq sn+1 0))
                (build-rythmic-pattern
                    (nthcdr 2 species-arr)
                    (nthcdr 2 cp-arr)
                    (append rythmic-arr (list 1/2))
                    (append notes-arr (list cn))
                )
            )

            ; 1/4 if [3 ...] (quarter note)
            ((eq sn 3)
                (build-rythmic-pattern
                    (nthcdr 1 species-arr)
                    (nthcdr 1 cp-arr)
                    (append rythmic-arr (list 1/4))
                    (append notes-arr (list cn))
                )
            )

            ; 1/2 if [4 0 1 ...] (penultimate note for the 4th species)
            ((and (eq sn 4) (eq sn+1 0) (eq sn+2 1))
                (build-rythmic-pattern
                    (nthcdr 2 species-arr)
                    (nthcdr 2 cp-arr)
                    (append rythmic-arr (list 1/2))
                    (append notes-arr (list cn))
                )
            )
        )
    )
)

; get the basic rythmic pattern for a given species
; - species: the species [1 2 3 4]
; - len: the length of the counterpoint
; examples:
; (1 5) -> (1 1 1 1 1)
; (2 5) -> (1/2 1/2 1/2 1/2 1/2 1/2 1/2 1/2 1)
; (3 5) -> (1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1)
; (4 5) -> ~(-1/2 1 1 1 1/2 1/2 1) depending on the counterpoint
(defun get-basic-rythmic (species len &optional (cp nil))
    (setq len-1 (- len 1))
    (setq len-2 (- len 2))
    (setq cp-len (+ (* 4 len-1) 1))
```

```
    (case species
        (1 (make-list len :initial-element 1))
        (2 (append (make-list (* 2 len-1) :initial-element 1/2) '(1)))
        (3 (append (make-list (* 4 len-1) :initial-element 1/4) '(1)))
        (4 (build-rythmic-pattern
                (get-4th-species-array len-2)
                (get-4th-notes-array cp cp-len)
        ))
    )
)

; return a species array for a 4th species counterpoint
; - len-2: the length of the counterpoint - 2
(defun get-4th-species-array (len-2)
    (append (list 0 0) (get-n-4040 len-2) (list 4 0 1))
)

; return a note array for a 4th species counterpoint
; - len: the length of the cantus firmus
(defun get-4th-notes-array (cp len)
    (let* (
        (notes (make-list len :initial-element 0)) ; notes that we don't care about can be 0
    )
        (loop
        for n from 2 below len by 2 ; we move from 4 to 4 (4 0 4 ...) after the silence (0 0) at
             the start
        for p in cp
        do
            (setf (nth n notes) p)
        )
        notes
    )
)

; return a list with n * (4 0 4 0), used to build the rythmic pattern for the 4th species
; - n: the number of times the pattern is repeated
(defun get-n-4040 (n)
    (if (eq n 0)
        nil
        (append (list 4 0 4 0) (get-n-4040 (- n 1)))
    )
)

; return the tone offset of the voice
; => [0, ...,  11]
; 0 = C, 1 = C#, 2 = D, 3 = D#, 4 = E, 5 = F, 6 = F#, 7 = G, 8 = G#, 9 = A, 10 = A#, 11 = B
(defun get-tone-offset (voice)
    (let (
        (tone (om::tonalite voice))
    )
        (if (eq tone nil)
            ; then default to C major
            0
            ; else check if the mode is major or minor
            (let (
                (mode (om::mode tone))
            )
                (if (eq (third mode) 300)
                    (midicent-to-midi-offset (+ (om::tonmidi tone) 300))
                    (midicent-to-midi-offset (om::tonmidi tone))
                )
            )
        )
    )
)

; converts a midicent value to the corresponding offset midi value
```

```lisp
; note:[0, 12700] -> [0, 11]
; 0 corresponds to C, 11 to B
(defun midicent-to-midi-offset (note)
    (print (list "midicent-to-midi-offset..." note))
    (mod (/ note 100) 12)
)

