UCLouvain

epl

École polytechnique de Louvain

FuxCP : A Validated Constraint Programming Formalization of Fux Four-Voice Counterpoint

Author: **Tom Lai**
Supervisor: **Peter Van Roy**
Readers: **Karim Haddad, Hélène Verhaeghe**
Academic year 2024–2025
Master [120] in Computer Science

Abstract

This thesis takes over the constraint-programming formalization of Fux's musical theory developed in prior work and consolidates, clarifies, and validates the rule system for Fuxian counterpoint. The model distinguishes hard rules (stylistic necessities) from soft preferences (graded tendencies) and adopts a figure-driven validation methodology based on Minimal Unsatisfiable Subsets (MUS) to diagnose and resolve tensions between the formalism, Fux's examples, and auxiliary sources. Validation proceeds along four complementary axes: (i) consistency with implicit theoretical knowledge and practical pedagogy; (ii) verification against worked examples from Gradus ad Parnassum using MUS to isolate minimal conflicts; (iii) cross-checking with the explicit formulations in the Traité de Contrepoint by Noël Gallon and Marcel Bitsch; and (iv) systematic correction of formulation inconsistencies and implementation errors inherited from previous iterations.

On the engineering side, the thesis extends and corrects the Gecode-based solver, introduces a modular activation/deactivation mechanism that enables controlled experimentation with subsets of rules, and integrates MUS diagnostics directly into the testing pipeline. Several rules are re-cast as soft constraints to encode stylistic latitude without sacrificing correctness. A unified, documented constraint library and reproducible test harness (including species/voice-count figure tests) provide a coherent, maintainable framework for future research and extensions.

Together, these contributions deliver a clearer and more faithful executable theory of species counterpoint: a consolidated catalogue of general and species-specific constraints with explicit domains of application; an implementation that explains failures at rule level via MUS; and a validated workflow that reproduces Fux's figures where supported and pinpoints precisely where the model must be refined or extended.

Acknowledgments

I am deeply grateful to my supervisor, **Peter Van Roy**, for his unwavering enthusiasm for this project, his availability, and his guidance that consistently steered me in the right direction. Thanks to him, I had the opportunity to meet **Hélène Verhaeghe** and **Karim Haddad**.

My sincere thanks go to **Hélène Verhaeghe** (École polytechnique de Louvain) for taking the time to discuss the project with us and for her crucial advice to employ *Minimal Unsatisfiable Subsets (MUSes)* to test the program against Fux's examples.

I also extend my gratitude to **Karim Haddad**, Doctor in Musicology and specialist in counterpoint, for sharing his expertise, for the many meetings we held, and for his remarkably swift and thoughtful email responses.

I warmly thank the previous contributors to this project—**Damien Sprockeels**, **Thibault Wafflard**, **Anton Lamotte**, **Luc Cleenewerk**, and **Diego de Patoul**—for the quality and rigor of their work, which laid the foundations for the present thesis.

Finally, I am indebted to my **family and friends** for their unfailing support and encouragement throughout this journey.

Contents

Chapter 1

Introduction

1.1 State of the Art in Musical Automation

From the earliest days of computational music, computer-assisted composition has balanced the intuitive with the formal rule. On one hand, machine learning embraces the organic flux of musical expression; on the other, constraint programming offers the precision of codified tradition. Together, these two paradigms define the breadth of contemporary musical automation—from impressionistic generative systems to rigorously rule-based solvers.

Machine Learning and Music

In the realm of machine learning, music becomes data—melodies, harmonies, and expressive timings encoded as patterns to be learned, mimicked, and transformed. Deep learning models now address composition across multiple levels: score generation, performance nuances, and even audio synthesis. A comprehensive survey highlights how these tiers are interwoven through evolving architectures, diverse datasets, and evaluation methods—and also shines a light on the persistent challenges of creativity, structure, and control.[3]

Projects like AIVA, an acclaimed AI composer trained on the masters of Western art music, illustrate how deep architectures and reinforcement learning yield expressive, stylistically rich compositions.[1]

Other systems—from Google's Magenta[11] (which powers interactive tools like AI Duet[10]) to Sony's Flow Machines[16]—highlight both the playful and practical potential of AI in composing music, while prompting ongoing reflection on originality and collaboration.

Yet the poetry of machine-generated music is not without unease. Critics caution that such systems often yield stereotyped patterns, constrained by their training data, and may lack the capacity for expressive deviation or meaningful invention. Deep learning models also struggle with direct user control over tonality, form, or expressive intent—a limitation particularly relevant in counterpoint, where rule shaping is paramount.

Constraint Programming and Music

In contrast, constraint programming (CP) offers a more disciplined artistry—a way of embedding music theory as solvable logic. You define the rules; the computer ensures compliance. The appeal lies in encoding precise stylistic norms leveraging declarative constraints and powerful solver frameworks.

Indeed, counterpoint—especially in the style of Fux—lends itself well to such formalism. Its strict rules, once translated into constraints, can be solved efficiently to generate valid and stylistically faithful musical lines. Early work like Ebcioglu's florid counterpoint system enumerated dozens of constraints to produce conservatory-level examples.[6]

Schottstaedt later modeled all five species of Fuxian counterpoint for up to six voices, even adding "penalty" values to compensate rules flexibility.[14]

Other musicians and researchers explored polyphonic textures ranging from atonal to Ligeti-like structures, further confirming the adaptability of CP in composition.

However, constraint-based approaches have their own limitations. They require exhaustive rules formalization, often sacrificing nuance or expressive exceptions that a human composer might apply "by ear." Moreover, such systems may lack fluid interactivity or the capacity to learn new stylistic variations.

1.2 FuxCP Context

FuxCP is a computer-aided composition tool that formalizes the counterpoint pedagogy of Johann Joseph Fux's *Gradus ad Parnassum* as a set of mathematical rules and solves them with a constraint solver to produce musically valid counterpoint. It was initiated as a collaboration between UCLouvain and IRCAM and conceived to assist composers—especially those without programming expertise—by turning species-specific rules into constraints the computer can enforce and explore systematically. In its most recent form, FuxCP comprises a rigorously specified rule base and an implementation over the Gecode[15] solver that can generate contrapuntal lines consistent with the relevant species and stylistic preferences.

Constraint programming (CP) is an appropriate foundation for this project because it offers transparency and control: rules are explicit, traceable constraints; preferences can be encoded as soft constraints or costs; and solutions can be inspected and steered rather than inferred opaquely from data. Prior work in FuxCP repeatedly emphasized these advantages over black-box generative approaches, arguing that CP "understands" and enforces the theory while still leaving room to shape musical outcomes. In practice, the framework separates hard stylistic rules from adjustable preferences, which is critical for balancing correctness with musical nuance.

Historically, the project grew alongside IRCAM's OpenMusic[12]. OpenMusic (OM) is IRCAM's visual programming environment for computer-assisted composition, built on Common Lisp. On this foundation, the project evolved through four theses. In 2022, Damien Sprockeels developed Melodizer, estab-

lishing the GiL bridge between OM (Lisp) and Gecode[15] (C++) and demonstrating interactive constraint-based composition in which mandatory rules can be combined with optional preferences[17]. In 2023, Thibault Wafflard introduced FuxCP proper for two-voice species, mapping Fux's rules to constraints and exposing them in OM for composer-facing use[18]. In 2024, Anton Lamotte generalized the model to three voices, formalizing interactions with the lowest voice and introducing strata while clarifying the role of soft constraints to preserve stylistic nuance[13]. Later in 2024, Luc Cleenewerk and Diego de Patoul unified the accumulated rule sets, extended FuxCP to four voices, and migrated the implementation to an object-oriented C++ codebase on Gecode[5].

OpenMusic is no longer used in the current line of work. FuxCP is now maintained as a standalone C++ library with a stable API, and a native C++ graphical interface for the project is being developed in a separate master's thesis. This decoupling keeps the solver transparent and testable while preparing the project for broader integration.

1.3 Contribution

This section presents the main outcomes of the work and how they advance FuxCP from both a theoretical and an engineering standpoint. On the conceptual side, the thesis consolidates, clarifies, and validates the formal rule system for Fuxian counterpoint—separating hard rules from soft stylistic preferences and using MUS-based analysis to resolve tensions between the model, Fux's examples, and auxiliary sources. On the implementation side, it extends and corrects the Gecode-based solver, introduces modular mechanisms for activating subsets of constraints and MUS diagnostics, and unifies the rule library and documentation into a coherent, testable framework. The contributions are summarised below under two complementary headings: conceptual and implementation.

Conceptual contributions

- Validation and correction of the formal model of Fuxian counterpoint
  - Validation of the mathematical definitions of the rules and their formulation as constraints.
  - Validation performed in four complementary directions:
    * Comparison with implicit theoretical knowledge of counterpoint practice.
    * Verification against musical examples in *Gradus ad Parnassum* by Johann Joseph Fux[8] [4], using Minimal Unsatisfiable Subset (MUS) analysis.
    * Cross-checking with the 20th-century treatise of Noël Gallon and Marcel Bitsch[9].

    - ∗ Identification and correction of formulation errors and inconsistencies in previous work.

- Refinement of the theoretical framework by distinguishing between rules, preferences, and stylistic constraints, thus providing a more flexible interpretation of the counterpoint tradition.
- Introduction of MUS techniques into the analysis of musical rules, demonstrating their applicability to music theory as a method for detecting conflicts and implicit assumptions.

Implementation contributions

- Extension and correction of the constraint-based model in Gecode[15]
    - Correction of erroneous rule implementations inherited from previous versions.
    - Transformation of certain rules into soft constraints or preferences, allowing for a graded handling of stylistic deviations.
- Integration of MUS analysis into the solver framework, enabling the systematic detection of inconsistent rule sets in practical musical examples.
- Development of a modular activation/deactivation mechanism for constraints, allowing controlled experimentation with subsets of rules and precise reproduction of Fux's musical figures.
- Testing and validation through computational experiments, reproducing Fux's examples (when compatible with the solver's scope) and identifying cases where the model had to be refined or extended.
- Documentation and unification of the constraint library, providing a coherent basis for future research and extensions of the system.

Roadmap

- **Chapter 2 — Background.** Fixes the terminology and scope that the formal model presupposes, positions the work within the historical and theoretical lineage of species counterpoint and outlines the roles of Fux and of Gallon–Bitsch.
- **Chapter 3 — Concepts, Variables, and Data Structures.** Introduces the computational abstractions (voices, parts, strata) and the core arrays used throughout the model, together with the notational conventions and preference machinery that support subsequent formalization.
- **Chapter 4 — Program Architecture.** Details the software design: constraint posting and search pipeline, class hierarchy and responsibilities, and the extension points that enable controlled experimentation and reproducibility.

- **Chapter 5 — Model Validation.** Presents the validation methodology—unit tests, figure-driven evaluation, and MUS-based analysis—together with the activation/deactivation mechanism for constraints and the control loop used to isolate inconsistencies.
- **Chapter 6 — Rule-by-Rule Analysis.** Conducts a systematic audit of the rule set, discussing delayed notes and strata, undocumented or derogated cases, and providing reasoned corrections grounded in both musical practice and solver evidence.
- **Chapter 7 — Full Validated Model of Fux Counterpoint.** Assembles the validated rule catalogue—general and species-specific, including fifth-species rhythmic logic—states their domains of application, and summarizes the rules scope in compact tables.
- **Chapter 8 — Limitations and Future Work.** Identifies current boundaries of the approach and outlines concrete avenues for extension, from rhythmic and ornamental coverage to multi-voice interactions and broader stylistic variants.
- **Chapter 9 — Conclusion.** Synthesizes the main results and contributions, and sketches how the framework may support future research in computational counterpoint.

Chapter 2

Background

This chapter fixes the musical and historical groundwork that the formal model presupposes. It clarifies the terminology used throughout the thesis, recalls the essentials of species counterpoint, and situates our reading of the treatises on which the model relies.

2.1 Music Concepts

This section provides a brief overview of the key terms, ensuring consistency of usage and avoiding ambiguity in the chapters that follow.

Staff A set of five parallel horizontal lines and the four spaces between them, used in Western music notation to indicate pitch and rhythm. The position of a symbol on the staff, in combination with a clef, determines the pitch it represents.

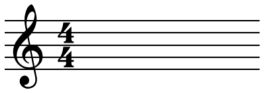


Figure 2.1: Staff with a treble clef, an empty key signature and a 4/4 time signature.

Note A symbol in musical notation indicating both a specific pitch and its duration. The visual form of the note (filled or unfilled head, stem, flags) represents its rhythmic value, while its vertical position on the staff specifies its pitch.

Beat The basic unit of time in music, representing the regular pulse that underlies a piece. Beats are grouped into measures according to the time signature.

Measure A segment of time in musical notation, defined by a set number of beats as indicated by the time signature, and separated from adjacent measures by bar lines. Measures provide a structural framework for rhythm and phrasing. In a common $\frac{4}{4}$ time signature, a measure is made up of four beats.

Pitch The perceived highness or lowness of a sound, determined primarily by the frequency of its vibration, measured in hertz (Hz).

MIDI Acronym for Musical Instrument Digital Interface. A communication protocol for electronic musical instruments and computers. In notation, MIDI pitch values are integers from 0 (C–1) to 127 (G9), each corresponding to a specific note frequency.

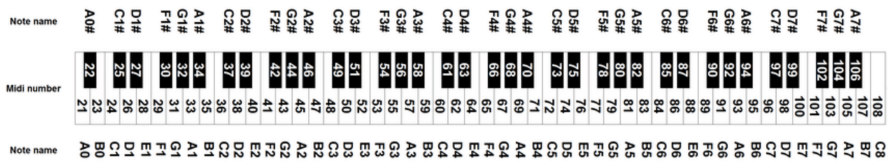


Figure 2.2: classical keyboard note names and MIDI values

Intervals The pitch distance between two notes, measured in semitones or by their ordinal name (e.g., minor third, perfect fifth). Intervals can be melodic (successive notes) or harmonic (simultaneous notes).

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

Figure 2.3: MIDI values of the intervals over an octave range

Semitone The smallest standard interval in Western music, equal to one twelfth of an octave. It corresponds to the distance in pitch between two consecutive keys on a piano keyboard, whether white-to-black or white-to-white where no black key intervenes (e.g., E to F, or B to C).

Step A melodic movement of either one semitone (half step) or two semitones (whole step) between consecutive notes in a scale.

Leap A melodic movement larger than a step, typically defined as an interval of a third or more between consecutive notes.

Types of notes Categories of note values defined by their relative duration. Common types include whole notes (longest standard duration), half notes, quarter notes, eighth notes, and sixteenth notes, each typically half the dura-

tion of the previous.

- **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.

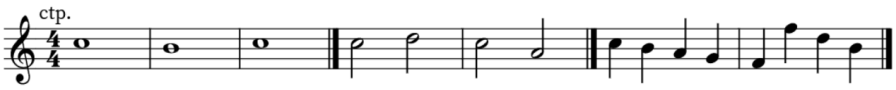


Figure 2.4: The 3 main types of notes used in the counterpoint.

Syncopation A rhythmic device in which emphasis is placed on normally weak beats or off-beats, often by sustaining a note across a strong beat or accenting unexpected positions in the measure. In the English translation of *Gradus ad Parnassum* by Alfred Mann[8], this practice is also referred to as a "ligature."

Delayed Note In counterpoint, a delayed note (also called a suspension) is a note that is held over from a previous harmony, creating a temporary dissonance on the thesis (downbeat) before resolving stepwise—usually downward—into a consonant interval on the following arsis (upbeat).

Tonic The central pitch of a scale, serving as its tonal "home" and the point of resolution for melodic and harmonic progressions.

Scale An ordered sequence of pitches spanning an octave, constructed according to a specific pattern of intervals (e.g., major, minor, modal).

Key A tonal framework defined by a tonic and a scale type (e.g., C major, A minor), determining the set of diatonic pitches used in a piece.

Mode A type of scale distinguished by its pattern of whole and half steps relative to the tonic. In Western music, common modes include Ionian (major), Dorian, Phrygian, Lydian, Mixolydian, Aeolian (natural minor), and Locrian.

Diatonic Pertaining to a set of seven pitches forming a major or minor scale within a given key, excluding chromatic alterations.

Chromatic Pertaining to the complete set of twelve pitches in the octave, including both the diatonic pitches and their altered (sharp/flat) counterparts.

Borrowed note A pitch that lies outside the diatonic scale of the current key, typically borrowed from a parallel mode or key.

Degree The position of a note within a scale, numbered from 1 (the tonic) to 7. Degrees are typically indicated using Roman numerals for harmonic analysis (I–VII).

Joint degree In melodic motion, a note that is one scale degree away from another note within the same scale, producing a stepwise movement (either ascending or descending by a second). Joint degrees correspond to motion by step.

Disjoint degree In melodic motion, a note that is more than one scale degree away from another note within the same scale, producing a leap. Disjoint degrees correspond to motion by intervals larger than a second.

Diminution In counterpoint, the process of filling the space of a larger interval (often a leap of a third) with one or more intermediate passing notes; more generally, the subdivision of a note into shorter rhythmic values.

Thesis or **Downbeat** The first and typically strongest beat of a measure.

Arsis or **Upbeat** A weaker beat or subdivision that leads into the next strong beat. With a common 4/4 time signature, the arsis is the third beat of any measure.

Ornaments Small, often quick embellishments added to a note or a short phrase to enrich the melody and make it more expressive.

2.2 Counterpoint

The counterpoint is a fundamental concept in the field of music theory and composition, representing a technique by which two or more independent melodic lines are combined to create a harmonious whole. The Counterpoint has had an enormous impact on classical Western music, particularly from the Renaissance and Baroque periods, and continues to influence contemporary musical practices.

2.3 *Gradus ad Parnassum* by Johann Joseph Fux

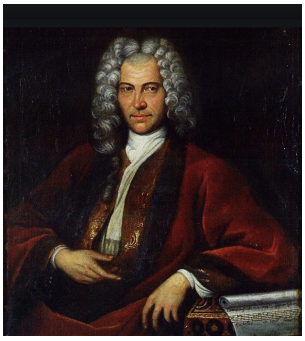


Figure 2.5: Portrait of Johann Joseph Fux by Jacob van Schuppen (around 1725).

Johann Joseph Fux, born in Hirtenfeld in 1660 and deceased in 1741, was an Austrian composer, music theorist, and pedagogue of the late Baroque era. He is best remembered for his treatise on counterpoint, *Gradus ad Parnassum*, first published in 1725 [7]. This work has become a cornerstone of Western music pedagogy, influencing generations of composers and theorists.

Gradus ad Parnassum is divided into two books. The first, now of mainly historical interest, reflects early eighteenth-century views on the mathematical foundations of music. The second is devoted to musical practice and addresses two central topics: counterpoint and fugue. Although Fux considered the fugue study of great importance, the counterpoint portion has achieved the most lasting influence and forms the basis for this thesis.

The treatise presents its ideas in the form of a dialogue between a master and a student, moving step by step through the method known as species counterpoint. This approach begins with simple note-against-note exercises and progressively introduces rhythmic variation, dissonance treatment, suspensions, and other contrapuntal devices. The dialogue form gives the work a pedagogical charm and historical authenticity but also means that rules are sometimes implied rather than systematically codified. Exceptions are occasionally given as examples rather than formal statements, and certain concepts rely on the shared assumptions of Fux's time, which can lead to interpretive ambiguities when attempting a strict formalization.

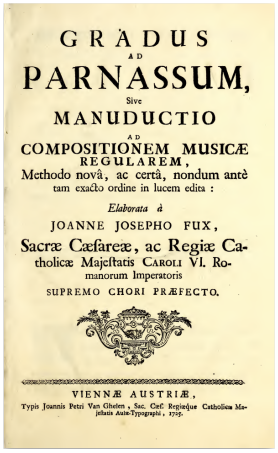

GRADUS
AD
PARNASSUM,
Sive
MANUDUCTIO
AD
COMPOSITIONEM MUSICÆ
REGULAREM,
Methodo novâ, ac certâ, nondum antè
tam exacto ordine in lucem edita :
Elaborata à
JOANNE JOSEPHO FUX,
Sacræ Cæfareæ, ac Regiæ Catholicæ Majeftatis CAROLI VI. Romanorum Imperatoris
SUPREMO CHORI PRÆFECTO.
VIENNÆ AUSTRIÆ,
Typis Joannis Petri Van Ghelen, Soc. Cæf. Regiæque Catholicæ Majeftatis Aulæ-Typographi, 1725.

Figure 2.6: Cover of *Gradus ad Parnasum*

Influence and Legacy

The influence of *Gradus ad Parnassum* has been immense. Composers such as J.S. Bach, Mozart, Haydn, and Beethoven are known to have studied it closely—Haydn working through every lesson, and Beethoven creating a condensed abstract for personal reference. Its pedagogical structure and gradual learning process have made it a reference point for counterpoint teaching for nearly three centuries. Even today, it remains part of academic curricula worldwide.

From Historical Text to Computational Model

In the field of computer-assisted composition, Gradus ad Parnassum offers an unparalleled starting point for formalization due to its clear pedagogical progression and emphasis on rule-governed practice. However, the conversational style and occasional lack of explicitness mean that certain rules require interpretation before they can be translated into executable constraints.

2.4 *Traité de Contrepoint* by Noël-Gallon and Marcel Bitsch

While *Gradus ad Parnassum* stands as the historical and pedagogical foundation of the present work, its dialogical format—cast as an exchange between

master and student—inevitably introduces a degree of ambiguity. Certain rules are implied rather than explicitly stated, and exceptions are often embedded in anecdotal or contextual remarks. This stylistic choice reflects its eighteenth-century pedagogical context but can create interpretative uncertainty when attempting to formalize its principles into strict computational constraints.

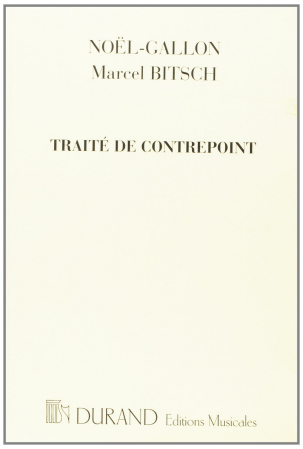


Figure 2.7: Cover of *Traité de Contrepoint*

By contrast, the *Traité de Contrepoint* [9] authored by Noël Gallon and later revised by Marcel Bitsch adopts a markedly different approach. First published in the 20th century within the framework of the French conservatoire tradition, this treatise is structured as a systematic and exhaustive exposition of counterpoint rules. Rather than unfolding as a narrative, it presents principles in direct, numbered form, each illustrated with precise musical examples. The organization is linear and explicit: rules are stated, justified, and exemplified without relying on implied context.

This difference in presentation is particularly valuable for a computational formalization such as FuxCP. Even though certain prescriptions in Gallon–Bitsch differ from those in Fux's text—reflecting both stylistic and pedagogical evolutions—the *Traité de Contrepoint* provides a clear reference point for resolving uncertainties. Where *Gradus ad Parnassum* might leave room for interpretation due to its conversational tone or its reliance on period-specific assumptions, Gallon and Bitsch's method offers unequivocal guidance, phrased in a way that directly lends itself to constraint definition.

For the purposes of the FuxCP project, the Gallon–Bitsch treatise serves not as a replacement for Fux, but as an authoritative auxiliary source. Its exhaustive structure ensures that rules are stated in their most operational form, enabling more precise encoding in constraint programming and reducing the risk of misinterpretation when translating historical principles into computational logic.

2.5 Forms and Configurations of Counterpoint

In the rest of this work, when we want to talk about the configuration of counterpoint, we will use the number of voices and the type of species used. It is therefore important to remember what this means.

Traditionally, as presented by Johann Joseph Fux in *Gradus ad Parnassum*, counterpoint may be written in two, three, or four voices. The simplest form involves a fixed melodic line, known as the *cantus firmus*, over which one or more counterpoint lines are composed. The *cantus firmus* (Latin for "fixed song") is a pre-existing melody, typically composed of long, evenly spaced notes and entirely consonant intervals. It serves as the structural backbone of

the composition, providing a stable framework against which the other voices are constructed.

In two-voice counterpoint, the interaction is limited to the cantus firmus and a single added voice, which allows a focused exploration of intervallic relations and melodic constraints. Three-voice counterpoint increases the complexity by requiring the composer to manage the interaction among all pairs of voices, while maintaining independence and avoiding forbidden motions such as parallel fifths. Four-voice counterpoint, commonly used in choral or harmonic contexts, demands full vertical coordination and careful distribution of motion, dissonance, and spacing.

First Species

In first species counterpoint, each note in the counterpoint line corresponds exactly to one note in the *cantus firmus*, creating a strict 1:1 rhythmic ratio.

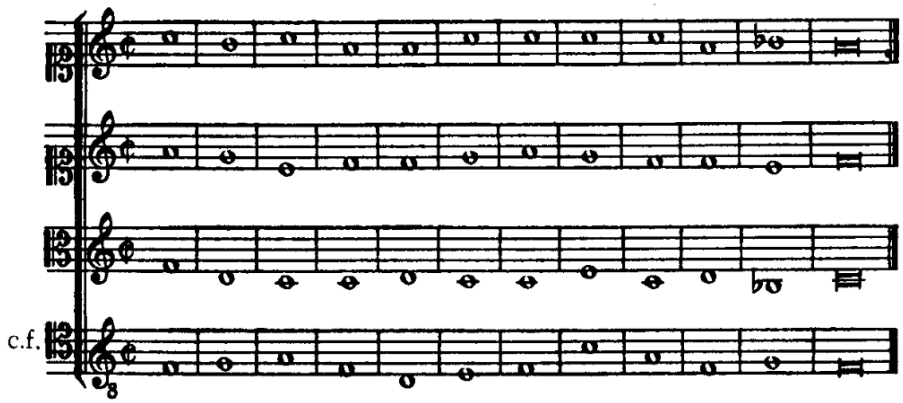


Figure 2.8: Example of a four-voice counterpoint composition in the first species. [8, p.116 fig.172]

Second Species

Second species introduces rhythmic variation by allowing two counterpoint notes for every *cantus firmus* note (2:1 ratio).

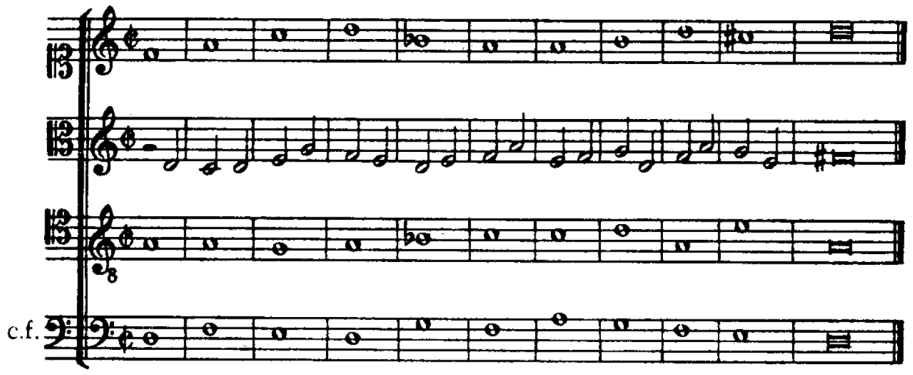


Figure 2.9: Example of a four-voice counterpoint composition in the second species. [8, p.118 fig.176]

Third Species

Third species expands rhythmic density further, with four counterpoint notes per *cantus firmus* note (4:1 ratio).

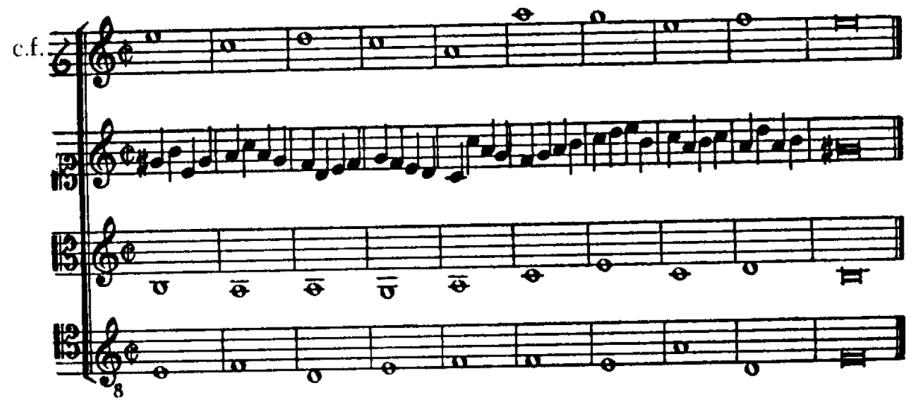


Figure 2.10: Example of a four-voice counterpoint composition in the third species. [8, p.124 fig.184]

Fourth Species

Fourth species centers on the use of suspensions, where notes are tied over from weak beats to strong beats. The rhythmic structure typically involves half-note values tied across barlines. This species introduces controlled metrical tension and delayed resolution, enhancing expressive potential through syncopation.

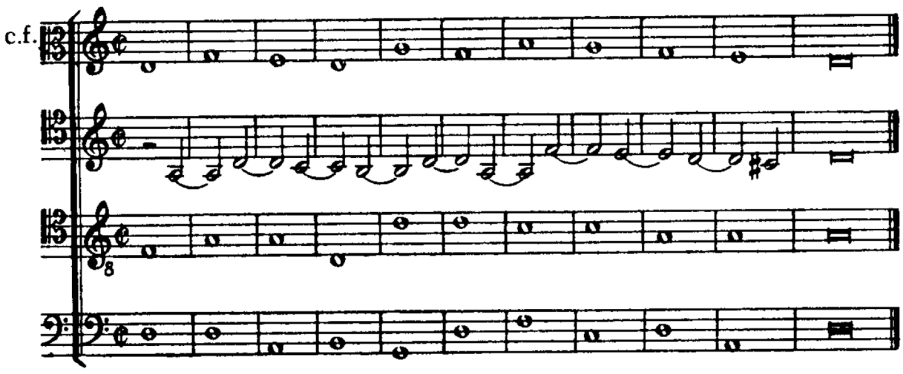


Figure 2.11: Example of a four-voice counterpoint composition in the third species. [8, p.132 fig.196]

Fifth Species

Fifth species, also known as Florid counterpoint, integrates elements of all previous species, combining various rhythmic values, suspensions, and ornamental figures within a single line. It aims to emulate the natural flow of real polyphonic music while still adhering to contrapuntal rules. This species allows for the greatest expressive freedom and serves as a bridge to free composition.