; return the absolute difference between two midi notes modulo 12
; or the brut interval if b is true
(defun inter (n1 n2 &optional (b nil))
    (if b
        (- n1 n2)
        (mod (abs (- n1 n2)) 12)
    )
)

; add constraint in sp such that the interval between the two notes is a member of interval-set
(defun inter-member-cst (sp n1-var n2-val interval-set)
    (let (
        (t1 (gil::add-int-var-expr sp n1-var gil::IOP_SUB n2-val)) ; t1 = n1 - n2
        (t2 (gil::add-int-var sp 0 127)) ; used to store the absolute value of t1
        note-inter
    )
        (gil::g-abs sp t1 t2) ; t2 = |t1|
        (setq note-inter (gil::add-int-var-expr sp t1 gil::IOP_MOD 12)) ; note-inter = t1 % 12
        (gil::g-member sp interval-set note-inter) ; note-inter in interval-set
    )
)

; add constraint such that n3-var = |n1-var - n2-val| % 12
(defun inter-eq-cst (sp n1-var n2-val n3-var)
    (let (
        (t1 (gil::add-int-var-expr sp n1-var gil::IOP_SUB n2-val)) ; t1 = n1 - n2
        (t2 (gil::add-int-var sp 0 127)) ; used to store the absolute value of t1
        (modulo (gil::add-int-var-dom sp '(12))) ; the IntVar just used to store 12
    )
        (gil::g-abs sp t1 t2) ; t2 = |t1|
        (gil::g-mod sp t2 modulo n3-var) ; n3-var = t2 % 12
    )
)

; add constraint such that
; brut-var = n1-var - n2
; abs-var = |brut-var|
(defun inter-eq-cst-brut (sp n1-var n2 brut-var abs-var)
    (let (
        (t1 (gil::add-int-var-expr sp n1-var gil::IOP_SUB n2)) ; t1 = n1-var - n2
    )
        (gil::g-rel sp t1 gil::IRT_EQ brut-var) ; t1 = brut-var
        (gil::g-abs sp t1 abs-var) ; abs-var = |t1|
    )
)

; add constraint such that
; brut-var = n1-var - n2
; abs-var = |brut-var|
(defun inter-eq-cst-brut-for-cst (sp n1-var n2 brut-var abs-var is-cst)
    (let (
        (t1 (gil::add-int-var-expr sp n1-var gil::IOP_SUB n2)) ; t1 = n1-var - n2
        (t2 (gil::add-int-var sp 0 12)) ; store the absolute value of t1
    )
        (gil::g-abs sp t1 t2) ; t2 = |t1|
        (gil::g-ite sp is-cst t1 ZERO brut-var) ; brut-var = t1 if is-cst, else brut-var = 0
        (gil::g-ite sp is-cst t2 ZERO abs-var) ; abs-var = t2 if is-cst, else abs-var = 0
    )
)
```

```
; return the last element of a list
(defun lastone (l)
    (first (last l))
)

; return the rest of a list without its last element
(defun restbutlast (l)
    (butlast (rest l))
)

; return the penultimate element of a list
(defun penult (l)
    (nth (- (length l) 2) l)
)

; return an approximative checksum of pitches associated to a rythmic
; - p: the list of pitches
; - r: the list of rythmic values (with the -1/2 at the beginning)
(defun checksum-sol (p r)
    (let (
        (l (length p))
    )
        (mod (floor (reduce #'+
            (mapcar #'* (range (+ l 5) :min 5) (rest r) p)))
        (expt l 12))
    )
)

; add a the sum of the @factor-arr as a cost to the *cost-factors array and increment *
    n-cost-added
(defun add-cost-to-factors (factor-arr)
    (gil::g-sum *sp* (nth *n-cost-added *cost-factors) factor-arr)
    (incf *n-cost-added)
)
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**