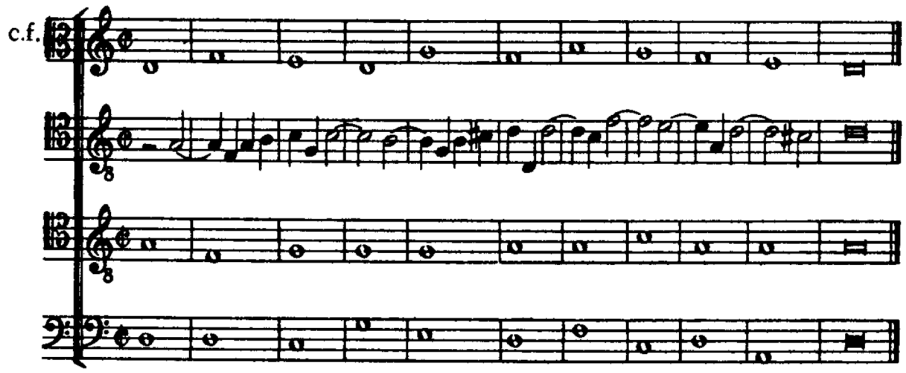


Figure 2.12: Example of a four-voice counterpoint composition in the third species. [8, p.135 fig.201]

Species Mixture

In addition to the five traditional species, Fux also acknowledges the possibility of *mixed species* counterpoint, where different species are combined within a single composition. This type of writing allows for more realistic polyphonic textures and serves as a transitional step toward free composition.

The title of this section follows the terminology used in the *Traité de contrepoint* by Gallon and Bitsch, where the authors explicitly refer to this practice as "mélange des espèces" [9, p.16].

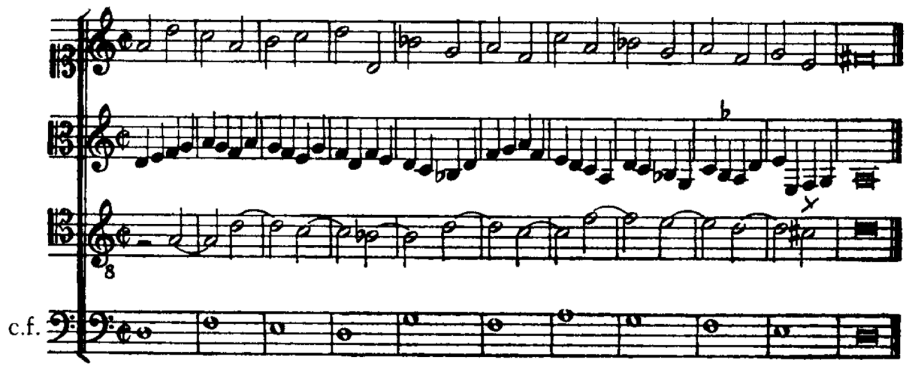


Figure 2.13: Example of a four-voice counterpoint with mixed-species. [8, p.137, fig.204]

We do not discuss this type of counterpoint in detail, since Fux himself treats it only briefly and provides a single illustrative example in his treatise which is shown above.

Chapter 3

Concepts, Variables, and Data Structures

In order to faithfully reproduce Fuxian counterpoint in a computational setting, it is essential to develop a precise formalization that translates musical concepts into well-defined mathematical structures. This chapter presents the formal framework used to model the rules and stylistic preferences of species counterpoint, serving as the theoretical foundation for the solver's implementation.

The formalization begins with a clear definition of the core musical elements—parts, strata, and voices—which are central to the structure of any polyphonic composition. From these concepts, we introduce the variables, functions, and constraints that govern note pitches, intervals, and motion. These components are then assembled into a constraint satisfaction problem (CSP), where musical rules are enforced through hard constraints, and stylistic guidelines are represented via cost functions.

Throughout this chapter, the formalization is presented in a mathematical language, but always grounded in musical logic. Where applicable, helper arrays and derived functions are introduced to improve the clarity and efficiency of constraint definitions. This chapter lays the groundwork for the solver's algorithmic behavior and ensures a rigorous mapping between musical theory and its computational counterpart.

3.1 Voices, Parts, and Strata

To understand the formal structure, we must first define three foundational concepts: parts, strata, and voices, as initially defined by A. Lamotte[13]. These concepts were developed to manage the increasing complexity of three-part counterpoint compared to earlier formalizations and it remains perfect for counterpoint to any number of parts.

- **Parts**: The most straightforward concept, parts represent the actual musical lines, corresponding to what a person sings or an instrument plays.

Each musical staff in a composition represents a part. In counterpoint compositions, parts include the *cantus firmus* and the counterpoint(s) line(s).

- **Strata**: Unlike parts, strata are an abstract concept primarily useful in the mathematical formalization of Fux's rules. Strata help identify, at any given measure, the relative pitch ranking of voices, determining which voice is the lowest, highest, or any rank in between. The number of strata equals the number of parts. If parts remain within their voice range and do not cross, strata align with parts. However, since parts can cross, the concept of strata becomes particularly useful.

  As explained by A. Lamotte[13], strata enable the application of more nuanced rules, such as defining a relation between the lowest sounding voice and another part, regardless of which part actually plays that note.

  It is important to note that the relative pitch ranking of voices is calculated according to the pitch of the first note in each measure. This means that the rank of strata is calculated per measure.

- **Voices**: This general concept covers both parts and strata. When discussing voices, the reference is collectively to both parts and strata.

Strata Formalization

To ensure a clear mapping between parts and strata, especially when two parts play the same pitch, a bijection is enforced at every measure: each stratum is associated with exactly one part, and vice versa. In case of ties (e.g., two parts playing the same lowest note), precedence is given according to a fixed order: *cantus firmus* first, followed by the counterpoints (cp_1, cp_2, cp_3). For example, if the $cantus firmus$ and cp_1 share the lowest pitch, the lowest stratum is assigned to the $cantus firmus$.

This mapping ensures that strata are well-defined and contain notes from only one part per measure. When sorting notes within a measure, the system orders them from lowest to highest, resolving ties according to the predefined part priority.

In the present work, the implementation of strata has been modified in order to improve the handling of delayed notes, particularly in the fourth and fifth species. In the previous version, strata sometimes failed to reflect the harmonic note actually carrying the weight of the measure (for example, by privileging the suspension rather than its resolution). The new implementation, described in detail in the section 6.1, ensures that each beat is assigned to the musically relevant note, thereby allowing harmonic intervals to be computed correctly in all contexts.

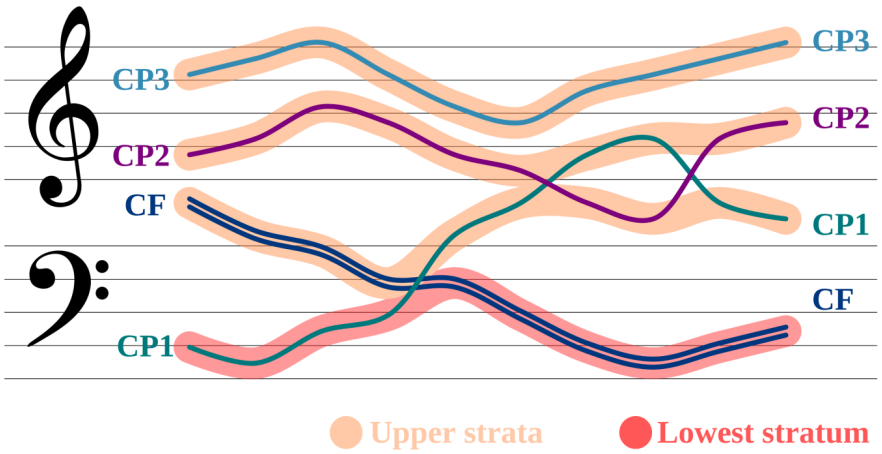


Figure 3.1: Visualisation of parts and strata

Mathematical definition of the notes of the α-th stratum

Given that when $\alpha = 0$, it is the lowest stratum:

$$\begin{aligned} &\forall \alpha \in [0,3] \quad \forall j \in [0, m-1] : N(l[\alpha])[0,j] = X[\alpha] \\ &\text{where } X = sorted([N(cf)[0,j], N(cp_1)[0,j], N(cp_2)[0,j], N(cp_3)[0,j]]) \\ &\text{and } l = [a,b,c,d] \text{ to keep the right stratum notation.} \end{aligned} \tag{3.1}$$

The remaining notes within each measure belong to the same musical part as the first one initially associated with the stratum, ensuring that no single stratum combines notes from multiple distinct parts. The sorted function organizes these notes from the lowest to the highest pitch. In cases where pitches are identical, priority is given first to the *cantus firmus*, followed by the counterpoint lines in their respective order, as previously described.

3.2 Variables and Arrays

As a continuation of the conceptual groundwork on voices, parts, and strata, this section presents the core variables used in the formalization of counterpoint rules.

Although this formalization is mathematical and abstract in nature, it serves as the theoretical basis for the practical implementation, which is discussed later in the thesis. Notably, the computer implementation in C++ introduces helper structures and class hierarchies that are not strictly reflected here. Nevertheless, the main variables and logic remain consistent, and both models yield the same set of solutions, namely, the attribution of MIDI pitches to the counterpoint voices.

Structure of the Constraint Problem

The formalization is defined as a Constraint Satisfaction Problem (CSP), characterized by:

- a set of **variables** (mostly integers),
- their respective **domains** (such as MIDI pitch ranges),
- and a set of **constraints** applied to them.

Most variables are stored in integer arrays, as the pitch of each note is represented by a MIDI number (0 to 127, with 1 semitone per unit). Each note in the generated counterpoints has its own variable, making the model mostly composed of arrays indexed by beat and measure.

Constants and Functions used in the Mathematical Formalization

Before delving into the discussion of the significant arrays to which the constraints will be applied, it is imperative to establish the constants and functions that will facilitate their definition.

Constants

- $\mathcal{B}(v)$ – the set of beats on which a note can be played according to the species of voice v. For the first species, the only beat in a measure is $\{0\}$, as there is only a note on the first beat. For the second species, the set of beats is $\{0, 2\}$. For the third species, it is $\{0, 1, 2, 3\}$. For the fourth species: $\{0, 2\}$. And for the fifth species: $\{0, 1, 2, 3\}$.
- $Cons$ – the set of all consonances, i.e. $\{0, 3, 4, 7, 8, 9\}$, where $Cons_p$ are the perfect consonances, i.e. $\{0, 7\}$, and $Cons_{h\_triad}$ are the consonances of a harmonic triad as defined by Fux, i.e. $\{0, 3, 4, 7\}$.
- Dis – the set of all dissonances, i.e. $\{1, 2, 5, 6, 10, 11\}$.
- $d(v)$ – the duration of a note according to the species of voice v.

Core Variable - Note Pitches: N

The array N is the primary structure in the formalization, storing the MIDI pitch values of each note in the composition. It is addressed as $N[i, j]$, meaning the note at beat i (ranging from 0 to 3) of measure j (ranging from 0 to m, the total number of measures).

To distinguish voices, the notation extends to:

- $N(cf)$: the notes of the *cantus firmus*,

- $N(cp1)$, $N(cp2)$, $N(cp3)$: the notes of counterpoints. Here the first, second, and third counterpoint, respectively
- $N(a)$, $N(b)$, $N(c)$, $N(d)$: notes of strata (from lowest to highest).

By default, when N is mentioned without a voice, it is understood to apply to all parts:

$$N \equiv \forall p \in cf, cp1, cp2, cp3 : N(p) \tag{3.2}$$

This omission can be made for all other arrays.

Although in theory all rules could be defined in terms of N alone, the formalization introduces derived arrays to reflect higher-level musical ideas which make constraints clearer and more efficient: **Harmony**, **Melody**, and **Motion**.

Harmonic Intervals: H_{brut} and H

The array H_{brut} stores the absolute interval, in semitones, between two notes in different voices. It does not account for octave equivalence and can take values from 0 to 127, consistent with the MIDI pitch number system.

$$\begin{gathered}\forall v_1, v_2 \in \{cf, cp_1, cp_2, cp_3, a, b, c, d\}, \forall i \in \mathcal{B}, \forall j \in [0, m-1] \\ H_{brut}(v_1, v_2)[i, j] = |N(v_1)[i, j] - N(v_2)[0, j]|\end{gathered} \tag{3.3}$$

To evaluate intervals independent of octave, the absolute interval H_{brut} is reduced modulo 12. This results in the array H, which contains values from 0 to 11, representing the 12 semitones of the octave.

$$H(v_1, v_2)[i, j]) = H_{brut}(v_1, v_2)[i, j] \mod 12 \tag{3.4}$$

Unless otherwise specified, H is the standard array used for evaluating harmonic intervals or harmonies.

It is important to note that for harmonic constraints, only the first beat of each measure is considered in the compared voice v_2. For convenience, $H(v)$ is the shorthand for $H(v, a)$: the interval between v and the lowest stratum.

Consonance: $IsCons(v)$

A Boolean array $IsCons(v)$ is also defined, indicating whether the interval between voice v and the lowest stratum is consonant (according to predefined sets of allowed intervals):

$$\begin{gathered}\forall i \in \mathcal{B}(v), \forall j \in [0, m-1] \\ IsCons(v)[i, j] = \begin{cases} \top & \text{if } H(v)[i, j] \in Cons \\ \bot & otherwise \end{cases} \\ \text{where } Cons = \{0, 3, 4, 7, 8, 9\}\end{gathered} \tag{3.5}$$

Interval name	Number of semitones	Example	Type of Interval
Perfect prime or perfect octave	0	C – C	Perfect Consonance
Minor second	1	C – D ♭	Dissonance
Major second	2	C – D	Dissonance
Minor third	3	C – E ♭	Imperfect Consonance
Major third	4	C – E	Imperfect Consonance
Perfect fourth	5	C – F	Dissonance
Tritone	6	C – G♭	Dissonance
Perfect fifth	7	C – G	Perfect Consonance
Minor sixth	8	C – A♭	Imperfect Consonance
Major sixth	9	C – A	Imperfect Consonance
Minor seventh	10	C – B♭	Dissonance
Major seventh	11	C – B	Dissonance

Table 3.1: Musical intervals with the number of semitones, examples, and type of interval

Melodic Intervals: M_{brut} and M

The melodic progression of each part is represented through:

- M_{brut}: the signed melodic interval between two successive notes,
- M: the absolute value of that interval.

Formally, the melodic interval when related to a part is given by:

$$\forall p \in \{cf, cp_1, cp_2, cp_3\},\ \forall x \in \{1, 2\},\ \forall i \in \mathcal{B},\ \forall j \in [0, m-1-x]$$
$$M^x_{brut}(p)[i,j] = \begin{cases} N(p)[i+xd, j] - N(p)[i,j] & \text{if } i + xd < 4 \\ N(p)[(i+xd) \bmod 4, j+1] - N(p)[i,j] & \text{if } i + xd \geq 4 \end{cases}$$
$$M^x(p)[i,j] = |M^x_{brut}(p)[i,j]| \tag{3.6}$$

With $d = d(p)[i,j]$. This is done to make the formula easier to read. It is interesting to note that for the first to fourth species, all values contained in the array d(v) are the same because the same type of note is always played (except for in last measure, where it's always a whole note). However, in the fifth species, the notes can be of different types, so in order to generalize the formula to all species, it is important to take the value of d(v) at position [i,j] to calculate $M^x_{brut}(p)[i,j]$.

x, on the other hand, corresponds to the rank of the next note in the sequence. More precisely, it identifies which next note to use when computing the melodic interval from a given note. For example, if $x = 1$, we are interested

in the interval between the current note and the very next note in the part; if $x = 2$, then we compute the interval between the current note and the second note that follows it in the part.

In practice, M^1 is the default, so $M \equiv M^1$. Adding the omission of the parameter p as previously explained in the section 3.2 on the set of variables N, we obtain:

$$M \equiv \forall p \in \{cf, cp_1, cp_2, cp_3\} : M^1(p). \tag{3.7}$$

Motion Between Voices: P

To evaluate how a part moves relative to the lowest stratum, a motion array $P(p)$ is defined which can contain the following value:

- 0 – Contrary motion,
- 1 – Oblique motion,
- 2 – Direct motion,
- -1 – Not applicable (when the part is itself the lowest).

Motion is computed between the current note and the first note of the next measure using an offset $x = b - i$ where b is the number of notes contained in the current measure.

$$\forall p \in \{cf, cp_1, cp_2, cp_3\}, \quad \forall x \in \{1, 2\}, \quad \forall i \in \mathcal{B}, \quad \forall j \in [0, m-1], \quad x := b - i$$

$$motion(p)[i,j] = \begin{cases} 0 & \text{if } (M^x_{brut}(p)[i,j] > 0 > M(a)_{brut}[j]) \\ & \quad \vee (M^x_{brut}(p)[i,j] < 0 < M(a)_{brut}[j]) \\ 1 & \text{if } M^x_{brut}(p)[i,j] = 0 \oplus M(a)_{brut}[j] = 0 \\ 2 & \text{if } (M^x_{brut}(p)[i,j] > 0 \wedge M(a)_{brut}[j] > 0) \\ & \quad \vee (M^x_{brut}(p)[i,j] < 0 \wedge M(a)_{brut}[j] < 0) \\ & \quad \vee (M^x_{brut}(p)[i,j] = 0 = M(a)_{brut}[j]) \end{cases}$$

$$P(p)[i,j] = \begin{cases} -1 & \text{if } A(p)[j] \\ motion(p)[i,j] & \text{if } \neg A(p)[j] \end{cases} \tag{3.8}$$

Bijection Helper: A

To preserve a bijective mapping between parts and strata, even when several notes tie as the lowest, a Boolean array $A(p)$ is used. It specifies whether part p is the lowest stratum at each measure j, using priority order: $cf > cp1 > cp2 > cp3$. Only one part can be assigned as the lowest stratum at any time:

$$\forall j \in [0, m-1]$$

$$A(cf)[j] = \begin{cases} \top & \text{if } N(cf)[0,j] = N(a)[0,j] \\ \bot & \text{else} \end{cases}$$

$$A(cp_1)[j] = \begin{cases} \top & \text{if } (N(cp_1)[0,j] = N(a)[0,j]) \wedge \neg A(cf)[j] \\ \bot & \text{else} \end{cases}$$

$$A(cp_2)[j] = \begin{cases} \top & \text{if } (N(cp_2)[0,j] = N(a)[0,j]) \wedge \neg A(cf)[j] \wedge \neg A(cp_1)[j] \\ \bot & \text{else} \end{cases}$$

$$A(cp_3)[j] = \begin{cases} \top & \text{if } \neg A(cf)[j] \wedge \neg A(cp_1)[j] \wedge \neg A(cp_2)[j] \\ \bot & \text{else} \end{cases} \tag{3.9}$$

Fifth Species and the S Array

While the first to four species maintain a consistent rhythmic nature, the fifth species introduces variable rhythms (e.g., suspensions, syncopations). Therefore, an additional array S is defined to specify the species constraint of each note:

$$\forall \rho \in \text{positions}(m) :$$

$$S[\rho] = \begin{cases} 0 & \text{if } N[\rho] \text{ is not constrained by any species} \\ 1 & \text{if } N[\rho] \text{ is constrained by the first species} \\ 2 & \text{if } N[\rho] \text{ is constrained by the second species} \\ 3 & \text{if } N[\rho] \text{ is constrained by the third species} \\ 4 & \text{if } N[\rho] \text{ is constrained by the fourth species} \end{cases} \tag{3.10}$$

This allows the model to enforce the correct rules for each type of note behavior within the same part.

Modeling Preferences: Costs C

Fux's treatise includes both hard rules (strictly enforced) and soft preferences (stylistic guidelines). To formalize preferences, the model introduces cost variables. These cost arrays quantify violations of preferences. When searching for solutions, the solver minimizes the sum of costs, turning the problem into an optimization task. In the mathematical framework used to formalize constraints throughout this thesis, cost functions are denoted by the symbol C. It is worth noting that most of these costs are represented as arrays of variables, with each element corresponding to a separate cost assigned per measure:

$$\forall j \in [0, m-1]$$

$$C[j] = \begin{cases} 0 & \text{if the preference is respected at measure j} \\ \mathbb{N}^+ & \text{otherwise} \end{cases} \tag{3.11}$$

Examples include: $C_{prefer\_harmonic\_triad}, C_{prefer\_harmonic\_triad}$, among others. All cost variables form the set C, which is crucial for guiding the solver toward musically satisfying results.

This comprehensive structure of variables ensures that the formalization accurately captures both the rigid rules and stylistic nuances of Fux's counterpoint, and provides a solid mathematical foundation for the subsequent computer implementation.

3.3 Notational Clarifications and Corrections

In the process of refining the formalization presented in this thesis, I made several adjustments to the mathematical notation used in previous work to improve clarity, consistency, and accuracy. These changes focus particularly on indexing conventions and the way ranges are expressed when referring to sequences of notes, measures, or harmonic intervals.

In earlier versions of the formalization, there was an inconsistent use of notation when referencing the boundaries of an array or range. For example, both $N[0, m)$ and $N[0, m-1]$ were used, despite representing the same note. To resolve this, I adopted a consistent approach aligned with standard mathematical and programming conventions:

- Indexing starts at 0, meaning the first element is N[0].
- When referring to a range, I now consistently use $N[0, m-1]$, explicitly including both the start and end indices, rather than the half-open interval $[0, m)$.

This change not only improves readability but also aligns the formalization with the indexing used in the C++ implementation, thereby reducing the risk of off-by-one errors or misinterpretations.

In fact, I believe that some errors in the previous version of the formalization may have been caused by such inconsistencies. For instance, in the definition of rule 1.H7 from the thesis of L. Cleenewerk and D. de Patoul[5], for three voices, the penultimate harmonic interval is incorrectly referred to as $H[0, m-1]$, which actually denotes the last harmonic interval. To correctly refer to the penultimate harmonic, the proper indexing should be $H[max(\mathcal{B}), m-2]$, where $max(\mathcal{B})$ selects the last beat of the measure, and $m-2$ correctly references the second-to-last measure. I have corrected this and similar errors throughout the formalization.

These notational clarifications and corrections are essential for ensuring that the mathematical model matches the intended musical behavior, and they help prevent errors both in theoretical reasoning and in practical implementation.

Chapter 4

Program architecture

This chapter provides an overview of the program architecture, as established in previous work. The architecture, originally introduced by L. Cleenewerk and D. de Patoul in their master thesis [5], focuses on the C++ implementation designed for counterpoint composition. It takes advantage of object-oriented programming principles to create a structured and maintainable system capable of handling the complexities of counterpoint theory.

4.1 Inheritance and Class Structure

The program utilizes an inheritance-based structure to manage different elements of counterpoint theory. Central to this structure is the Voice class, which serves as a base class for various types of voice, such as the *cantus firmus* and different species of counterpoints. This approach centralizes common structural elements and behaviors, reducing redundancy and enhancing code maintainability.

- **Voice Class**: The foundational class that encapsulates common properties and methods shared by all types of voices.
- **Part Class**: Inherits from Voice and serves as a base for different species of counterpoints, providing a consistent interface and shared functionality.
- **Species-Specific Classes**: Classes such as `FirstSpeciesCounterpoint` and `SecondSpeciesCounterpoint` inherit from `Part` and implement species-specific constraints and behaviors.

This hierarchical structure allows for the efficient application of constraints while maintaining a clear and consistent interface.

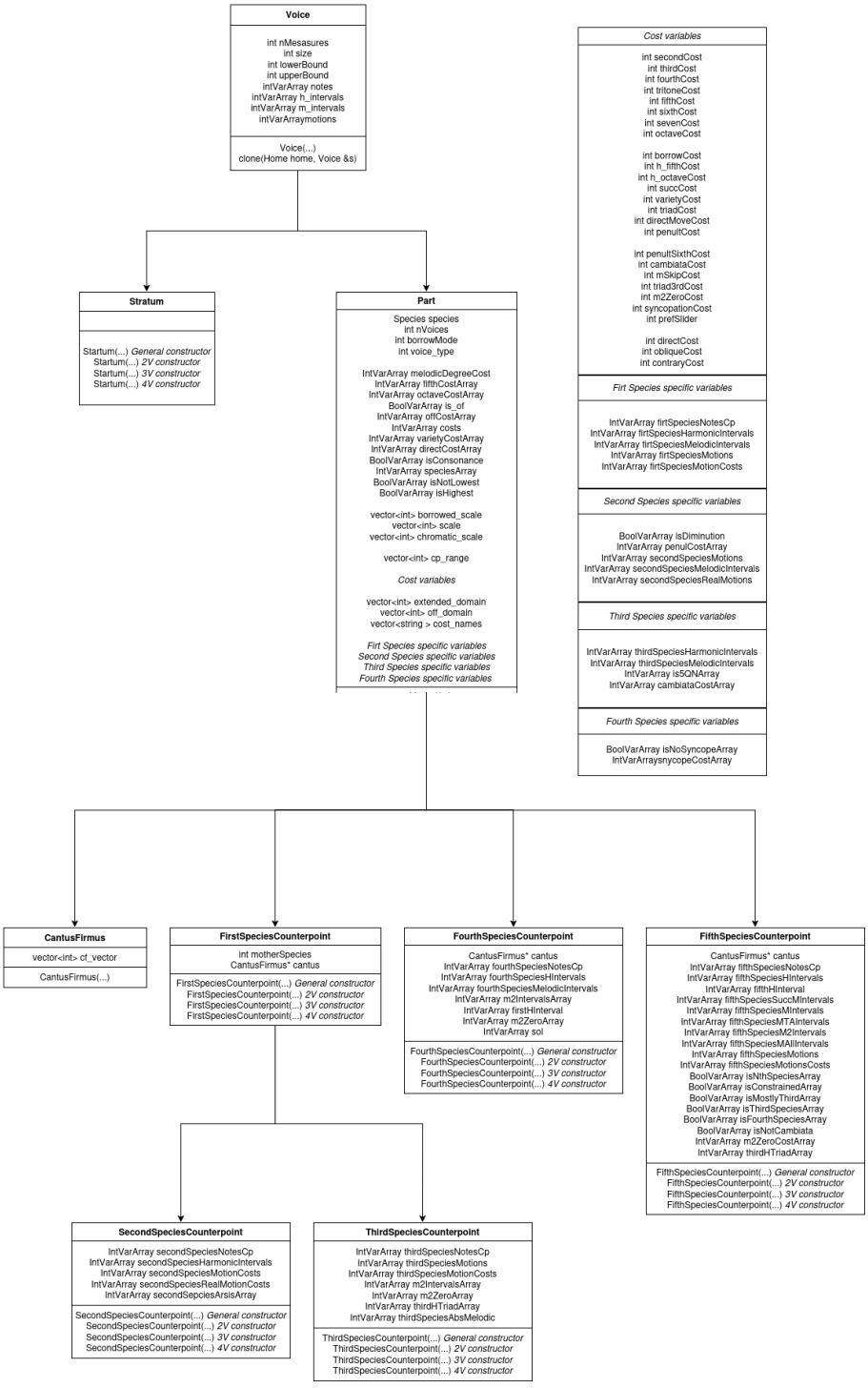


Figure 4.1: UML-style diagram of the Voice class hierarchy

Note on the `Part` class: Beyond inheriting from `Voice`, `Part` functions as a shared interface: it concentrates the common state, helper predicates/arrays, and cost structures that all species rely on, so subclasses don't have to re-implement the plumbing. To keep the diagram legible, these members are grouped together in the large rectangle adjacent to `Part` rather than crowded into the class box itself.

4.2 Managing Context and Constraints

One of the key challenges in implementing counterpoint theory is managing the context in which constraints are applied. The set of rules depends on parameters such as species and the number of voices. The program addresses this challenge through a combination of inheritance and multiple constructors.

Each species class has constructors tailored to different numbers of voices, such as two, three, and four voices. These constructors call a shared, internal constructor that applies constraints common to all instances of that species, regardless of the number of voices. This approach isolates different contexts and ensures that the appropriate set of constraints is applied automatically when a new counterpoint object is instantiated.

4.3 Hierarchy of Problem Classes

The program defines an abstract base class, `CounterpointProblem`, which is a child of the Gecode class `Space`. This class is implemented by three concrete classes: `TwoVoiceCounterpoint`, `ThreeVoiceCounterpoint`, and `FourVoiceCounterpoint`. Each of these classes handles the constraints that apply to the composition as a whole, depending on the number of voices.

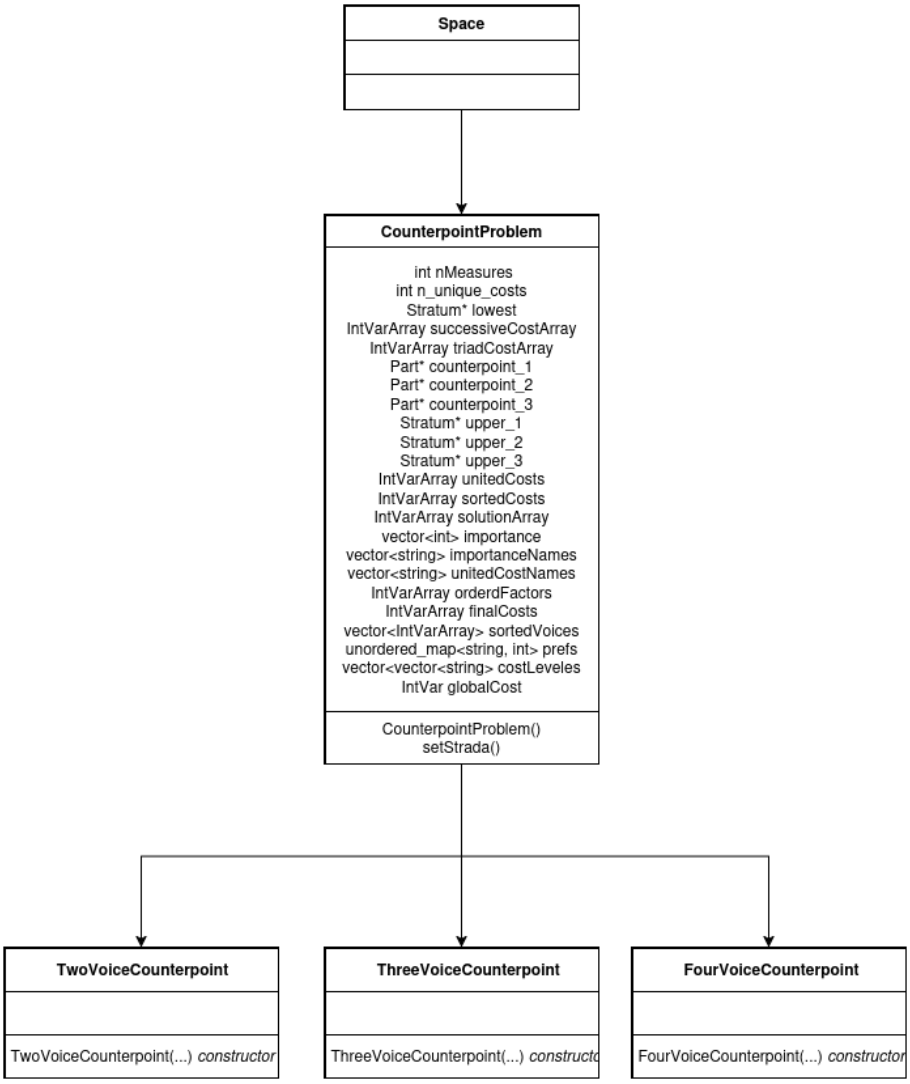


Figure 4.2: UML-style diagram of the CounterpointProblem class hierarchy

4.4 Flow of the Program

To illustrate the flow of the program, consider the example of creating a three-voice counterpoint problem with a counterpoint of the third species and another of the first species:

1. **Problem Parameters**: The user defines the problem parameters as the *cantus firmus*, the number of voices, their species, etc.

2. **Problem Creation**: The `create_problem` function dispatches the creation of the appropriate `CounterpointProblem` object according to the parameters given. In this example, a `ThreeVoiceCounterpoint` object is created.

3. **Constructor Invocation**: The `ThreeVoiceCounterpoint` constructor is invoked, which calls the necessary functions to construct the Constraint Satisfaction Problem (CSP):

   - The `CounterpointProblem` constructor. As a child of `CounterpointProblem` class, `ThreeVoiceCounterpoint` constructor call the constructor of its parent, which creates general variables, arrays and call the `CantusFirmus` constructor.
   - The `counterpoint` objects are created using the `create_counterpoint` function with the appropriate arguments which call the right constructor given the context similarly to the `create_problem` function. In this scenario, the function is called twice:
     a) For the creation of the first species counterpoint object, it invokes the relevant counterpoint constructor, specifically the three-voice constructor of the `FirstSpeciesCounterpoint` class. This constructor starts by calling the general constructor of the `FirstSpeciesCounterpoint` class, which inherits from the `Part` class. The `Part` class, in turn, inherits from the `Voice` class. This ensures that all common variables and constraints from the `Voice` and `Part` classes are applied and then those of the `FirstSpeciesCounterpoint` class. After this, the three-voice specific constructor proceeds to post the constraints specific to first species counterpoints in three-voice compositions.
     b) For the creation of the third species counterpoint object, it follows a similar process. Since the `ThirdSpeciesCounterpoint` class inherits from the `FirstSpeciesCounterpoint` class, it first calls the general constructor of the `FirstSpeciesCounterpoint` class before proceeding to its constructor. The call cascade is then the same as explained above from the general constructor of `FirstSpeciesCounterpoint`.

4. **CSP Construction**: The CSP is now constructed, and the CounterpointProblem pointer is returned by the `create_problem` function. This pointer will later be used to start the search for solutions.

Chapter 5

Model Validation

After reviewing all the essential concepts of counterpoint and how the program works, it is now time to present my contribution to the project: the systematic testing of the program and the correction of implementation errors. While the theoretical framework and formal constraints were already defined, the proper functioning of the system required careful validation through practical execution. My role consisted in thoroughly testing the program across a variety of counterpoint configurations, identifying inconsistencies between expected and actual behavior, and resolving logical or computational errors in the code. This work was essential to ensure that the system reliably applies the rules of counterpoint across different species and voice combinations, in line with both the theoretical model and musical expectations.

We have validated and corrected the formal model of Fux counterpoint that was developed in three previous master theses. Validations were done in four directions:

1. Validation with respect to implicit knowledge. Fux does not make all rules explicit in his treatise. He assumes much implicit knowledge of music theory and counterpoint. We have corrected and extended the rules with respect to this implicit knowledge.

2. Validation of the Fux rules with respect to the musical examples in the Fux treatise. The Fux treatise has many musical examples. The examples do not always respect the rules that Fux gives in the text of the treatise. We consider that Fux, being a musician, gives correct examples. But since Fux is not a mathematician, the textual explanation of the rules is not always precise or complete. We have made a technical analysis of all the examples with respect to the formal model. We use the concept of MUS (Minimal Unsatisfiable Subset) from the theory of constraint programming. This lets us determine which rules must be changed when we find an example that does not satisfy the formal model. The results of the MUS analysis are summarized in section 5.2.

3. Validation of the Fux rules with respect to the treatise of Noël Gallon and Marcel Bitsch [9]. This treatise presents Fux counterpoint in a more explicit way than the Fux treatise. In the Bitsch treatise, all rules are given explicitly

and numbered, and given a much more precise definition than what Fux has given.

4. Correction of errors in the previous formalization. In the previous formal model, we have discovered occasional errors in the constraint code that implements the rules. For example, incorrect or incomplete translation of the mathematical definitions into constraint code, or incorrect mathematical definition of textual rules. We have corrected all these errors.

The first three validations were done in close collaboration with Karim Haddad. Every single Fux rule was discussed in detail with Dr. Haddad and studied in terms of its musicality (does it give correct musical results) and in terms of the corresponding rules in the treatise of Gallon and Bitsch. Any ambiguity and error was removed to the best of our ability. Occasionally this required a musical judgement, which was decided upon after discussion with Dr. Haddad. The result of these discussions for all the rules are given in chapter 6. The full, corrected model is given in chapter 7.

5.1 Unit Tests

I started by implementing a set of unit tests to verify the correct behavior of individual constraints in the counterpoint solver. The aim was to ensure that each rule functioned as expected when applied across various counterpoint configurations, including different numbers of voices (two, three, or four) and species (first, second, etc.). For each tested rule, I created example figures that intentionally violated the constraint and checked whether the solver correctly failed to find a valid solution.

I implemented that in the FuxTest.cpp file which had already been created by L. Cleenewerk and D. de Patoul for their own test. To build these test cases, I defined a new problem instance with a *cantus firmus* and explicitly fixed the solution notes in the counterpoint voices so that the targeted constraint was broken. The rationale was that, if the constraint was correctly implemented, the solver should reject the solution.

For example, let's take rule **1.H1** which says: *"All notes on the downbeat are consonant with the notes (on the downbeat) of the lowest stratum."* Which is formalized mathematically as:

$$\forall j \in [0, m-1] \quad H[0, j] \in Cons \tag{5.1}$$

The test of this rule on the two voice counterpoint with the first species is:

```
1   void FuxTest::test_1H1_2v_1sp() {
2       cout << "Start test 1H1 2v 1sp..." << endl;
3       // define dissonant intervals
4       int dis[] = {1, 2, 5, 6, 10, 11};
5       // define the species
6       spList = {FIRST_SPECIES};
7       // define the voice types (=pitch range):
8       // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
9       v_type = {3};
10      // Test that dissonant notes are forbidden
11      for (int interval : dis) {
12          for (size_t i = 0; i < cfSize; i++) {
13              CounterpointProblem* problem = create_problem(cantusFirmus,
                    ↪ spList, v_type, melodic_params, general_params,
                    ↪ specific_params, importance, borrowMode);
14              int note = cantusFirmus[i] + 12 + interval; // note is dissonant
15              rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ,
                    ↪ note); // fix the note i
16              if (has_solution(problem)) {
17                  std::cerr   << "/!\\ ERROR test 1H1 2v 1sp: It exists a
                        ↪ solution but it shouldn't with the following
                        ↪ configuration:\n"
18                              << "Cantus firmus: ";
19                  printVector(cantusFirmus);
20                  std::cerr   << "Solution array: ";
21                  printIntVarArray(problem->getSolutionArray());
22                  std::cerr   << "> Dissonant harmonic at the mesure " << i <<
                        ↪ std::endl;
23              }
24              delete problem;
25          }
26      }
27      cout << "End test 1H1 2v 1sp..." << endl;
28  }
```

Listing 5.1: 1.H1 Test on two voice counterpoint with first species

The same tests obviously exist for all other counterpoint configurations involving this rule.

Unfortunately, this approach quickly showed its limitations. It was both time-consuming and difficult to isolate constraints, as solver failure could result from the interaction with other rules, not necessarily the one being tested.

As a result, I implemented unit tests for only about ten constraints before deciding to adopt a different, more effective method for analyzing and correcting errors in the system. After this initial testing phase, I explored more effective strategies for debugging and verifying constraint behavior, which will be presented in the subsequent chapters.

Managing Subclass Conflicts When Switching Voice Configurations

One of the technical challenges encountered during the implementation of unit tests was the inability to run multiple tests sequentially in a single execution, particularly when these tests involved different numbers of voices. As previously explained, each configuration (two, three, or four voices) corresponds to a different subclass of the `CounterpointProblem` class: `TwoVoiceCounterpoint`,

ThreeVoiceCounterpoint, and FourVoiceCounterpoint. These subclasses each initialize only the attributes required for their specific configuration. For instance:

- TwoVoiceCounterpoint initializes only Part* counterpoint_1 and Stratum* stratum_1;
- ThreeVoiceCounterpoint initializes Part* counterpoint_1, Part* counterpoint_2, Stratum* stratum_1 and Stratum* stratum_2;
- FourVoiceCounterpoint adds Part* counterpoint_3 and Stratum* stratum_3.

Because these attributes are not shared across subclasses and are initialized only when explicitly declared in the relevant class, running multiple tests in sequence, each requiring a different subclass, led to undefined behavior or segmentation faults when accessing uninitialized attributes. The root of the problem lies in the fact that the solver expects all Part* pointers to be valid, regardless of whether they are used in a specific test.

To resolve this, I chose to explicitly initialize all unused Part* and Stratum* pointers to nullptr. This ensured that the solver could safely detect and skip over unused voices, avoiding memory access issues and allowing different CounterpointProblem subclasses to be instantiated in a single execution without error.

```
1   ThreeVoiceCounterpoint::ThreeVoiceCounterpoint(vector<int> cf, vector<Species
      ↪ > sp, vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
2   vector<int> s_costs, vector<int> imp, int bm) :
3       CounterpointProblem(cf, -1, m_costs, g_costs, s_costs, imp, THREE_VOICES)
          ↪ {
4       species = sp;
5
6       //initialize upper strata
7       upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(),
          ↪ THREE_VOICES);
8       upper_2 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(),
          ↪ THREE_VOICES);
9       upper_3 = nullptr;
10
11      //create counterpoints
12
13      counterpoint_1 = create_counterpoint(*this, species[0], nMeasures, cf, (6
          ↪  * v_type[0] - 12) + cf[0], (6 * v_type[0] + 12) + cf[0], lowest,
14          cantusFirmus, v_type[0], m_costs, g_costs, s_costs, bm, THREE_VOICES)
              ↪ ;
15      counterpoint_2 = create_counterpoint(*this, species[1], nMeasures, cf, (6
          ↪  * v_type[1] - 12) + cf[0], (6 * v_type[1] + 12) + cf[0], lowest,
16          cantusFirmus, v_type[1], m_costs, g_costs, s_costs, bm, THREE_VOICES)
              ↪ ;
17      counterpoint_3 = nullptr;
18      ...
19  }
```

Listing 5.2: ThreeVoiceCounterpoint constructor with unused part and stratum initialized to nullptr

5.2 Testing Fux Examples using MUS

Motivation

After the limitations of the unit testing approach became apparent, I shifted to a new strategy based on Minimal Unsatisfiable Subsets (MUS). This pivot was motivated by a surprising and problematic observation: although the solver was designed to model Fuxian counterpoint, it frequently failed to validate examples taken directly from Fux's treatise *Gradus ad Parnassum*.

One of the most frustrating aspects of the previous work on the project was that the solver almost never returned the solution actually given by Fux, even when provided with the exact same *cantus firmus*. In principle, if the constraints implemented in the solver are truly based on Fux's theory, then the examples from his treatise should at the very least be accepted as valid. The fact that this was not the case indicated a serious gap between the theoretical model and its computational implementation.

Using Fux's examples as test material has a major advantage: these figures are guaranteed to be correct. As the system is explicitly built on Fux's rules, his published exercises constitute a reliable reference. Therefore, if the solver fails to validate one of these examples, the problem necessarily lies in the formalization—whether due to an error in logic, a misinterpretation, or an overly rigid encoding of the rule set.

Naïve Approach: Activating Constraints One by One

To identify which constraints were responsible for the failure of each figure, I first attempted a naïve diagnostic method: enabling one constraint at a time. I modified the system to allow individual activation and deactivation of each rule, and ran the solver repeatedly for each figure, checking whether a solution could be found when only a single constraint was active.

During this process, I discovered that some constraints were not documented or explained in the previous master thesis. These rules appeared in the code but were not referenced in the theoretical material available. To keep track of these, I labeled them as UX, where "U" stands for "unknown" and "X" is a unique identifier for each undocumented constraint. These UX rules are marked accordingly in the results table presented later in this chapter. A more detailed discussion of these unknown constraints and their possible roles will be provided in the next chapter.

Despite offering some preliminary insights, this one-constraint-at-a-time method has limits. Constraints in a system often interact in subtle ways. For instance, a solution might be possible when only constraint C_1 is active, or when only C_2 is active, but still fail when both are active together. This means that testing each rule in isolation is insufficient: it fails to detect combinatorial conflicts between constraints, which are often the true source of the problem.

MUS: A Systematic and Reliable Strategy

To address these limitations, I implemented a method based on Minimal Unsatisfiable Subsets (MUS). A MUS is defined as the smallest set of active constraints that, when combined, render a problem unsolvable, i.e., no valid solution can be found. Importantly, all constraints in the MUS are necessary for the failure: removing any one of them makes the figure solvable again.

This approach allows us to identify not just individual faulty rules, but also specific interactions between rules that cause a breakdown in the system. It provides a far more precise diagnostic tool than the earlier method and helps isolate the exact causes of unsatisfiability.

Implementation Details

The implementation of this MUS-based approach follows a structure very similar to the one used during the unit testing phase. However, to keep this new strategy independent and more focused on testing real musical examples, I decided to re-implement it in a dedicated file, named `figureTests.cpp`. This file contains a separate function for each Fuxian figure that could be encoded in our system. Each function defines the *cantus firmus*, fixes the counterpoint voices according to the notes provided by Fux, and runs the solver under the full set of constraints.

```
1    void FigureTests::test_2v_1sp_fig22() {
2        cout << "Start test_2v_1sp_fig22" << endl;
3        spList = {FIRST_SPECIES};
4        cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
5        cp =           {69,64,67,65,64,72,69,71,71,69,68,69};
6        v_type = {1};
7        findAllMUSes();
8    }
```

Listing 5.3: Figure Test of the figure 22 given by Fux [8]

As before, some examples from Fux's treatise had to be excluded due to stylistic or rhythmic complexities that exceeded the capabilities of the solver. In addition, all fifth species examples had to be modified, since the current program does not yet handle certain rhythmic patterns and ornaments. In this version of the solver, the fifth species can only be composed using elements from the third species (quarter notes) and the fourth species (half notes tied to create syncopation). Consequently, every tested fifth species exercise was adapted to fit this restricted rhythmic vocabulary.

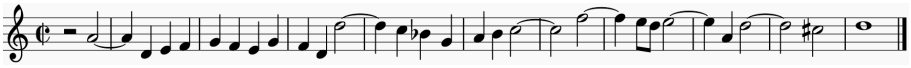


Figure 5.1: Original version of the fifth species voice from the figure 82 [8, p.64]

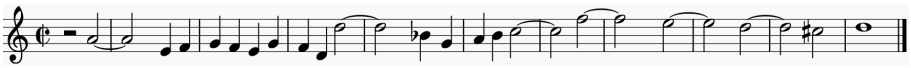


Figure 5.2: Modified version of the fifth species voice from the figure 82 to be well handled by FuxCP [8, fig.82]

In order to enable MUS extraction, I first modified the program so that each constraint could be individually switched on or off. Every call to a constraint function has been wrapped inside an `if` block, controlled by a Boolean entry in the global array `activeConstraints`.

```
1     // G4 :
2     if (activeConstraints[SP1_G4]) {
3         G4_counterpointMustBeInTheSameKey(home, this);
4     }
```

Listing 5.4: Activation control of the G4 constraints in the FirstSpeciesCounterpoint.cpp file

Each constraint is assigned a unique identifier in the constraints enumeration defined in `utilities.hpp`. This enumeration ensures a one-to-one mapping between constraints, their activation state in `activeConstraints`, and their textual label in the constraintNames vector used for debugging and MUS reporting. The solver can therefore dynamically activate or deactivate any subset of rules by simply adjusting the entries of the `activeConstraints` vector before posting the constraints.

The MUS extraction algorithm that I implemented is based on the first method described in the article *Towards Efficient MUS Extraction* [2]. In my implementation, this corresponds to the function `minimize()`, which is directly inspired by the original algorithm presented in the paper. The original version was designed to return only one Minimal Unsatisfiable Subset per call, whereas I adapted and extended it to compute all MUS associated with a given figure. This modification was essential to obtain a comprehensive view of the possible minimal subsets of constraints that lead to failure.

The MUS search itself is implemented in the function `findAllMUSes()`. The algorithm begins by enabling the full set of constraints and then explores subsets using a breadth-first strategy. For each candidate set:

1. The solver checks satisfiability with `is_unsat(current)`.
2. If the set is unsatisfiable, the function `minimize(current)` extracts a minimal unsatisfiable subset (MUS).

3. This MUS is stored, and new candidate sets are generated by removing one constraint at a time from it.

4. Each new set is added to the queue if it has not been visited already.

This iterative process guarantees that all MUSes are discovered. Finally, the indices of each MUS are mapped back to their human-readable names using `get_constraint_name(cons)` and printed to the console.

Listing 5.5: Pseudo-code of the all MUS finder algorithm used

```
//Main Function
Function findAllMUSes():
    //Initialize:
    cons_set ← {0, 1, 2, ..., N-1}  //all constraint
        indices
    mus_found ← empty set of sets
    visited ← empty set of sets
    queue ← empty queue
    Enqueue cons_set into queue
    Add cons_set to visited
    While queue is not empty:
        current_set ← dequeue from queue
        If is_unsat(current_set) == FALSE:
            //Current set is satisfiable, skip it and go to
                while loop
            Continue
        mus ← minimize(current_set)
        If mus not in mus_found:
            Add mus to mus_found
            #Generate new candidate sets by removing one
                constraint at a time
            For each constraint c in mus:
                next_set ← current_set without c
                If next_set not in visited:
                    Add next_set to visited
                    Enqueue next_set into queue
    //Output all MUSes found
    For each mus in mus_found:
        Print "MUS found:"
        For each constraint c in mus:
            Print get_constraint_name(c)


//Helper Functions
Function minimize(constraint_set):
    mus ← copy of constraint_set
    For each constraint c in mus:
        trial ← mus without constraint c
        If is_unsat(trial):
            mus ← trial
```

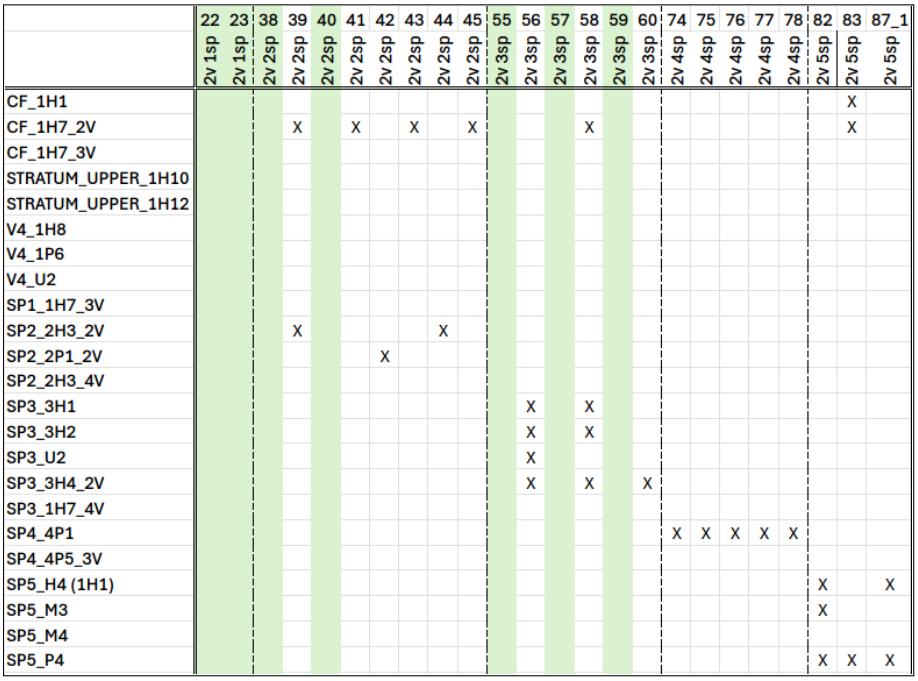
```
        Return mus

    Function is_unsat(constraint_set):
        Deactivate all constraints
        For each constraint c in constraint_set:
            Activate constraint c
        problem ← create new constraint satisfaction problem
        If problem has no solution:
            Return TRUE  //the set is unsatisfiable
        Else:
            Return FALSE //the set is satisfiable
```

Thanks to this algorithm, it was possible to identify exactly which constraints—or combinations thereof—block the validation of a figure.

5.3 Results and Interpretation

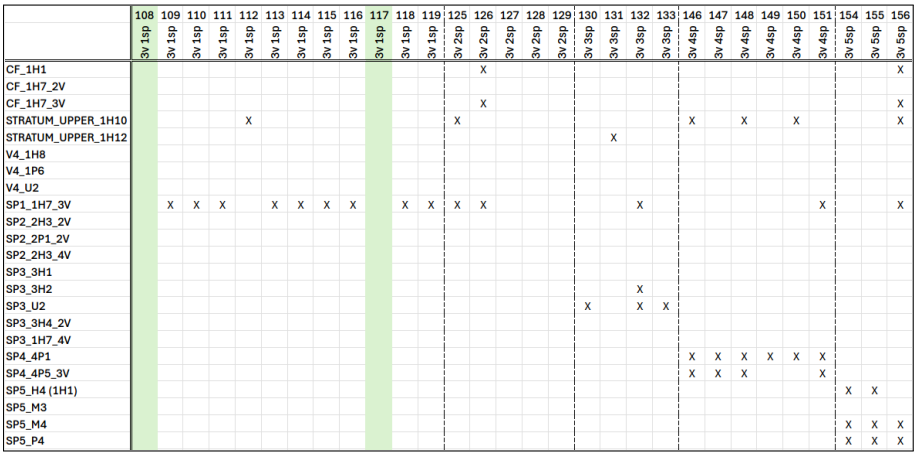
The first interesting result is that the MUS extraction algorithm never identified any subset containing more than one constraint. This suggests that, despite our apprehensions, the constraints in the system are largely independent, at least with respect to the Fuxian figures tested. In other words, solver failures can generally be attributed to a single rule, rather than a problematic interaction between multiple constraints. This independence both simplifies debugging and supports the modular structure of the current formalization.

To synthesize the results of the MUS-based testing approach, I compiled a table that presents, for each Fuxian figure tested, the specific constraint that prevented the solver from validating the example. This table was constructed directly from the output of the MUS extraction algorithm, which, for every figure, printed the corresponding Minimal Unsatisfiable Subset—in this case, always reduced to a single constraint, as previously discussed. By aggregating this output, I was able to mark, with an "X", each instance where a constraint was solely responsible for the solver's failure on a given figure. The resulting table provides a clear and systematic overview of which rules are too restrictive, incorrectly implemented, or potentially misaligned with Fux's treatise, and offers a basis for further revision and refinement of the constraint system.



	22	23	38	39	40	41	42	43	44	45	55	56	57	58	59	60	74	75	76	77	78	82	83	87_1
	2v 1sp	2v 1sp	2v 2sp	2v 2sp	2v 2sp	2v 2sp	2v 2sp	2v 2sp	2v 2sp	2v 2sp	2v 3sp	2v 3sp	2v 3sp	2v 3sp	2v 3sp	2v 3sp	2v 4sp	2v 4sp	2v 4sp	2v 4sp	2v 4sp	2v 5sp	2v 5sp	2v 5sp
CF_1H1																							X	
CF_1H7_2V				X		X		X		X				X									X	
CF_1H7_3V																								
STRATUM_UPPER_1H10																								
STRATUM_UPPER_1H12																								
V4_1H8																								
V4_1P6																								
V4_U2																								
SP1_1H7_3V																								
SP2_2H3_2V				X					X															
SP2_2P1_2V							X																	
SP2_2H3_4V																								
SP3_3H1												X		X										
SP3_3H2												X		X										
SP3_U2												X												
SP3_3H4_2V												X		X		X								
SP3_1H7_4V																								
SP4_4P1																	X	X	X	X	X			
SP4_4P5_3V																								
SP5_H4 (1H1)																						X		X
SP5_M3																						X		
SP5_M4																								
SP5_P4																						X	X	X

Figure 5.3: Derogated rules for two voice counterpoint



	108	109	110	111	112	113	114	115	116	117	118	119	125	126	127	128	129	130	131	132	133	146	147	148	149	150	151	154	155	156
	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 1sp	3v 2sp	3v 2sp	3v 2sp	3v 2sp	3v 2sp	3v 3sp	3v 3sp	3v 3sp	3v 3sp	3v 4sp	3v 4sp	3v 4sp	3v 4sp	3v 4sp	3v 4sp	3v 5sp	3v 5sp	3v 5sp
CF_1H1														X																X
CF_1H7_2V																														
CF_1H7_3V														X																X
STRATUM_UPPER_1H10					X								X									X		X		X			X
STRATUM_UPPER_1H12																			X											
V4_1H8																														
V4_1P6																														
V4_U2																														
SP1_1H7_3V		X	X	X		X	X	X	X		X	X	X	X						X							X			X
SP2_2H3_2V																														
SP2_2P1_2V																														
SP2_2H3_4V																														
SP3_3H1																														
SP3_3H2																				X										
SP3_U2																		X		X	X									
SP3_3H4_2V																														
SP3_1H7_4V																														
SP4_4P1																						X	X	X	X	X	X			
SP4_4P5_3V																						X	X	X			X			
SP5_H4 (1H1)																												X	X	
SP5_M3																														
SP5_M4																												X	X	X
SP5_P4																												X	X	X

Figure 5.4: Derogated rules for three voice counterpoint

	166	167	168	169	170	171	172	173	174	175	176	183	184	185	186	196	201	204
	4v 1sp	4v 1sp	4v 1sp	4v 1sp	4v 1sp	4v 1sp	4v 1sp	4v 2sp	4v 2sp	4v 2sp	4v 2sp	4v 3sp	4v 3sp	4v 3sp	4v 3sp	4v 4sp	4v 5sp	4v Xsp
CF_1H1																		
CF_1H7_2V																		
CF_1H7_3V																		
STRATUM_UPPER_1H10																		
STRATUM_UPPER_1H12																		
V4_1H8				X	X	X	X		X	X		X	X			X	X	X
V4_1P6								X										
V4_U2																	X	X
SP1_1H7_3V																		
SP2_2H3_2V																		
SP2_2P1_2V																		
SP2_2H3_4V										X								
SP3_3H1															X			
SP3_3H2																		
SP3_U2														X	X			
SP3_3H4_2V																		
SP3_1H7_4V												X	X					
SP4_4P1																X		X
SP4_4P5_3V																		
SP5_H4 (1H1)																	X	
SP5_M3																		
SP5_M4																	X	
SP5_P4																	X	

Figure 5.5: Derogated rules for four voice counterpoint

The tables above summarize the results of the MUS-based analysis performed on a selection of Fuxian figures. The first row of the tables list the figure numbers as they appear in the English edition of *Gradus ad Parnassum* by Alfred Mann [8]. The second row indicates the voice configuration and species of each example, following the format: number of voices (e.g., 2v, 3v, 4v) and species type (e.g., 1sp, 2sp, 3sp, ...).

Each row from the third line onward corresponds to a specific constraint. The constraint naming convention follows the pattern:
<File name>_<Constraint name>_<Voice number> with <Voice number> specified only if the rule is called in the specific number of voice constructor of the class related to the filename, otherwise, it means that the rule is called in the general constructor. This structure makes it easy to identify where each rule is implemented in the codebase and to which configuration it applies. For instance, a row labeled SP3_3H1_2V refers to the constraint 3.H1 implemented in the file ThirdSpeciesCounterpoint.cpp, specifically for the two-voice configuration.

The cells within the table are marked with an "X" when the corresponding figure violates the associated constraint. In other words, an "X" indicates that the MUS analysis identified this rule as a reason why the solver failed to accept the given Fuxian example. Cells left blank imply that the constraint was not part of any MUS for that figure. If a column is green, it means that this figure satisfiable all the rule applied to it.

This table provides a clear diagnostic overview of the mismatch between the implemented rule set and the original examples from Fux's treatise, and serves as a guide for identifying which constraints may require revision, relaxation, or refinement. The next chapter presents this diagnostic table and provides detailed commentary on selected examples.

Chapter 6

Rule-by-Rule Review of Validation Results

This chapter focuses on a detailed review of the constraints implemented in the solver, with particular attention to two categories of rules identified during the MUS-based analysis. First, I address the undocumented rules—constraints present in the codebase but not included or explained in the previous master thesis. These were labeled UX during testing (for "Undocumented") and require clarification regarding their purpose, validity, and relevance to Fuxian counterpoint. Second, I examine the derogated rules—that is, the constraints that were violated by one or more Fuxian figures during testing. For each of these, I analyze whether the failure was due to a flaw in implementation, an overly strict formalization, or a deeper misalignment with Fux's treatise.

The goal of this chapter is twofold: to bring transparency to the full set of constraints governing the solver's behavior, and to propose corrective actions for rules that either lack justification or have been shown to block valid Fuxian examples. This review is a key step toward improving the solver's accuracy and consistency with historical counterpoint practice.

Each rule will be presented using a two-part format. First, a brief sentence in English will describe the intended function or effect of the rule in musical terms, as far as it can be inferred from the code or behavior of the solver. This will be followed by the mathematical formalization of the constraint as implemented, to precisely capture how it operates within the system. This structure ensures both conceptual clarity and technical accuracy, allowing for a critical assessment of each rule's relevance and correctness.

6.1 Delayed Notes and Stratum

Many inconsistencies identified by MUS trace back to when a note is considered to "count" for rule checking. In Fuxian practice—especially in fourth species and parts of fifth—notes are often delayed across strong beats by ties; this is the classic syncopation (also called a ligature in English translations of Gradus ad

Parnassum). If the model evaluates intervals and motions strictly at onsets, a sustained (delayed) note can be misread as "absent," producing false violations; if it evaluates everywhere, rules can fire multiple times per beat and inflate conflicts.

To resolve this, we introduce the notion of stratum: a time grid aligned with arsis/thesis (weak/strong) positions on which harmonic and motion rules are evaluated. Delayed notes contribute their pitch at the appropriate stratum boundary (typically the strong beat they tie into), while purely rhythmic detail between strata is ignored by rules that are not meant to see it. This separation between raw events and stratum-aligned evaluation restores the intended reading of suspensions and syncopations, makes parallel/melodic checks well-defined across voices, and eliminates spurious MUS caused by timing ambiguities. The remainder of this section formalizes the stratum mapping and updates the affected rules accordingly.

Delayed Notes Definition

A note is delayed when its attack on the thesis is not a fresh onset but the continuation (tie) of the previous arsis from the preceding bar. This creates the classical preparation $\rightarrow$ suspension $\rightarrow$ resolution pattern: the tied note sounds on the thesis (often forming a dissonance against the other voice[s]) and resolves by step (joint) on the next arsis.

If the resolution is not achieved by a joint degree (i.e., no stepwise motion into the arsis), then the note is not considered a true delayed note. In such cases, the continuation (tie) of the previous arsis is instead treated as part of the harmony of the measure. This distinction was not correctly handled in the previous implementation, which is why the stratum assignment had to be revised. The updated approach, described in the next subsection, ensures that only proper suspensions with joint resolution are recognised as delayed notes, while non-joint continuations are incorporated directly into the harmony.

Stratum Implementation

In the previous implementation of the solver, strata were defined only at the level of measures. Concretely, only the first beat of each measure contained a note for each stratum, and all other beats in that measure were left undefined. Furthermore, the harmonic interval at each beat was systematically computed with respect to the lowest voice in the texture.

This design led to important limitations. For example, consider a two-voice counterpoint where the cantus firmus is placed in the upper voice and the counterpoint is written in the third species in the lower voice. If one wishes to evaluate the harmonic interval formed by the final note of the penultimate measure, the previous implementation incorrectly computed it between the cantus firmus and the *first* note of the penultimate measure of the third-species line. The actual last note of that measure, which should bear the harmonic weight at

that position, was ignored. As a result, several rules involving penultimate-beat checks could not be evaluated correctly.

To overcome this limitation, I redesigned the stratum assignment so that each beat is explicitly associated with the note actually sounding at that moment. For example, in the first species, where a whole note spans the full measure, every beat of the measure is assigned to the same note. In species with shorter rhythmic values (e.g., second, third, or fifth species), each beat is linked to the corresponding played note. In the case of delayed notes (suspensions), the assignment has also been corrected: the thesis is replaced, where appropriate, by the resolution or jointly treated with the arsis, according to the true harmonic function described in the previous subsection.

This new approach ensures that the strata reflect the actual musical reality at every beat. Harmonic intervals can now be computed consistently for all positions, including cases such as the penultimate note of a measure, which were previously mishandled. At the same time, the treatment of delayed notes has been aligned with their functional role, thereby integrating rhythmic and harmonic accuracy into a single coherent implementation.

6.2 Undocumented Rules

During the MUS-based analysis, I identified several constraints that were not documented or explained in the previous master thesis while present and active in the code. For clarity, I labeled them as UX (for "Undocumented") followed by a unique number. This section aims to review these undocumented rules, explore their possible purpose or origin, and assess whether they should be retained, modified, or removed to improve the solver's fidelity to Fuxian counterpoint.

STRATUM_1H12_UPPER (1.H12)

In three- and four-voice counterpoint, the final chord must not contain a minor third.
This rule applies to any species in three and four voice counterpoints.

$$H[0, m-1] \neq 3$$

This rule was first introduced into the formalisation by A. Lamotte[13] as **1.H12**, but was later omitted in the thesis of L. Cleenewerk and D. De Patoul[5]. Let's examine it again.

Fux consistently prefers the major third for the final chord, regardless of the mode of the counterpoint:

"If you mean the minor third don't you realize that is not capable of giving a sense of conclusion" [8, p. 80].

"Take a major third instead; for the minor third, being a more perfect consonance, is still less suitable for the end" [8, p.82].

In contrast, other composers such as Bitsch use the minor third freely when it better fits the mode.

In the present work, it has been explicitly reintroduced, as the aim is to remain as close as possible to Fux's style.

V4_U2 (1.H13)

For every pair of upper voices, the thesis note of each measure must never form a minor second interval.

$$\begin{gathered} \forall j \in [0, m-1], \quad s_1, s_2 \in \{b, c, d\} \\ H(s_1, s_2)[0, j] \neq 1 \end{gathered} \tag{6.1}$$

In traditional counterpoint, the minor second interval is considered a dissonance and is therefore prohibited in this position. Since this principle also applies to three-voice counterpoint, the rule has been extended accordingly and incorporated into its constraint set. Following the established rule nomenclature, it has been designated as rule **1.H13**.

From an implementation perspective, the rule was originally enforced only in `FourVoiceCounterpoint.cpp`. To ensure its scope covers three-voice counterpoint as well, it is now also explicitly called in `ThreeVoiceCounterpoint.cpp`.

SP3_U1 (3.M2)

No melodic interval between 9 and 11 semitones.
This rule applies to all third-species counterpoints.

$$\begin{gathered} j \in [0, m-2] \\ M[3, j] \notin \{9, 10, 11\} \end{gathered} \tag{6.2}$$

This rule is the same as **1.M2**, but applied to the third species. It is supported by Bitsch:

"27 - Melodic intervals. Rules concerning two successive notes.
A. The following are permitted:
- [...]
B. The following are prohibited:
- [...]
- intervals exceeding the minor sixth (except for the perfect octave)" [1] [9, p.18-19].

[1] Translated from French

In the previous implementation, this rule was applied only to the interval between the last note of a measure and the first note of the following measure. However, as Bitsch states, this restriction applies to *every* melodic interval. I therefore extend the rule as follows:

$$\begin{gathered} \forall i \in \mathcal{B}, \quad \forall j \in [0, m-2], \\ M[i,j] \notin \{9, 10, 11\} \end{gathered} \tag{6.3}$$

The implementation has been updated to follow this new formalisation, ensuring that all melodic intervals are checked rather than only those across barlines.

Despite its resemblance to rule **1.M2**, this rule has been added to the complete set of rules under the designation **3.M2**, since its formalisation is specific to the third species.

SP3_U2 (REMOVED)

Third note of the penultimate measure must be below the fourth one.
This rule applies to all third species counterpoints.

$$M[2, m-2] > 0 \tag{6.4}$$

This rule lacks an independent musical foundation; it appears to be an observational byproduct of other cadence-related constraints. Moreover, it does not hold when the third-species voice is in the bass. Even acknowledging that exception, the figure 185 still derogates the rule. Consequently, I removed **SP3_U2** from the implementation.

SP3_U3 (3.M3)

The second note of the penultimate measure must be more than a semitone away from the fourth note.
This rule applies to all third-species counterpoints.

$$M_{brut}[1, m-2] + M_{brut}[2, m-2] > 1 \tag{6.5}$$

This is primarily a convenience rule, intended to prevent the introduction of chromatic motion in the penultimate measure—a situation that should already be avoided by other rules. Nevertheless, since it has no negative impact, it has been retained in the present work and named **3.M3**.

SP4_U2 (REMOVED)

No chromatic motion between 3 consecutive notes.
This rule applies to all fourth species counterpoints.

$$\begin{gathered}\rho := \text{any note position} \\ [(M_{brut}[\rho+1] = +1 \implies M^2_{brut}[\rho] \neq +2) \\ \wedge \\ (M_{brut}[\rho+1] = -1 \implies M^2_{brut}[\rho] \neq -2)]\end{gathered} \tag{6.6}$$

At first glance, this rule seems intended to avoid monotony and encourage melodic variety. However, in fourth species the notes are grouped by pairs (thesis–arsis), which means the melodic interval between the arsis and the thesis of the same pair is always 0. As a result, the rule could only be meaningfully applied to the intervals between successive arsis notes.

Because there is no explicit trace of this rule in the theoretical sources, I decided to remove it from the implementation.

SP5_P4 (REMOVED)

In a sequence of musical intervals, if an interval is skipped (disjoint degree), the motion between the current and next interval must be contrary.
This rule applies to all fifth species counterpoints.

$$\begin{gathered}\rho := \text{any note position} \\ M[\rho] > 2 \implies P[\rho] = 0\end{gathered} \tag{6.7}$$

In the code, this rule is commented as "marcel's rule." We may therefore assume that it originates from the treatise of Noël Gallon and Marcel Bitsch. The closest formulation we could find is the following:

> "26 - Disjoint movement at the bar line.
> Disjoint movement should be avoided, particularly when moving from one bar to another, especially in quarter notes.
> Tolerance:
> At the bar line, disjoint intervals preceded by contrary motion are acceptable"[2] [9, p.18].

However, this is not a rule but a tolerance and explicitly concerns only motion at the barline, whereas the implementation in our solver generalises it to all positions. Such a generalisation is too strict for Fux's style, and for this reason the rule has been removed from the formalisation.

A commented-out version of this rule exists in the file `ThirdSpeciesCounterpoint.cpp`, but it has never been activated in the implementation.

[2] Translated from French

SP5_H3 (REMOVED)

In fifth-species counterpoint, the thesis note of the penultimate measure must not be consonant with the cantus firmus if it belongs to the fourth species (half note tied to form a syncopation).
This rule applies to all fifth-species counterpoints.

$$S[m-2,0] = 4 \implies H[m-2,0] \notin Cons \tag{6.8}$$

In the tested examples, this rule is never violated. However, an exception could occur if there are two distinct harmonies in the penultimate measure. For this reason, I decided to disable this rule in the current implementation.

SP5_M2 (5.M1)

No unison between two consecutive notes.

$$\begin{gathered} \forall j \in [0, m-2], \\ N[0,j] \neq N[1,j], \\ N[2,j] \neq N[3,j] \end{gathered} \tag{6.9}$$

In its original form, this rule checks only that, within each measure, the first and second notes are not in unison, and the third and fourth notes are not in unison.

While musically logical—helping to avoid monotony and encouraging melodic movement—this application is overly narrow. Ideally, unisons should be avoided between *any* pair of consecutive notes, not just between the first–second and third–fourth notes of each measure.

I therefore changed the implementation, leading to the following formalization:

$$\begin{gathered} \forall j \in [0, m-2], \\ N[0,j] \neq N[1,j], \\ N[1,j] \neq N[2,j], \\ N[2,j] \neq N[3,j] \\ N[3,j] \neq N[0,j+1] \end{gathered} \tag{6.10}$$

In the present work, this rule is retained and renamed **5.M1**.

SP5_M3 (REMOVED)

No more than minor sixth interval between arsis and thesis notes
This rule applies to all fifth species counterpoints.

$$\begin{gathered} \forall j \in [0, m-2] \\ H[2,j] - H[0,j] \notin \{9, 10, 11\} \end{gathered} \tag{6.11}$$

Like rule SP3_U1 (renamed in **3.M2**), this rule essentially mirrors **1.M2**, but in the context of the fifth species. Its theoretical justification is therefore the same as the one provided in the explanation of SP3_U1.

However, in this formalization the condition is expressed specifically as the interval between the arsis and the thesis of the same measure, whereas the original formulation applies more generally to any pair of consecutive notes. Since the rule is never derogated in practice, I have currently disabled it. Nevertheless, it could be reintroduced in the future, either with an explicit justification or by reformulating it in the same generalized way as the updated rule **3.M2**.

SP5_M4 (5.M2)

For each measure from the second to the second-to-last, if a note is part of the fourth species and the corresponding subsequent note is constrained, then these two notes must not be the same.
This rule applies to all fifth species counterpoints.

$$\begin{gathered} \forall j \in [1, m-2] \\ (S[0,j] = 4 \land N[2,j] \neq -1) \implies N[0,j] \neq N[2,j] \end{gathered} \tag{6.12}$$

The original formulation of this rule arose directly from the implementation. In code, the notion of a "constrained note" referred to an auxiliary array indicating, for each beat, whether a note was imposed by a given species. However, in practice a note was marked as "not constrained" only if no note existed at that beat. This definition was problematic in the context of the fifth species, since an arsis note must always be present.

In the implementation, this rule was labelled "no same syncopation". It seems intended to enforce melodic variety and avoid repetition in a manner similar to rule **4.M2** for the fourth species. However, unlike the fourth species, the fifth species allows quarter notes at the arsis beat, which are not covered by the original condition.

To correct this, I reformulated the rule as follows: *For each measure from the second to the second-to-last, if both the thesis and the arsis belong to the fourth species (i.e., tied half notes forming a syncopation), then these two notes must not be the same.*

$$\begin{gathered} \forall j \in [1, m-2] \\ (S[0,j] = 4 \land S[2,j] = 4) \implies N[0,j] \neq N[2,j] \end{gathered} \tag{6.13}$$

With this correction, no tested figure derogates the rule. Nevertheless, since rule **4.M2** is treated as a stylistic preference rather than a strict prohibition, it would be more consistent in future versions to reclassify this fifth-species analogue as a preference as well.

For the moment, however, the rule is included in the complete model set under the designation **5.M2**.

SP5_P3 (REMOVED)

If the third note of a measure forms a unison with the cantus firmus, and the cantus firmus is not the lowest voice, and the first note of the next measure is fourth species, then the harmonic interval at the start of the next measure must not be a minor or major second.
This rule applies to all fifth species counterpoints.

$$\begin{aligned} &species(p) = 5 \quad \forall j \in [0, m-2] \\ &H(p, cf)[2, j] = 0 \wedge \neg A(cf)[j] \wedge S(p)[0, j+1] = 4 \\ &\implies H(p)[0, j+1] \in \{1, 2\} \end{aligned} \tag{6.14}$$

This rule aims to prevent weak harmonic progressions after a unison, ensuring stronger voice leading in fifth species counterpoint. However, it is long, complex, and lacks a solid musical foundation. For these reasons, it has been removed from the implementation.

6.3 Derogated Rules

This section examines the rules that are not respected by the solver when tested on Fux's examples. For each derogated rule, we provide both a musical explanation of the rule and a discussion of why it fails in specific cases. To facilitate the connection between the theoretical rule and its implementation in the code, each subsection is titled using the name of the rule, followed by the different function or file names where the rule is invoked in the program. This naming convention ensures clarity and traceability between the formal definition and its practical application, helping to identify potential sources of misimplementation or oversights.

1.H1 (CF_1H1, SP1_1H1 and SP5_H4)

All notes on the downbeat are consonant with the notes (on the downbeat) of the lowest stratum.
This rule applies to all configurations.

$$\begin{aligned} &\forall j \in [0, m-1] \\ &H[0, j] \in Cons \end{aligned} \tag{6.15}$$

This rule is only violated in cases involving ligatures. In the second species, for instance, Fux occasionally uses a syncopation in the penultimate measure, which technically contradicts the rule. He briefly justifies this exception with the explanation:

> "[...] in the measures with the ligatures, either a bad unison or an empty sounding octave would have resulted from using plain half notes" [8, p.87-88].

This specific case occurs in figures 126, 127, and 128. A more precise justification is offered by Gallon and Bitsch, in their rule 17 for the second species:

> "The superior delay of the leading note in syncopated half notes is admitted in the penultimate measure for the variety of endings"[3] [9, p.15].

At present, syncopation in the second species at the penultimate measure is not supported by our solver. For this reason, figures 126, 127, and 128 were tested using two half notes instead of syncopation which explains the observed violations of the rule in those figures. Implementing this exception would be a useful improvement for future work.

This rule is also violated in figures 151 and 156 because a diminished fifth occurs in the penultimate measure. This exception is explained by Gallon and Bitsch in their rule 55:

> "Tolerance: From four voices onwards, a diminished fifth between the bass and an upper part is permitted, in the penultimate measure only, when the leading tone is delayed in the bass."[3] [9, p.30]

Interestingly, Fux applies this tolerance in three-voice compositions as well. This rule should therefore be explicitly added to the solver.

The other figures that derogated this rule (listed in the table of the previous chapter) now respect it thanks to the delayed note handling explained in the Stratum Implementation section.

1.H5 (V2_1H5, V3_1H5)

The voices cannot play the same note at the same time except in the first and last measure.
This rule applies for thesis notes to all species in two and three voices.

$$\forall p_1, p_2 \in \{cf, cp_1, cp_2\} \text{ with } p_1 \neq p_2 \quad \forall i \in \{0,1,2,3\} \quad \forall j \in [1, m-2] \\ N(p_1)[i,j] \neq N(p_2)[i,j] \tag{6.16}$$

In the previous formalization, this rule was incorrectly implemented so that no voice could play the same note (unison) as another at any beat. This is not the correct interpretation. The rule was introduced by T. Wafflar, who justified it with the statement from the Fux's treatise:

> "[...] it should never be used except at the beginning and end"[4] referring to the unison [4, p.62]

[3] Translated from French
[4] Translated from French

. However, Fux was specifically referring to the first species. Gallon and Bitsch clarify the rule in their rule 53 as follows:

> "Unison may be used on the weak beat or on the weak part of the beat. Unison is permitted on the strong beat in the first and last measures only.
> Tolerance: For five or more voices, unison is permitted on the strong beat during an exercise." [9, p.29]

Based on this, the rule should only be enforced on the first beat. With this correction, no figures currently derogate the rule. The formalization is now:

$$\begin{gathered}\forall p_1, p_2 \in \{cf, cp_1, cp_2\} \text{ with } p_1 \neq p_2 \quad \forall j \in [1, m-2] \\ N(p_1)[0, j] \neq N(p_2)[0, j]\end{gathered} \tag{6.17}$$

Moreover, although Gallon and Bitsch states that the rule can be relaxed for five voices, Fux already derogates it in a four-voice example (figure 166, second measure, between the second and third voices). For this reason, the rule is applied only in two- and three-voice compositions.

1.H7 (CF_1H7_2V, CF_1H7_3V, SP1_1H7_2V, SP1_1H7_3V, SP3_1H7_3V, SP3_1H7_4V)

1.H7 *In two voice composition, the harmonic interval of the penultimate note must be a major sixth or a minor third depending on whether or not the cantus firmus is the lowest stratum. In three voice composition, that harmonic interval must be either a minor third, a perfect fifth, a major sixth or an octave.*
This rule applies to all species.

`2V:`

$$\begin{gathered}\rho := max(positions(m)) - 1 \\ H[\rho] = \begin{cases} 9 & if\ A(cf)[0, m-2] \\ 3 & \text{other} \end{cases}\end{gathered} \tag{6.18}$$

`3V:`

$$H[0, m-2] \in \{0, 3, 7, 9\} \tag{6.19}$$

Although the statement of rule 1.H7 specifies that it applies to all configurations of two- and three-voice counterpoint, the implementation did not cover every case. Specifically:

- In two-voice counterpoint, the rule was not applied to the second, third, fourth, or fifth species.
- In three-voice counterpoint, it was not applied to the second, fourth, or fifth species.

The reason lies in the inheritance structure of the code: the rule was implemented only in the two-voice and three-voice constructors of the `FirstSpeciesCounterpoint` class, and in the three-voice constructor of the `ThirdSpeciesCounterpoint` class. Since the classes for the other species only call the generic constructor of the parent class, the specific constructors where 1.H7 is implemented were never invoked. As a result, the rule was never triggered in many derived species configurations. In future work, the implementation should be modified so that the constraint is called directly in `TwoVoiceCounterpoint.cpp` and `ThreeVoiceCounterpoint.cpp`, ensuring that it is enforced for all species.

A second issue came from the way harmonic intervals were computed in the previous implementation. Rule 1.H7 requires the harmonic interval of the penultimate note of the upper voice(s). However, harmonic intervals were always calculated against the first note of the lowest stratum in each measure. This caused errors whenever the lowest voice contained more than one note per measure (e.g., second or third species in the bass). For example, if the cantus firmus was the upper voice and the lower voice was in the third species, the intended interval is that between the final note of the cantus firmus in the penultimate measure and the final note of the lower voice in the same measure. Instead, the old implementation incorrectly measured it against the first note of the lower voice in that measure. With the new Strata implementation (see section 6.1), each beat is assigned to the note actually sounding, which allows the correct interval to be computed.

In the current implementation, two main reasons explain why some figures derogate the rule. First, Figures 56, 109, 110, 111, 113, 114, 115, 116, 118, 119, 125, and 132 use a major third instead of a minor third or major sixth. There is no strong reason to reject the major third when it belongs to the mode of the counterpoint. As Gallon and Bitsch note:

> "The penultimate measure may be harmonized, in both major and minor keys" [9, p.59].

The major third is therefore now authorized in the implementation, leading to the following formalisation:

`2V:`

$$\rho := \max(positions(m)) - 1$$

$$H[\rho] = \begin{cases} 9 & if\ A(cf)[0, m-2] \\ 3 \vee 4 & \text{otherwise} \end{cases} \tag{6.20}$$

`3V:`

$$H[0, m-2] \in \{0, 3, 4, 7, 9\} \tag{6.21}$$

Second, in Figures 151 and 156, when a fourth-species counterpoint appears in the lowest stratum, a diminished fifth occurs in the penultimate measure. This case is explicitly covered by a tolerance described by Gallon and Bitsch:

> "From four voices onward, a diminished fifth is permitted between the bass and an upper part, but only in the penultimate measure, when the leading tone is delayed in the bass"[5] [9, p.30, rule 54].

Fux himself applies this tolerance to three-voice compositions as well. This exception has not yet been implemented in the solver but should be considered in future work.

1.H8 (V3_1H8, V4_1H8)

The harmonic triad should be used as much as possible, and in four voices it should be used in its natural order with the octave if possible.
This rule applies to all thesis chords in three and four part composition, in all species.

This rule is implemented as a preference. For 4 voices, the cots depend of the case. We will start with the worst case (not using the harmonic triad) and work our way up to the best case (using the harmonic triad in its natural order, with the octave). By "natural order", Fux refers to the order given by the harmonic series [8, p.161]. We have the lower note then a fifth, an octave, and a third which is in fact a tenth. These are the second, third, fourth and fifth harmonics, respectively.

`3V:`

$$\begin{gathered}\forall j \in [0, m-1] \\ (H(b)[0,j] \notin 3,4) \vee (H(c)[0,j] \neq 7) \iff C_{prefer\_harmonic\_triad}[j] = 1\end{gathered} \tag{6.22}$$

`4V:`

`Worst case:` Not using the harmonic triad : either there is no third or fifth, or there is a note that does not belong to the harmonic triad.

$$\begin{aligned}&\forall j \in [0, m-1] \\ &(\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{3,4\})) \\ &\vee \ (\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{7\})) \\ &\vee \ \big(\exists h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \text{Cons}_{\text{h-triad}})\big) \\ &\implies C_{prefer\_harmonic\_triad}[j] = not\_harmonic\_triad\_cost\end{aligned} \tag{6.23}$$

`Better:` no octave, doubling the fifth.

[5] Translated from French

$$\forall j \in [0, m-1]$$
$$\left(\sum_{h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\}} \mathbf{1}_{h=7} = 2 \right) \Longrightarrow C_{prefer\_harmonic\_triad}[j] = double\_fifth\_cost \tag{6.24}$$

`Better yet:` no octave, doubling the third. In this case, we must also check that the two thirds are of the same nature (minor or major).

$$\begin{gathered}\forall j \in [0, m-1] \\ \left(\sum_{h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\}} \mathbf{1}_{h \in \{3,4\}} = 2 \right) \Longrightarrow (C_{prefer\_harmonic\_triad}[j] = double\_thirds\_cost) \\ (H_b[0,j] \in \{3,4\} \wedge H_c[0,j] \in \{3,4\}) \Longrightarrow (H_b[0,j] = H_c[0,j]) \\ (H_b[0,j] \in \{3,4\} \wedge H_d[0,j] \in \{3,4\}) \Longrightarrow (H_b[0,j] = H_d[0,j]) \\ (H_c[0,j] \in \{3,4\} \wedge H_d[0,j] \in \{3,4\}) \Longrightarrow (H_c[0,j] = H_d[0,j])\end{gathered} \tag{6.25}$$

`Even better:` octave used, but not in the natural order.

$$\begin{gathered}\forall j \in [0, m-1] \\ (H_b[0,j] = 7) \wedge (H_c[0,j] \in \{3,4\}) \wedge (H_d[0,j] = 0) \\ \quad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost \\ (H_b[0,j] \in \{3,4\}) \wedge (H_c[0,j] = 0) \wedge (H_d[0,j] = 7) \\ \quad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost \\ (H_b[0,j] \in \{3,4\}) \wedge (H_c[0,j] = 7) \wedge (H_d[0,j] = 0) \\ \quad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost \\ (H_b[0,j] = 0) \wedge (H_c[0,j] \in \{3,4\}) \wedge (H_d[0,j] = 7) \\ \quad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost \\ (H_b[0,j] = 0) \wedge (H_c[0,j] = 7) \wedge (H_d[0,j] \in \{3,4\}) \\ \quad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost\end{gathered} \tag{6.26}$$

`Best:` Harmonic triad with the octave, in natural order.

$$\begin{gathered}\forall j \in [0, m-1] \\ (H_b[j] = 7) \wedge (H_c[j] = 0) \wedge (H_d[j] \in \{3,4\}) \Longleftrightarrow C_{\text{prefer-harmonic-triad}}[j] = 0\end{gathered} \tag{6.27}$$

This rule is derogated only for 4 voices counterpoint.

Each voice is associated with an array of values representing the costs linked to harmonic intervals at each note position. However, in four-voice compositions, a problem arises: multiple costs can be assigned to the same variable, which is not permitted and leads to an error during execution. For example, a chord might include both a note outside the harmonic triad and two minor thirds, raising the question: which cost should be assigned to this chord - *not_harmonic_triad_cost* or *double_thirds_cost*? (see, for instance, figure 169, measure 2). To resolve this issue, I modified the formalization and the code to introduce a conditional check for cases involving doubling of fifths or thirds. Specifically, the cost is only applied if no more severe violation has already been detected, ensuring that only the most critical cost is assigned when multiple rule violations overlap.

Here is the new Formalization for `Better` and `Better yet` cases:

`Better:` no octave, doubling the fifth.

$$\begin{gathered}\forall j \in [0, m-1] \\ \left(\begin{array}{l}\left(\displaystyle\sum_{h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\}} \mathbf{1}_{h=7} = 2\right) \\ \wedge \neg(\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{3,4\})) \\ \wedge \neg\big(\exists h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \text{Cons}_{\text{h-triad}})\big)\end{array}\right) \\ \implies C_{prefer\_harmonic\_triad}[j] = double\_fifth\_cost\end{gathered} \tag{6.28}$$

`Better yet:` no octave, doubling the third. In this case, we must also check that the two thirds are of the same nature (minor or major).

$$\begin{gathered}\forall j \in [0, m-1] \\ \left(\begin{array}{l}\left(\displaystyle\sum_{h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\}} \mathbf{1}_{h \in \{3,4\}} = 2\right) \\ \wedge \neg(\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{7\})) \\ \wedge \neg\big(\exists h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \text{Cons}_{\text{h-triad}})\big)\end{array}\right) \\ \implies (C_{prefer\_harmonic\_triad}[j] = double\_thirds\_cost) \\ (H_b[0,j] \in \{3,4\} \wedge H_c[0,j] \in \{3,4\}) \implies (H_b[0,j] = H_c[0,j]) \\ (H_b[0,j] \in \{3,4\} \wedge H_d[0,j] \in \{3,4\}) \implies (H_b[0,j] = H_d[0,j]) \\ (H_c[0,j] \in \{3,4\} \wedge H_d[0,j] \in \{3,4\}) \implies (H_c[0,j] = H_d[0,j])\end{gathered} \tag{6.29}$$

Here is the previous code:

```
1    void H8_4v_preferHarmonicTriad(Home home, IntVarArray triadCostArray, Stratum
       ↪  * upper1, Stratum* upper2, Stratum* upper3){
```

```
        for(int i = 0; i < triadCostArray.size(); i++){

            IntVar H_b = upper1->getHInterval()[i*4];
            IntVar H_c = upper2->getHInterval()[i*4];
            IntVar H_d = upper3->getHInterval()[i*4];


            BoolVar H_b_is_third    = expr(home, H_b==MINOR_THIRD || H_b==MAJOR_THIRD);
            BoolVar H_b_is_fifth    = expr(home, H_b==PERFECT_FIFTH);
            BoolVar H_b_is_octave   = expr(home, H_b==UNISSON);

            BoolVar H_c_is_third    = expr(home, H_c==MINOR_THIRD || H_c==MAJOR_THIRD);
            BoolVar H_c_is_fifth    = expr(home, H_c==PERFECT_FIFTH);
            BoolVar H_c_is_octave   = expr(home, H_c==UNISSON);

            BoolVar H_d_is_third    = expr(home, H_d==MINOR_THIRD || H_d==MAJOR_THIRD);
            BoolVar H_d_is_fifth    = expr(home, H_d==PERFECT_FIFTH);
            BoolVar H_d_is_octave   = expr(home, H_d==UNISSON);


            // worst case : not a harmonic triad with at least a third and a fifth
            BoolVar no_fifth_or_no_third = expr(home, H_b_is_third + H_c_is_third + H_d_is_third == 0 || H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 0);
            BoolVar note_outside_harmonic_triad = expr(home, H_b_is_third + H_b_is_fifth + H_b_is_octave == 0 || H_c_is_third + H_c_is_fifth + H_c_is_octave == 0 || H_d_is_third + H_d_is_fifth + H_d_is_octave == 0);

            rel(home, (note_outside_harmonic_triad || no_fifth_or_no_third) >> (triadCostArray[i] == not_harmonic_triad_cost));

            // now we are left with only combinations with at a third, a fifth, and another note of the harmonic triad.
            // Doubling the fifth
            rel(home, expr(home, H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 2) >> (triadCostArray[i] == double_fifths_cost));

            // Doubling the third, and ensure it is the same type of third (not a major and a minor)
            rel(home, expr(home, H_b_is_third + H_c_is_third + H_d_is_third == 2) >> (triadCostArray[i] == double_thirds_cost));
            rel(home, (H_b_is_third && H_c_is_third) >> (H_b == H_c));
            rel(home, (H_b_is_third && H_d_is_third) >> (H_b == H_d));
            rel(home, (H_c_is_third && H_d_is_third) >> (H_c == H_d));

            // triad with octave
            rel(home, (H_b_is_fifth && H_c_is_third && H_d_is_octave) >> (triadCostArray[i] == triad_with_octave_cost)); // 5 3 8
            rel(home, (H_b_is_third && H_c_is_octave && H_d_is_fifth) >> (triadCostArray[i] == triad_with_octave_cost)); // 3 8 5
            rel(home, (H_b_is_third && H_c_is_fifth && H_d_is_octave) >> (triadCostArray[i] == triad_with_octave_cost)); // 3 5 8
            rel(home, (H_b_is_octave && H_c_is_third && H_d_is_fifth) >> (triadCostArray[i] == triad_with_octave_cost)); // 8 3 5
            rel(home, (H_b_is_octave && H_c_is_fifth && H_d_is_third) >> (triadCostArray[i] == triad_with_octave_cost)); // 8 5 3

            // Best case : 5 8 3
            rel(home, (H_b_is_fifth && H_c_is_octave && H_d_is_third) >> (triadCostArray[i] == 0));

        }
    }
```

Listing 6.1: Previous implementation of 1.H8 for four voices configuration

In which I modified line 30 in the previous frame to become:

```
1     rel(home, expr(home, (H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 2) && !
        ↪  note_outside_harmonic_triad && !no_fifth_or_no_third) >> (
        ↪  triadCostArray[i] == double_fifths_cost));
```

Listing 6.2: Modification of double fifths case in 1.H8 implementation for four voices configuration

And line 33:

```
1     rel(home, expr(home, (H_b_is_third + H_c_is_third + H_d_is_third == 2) && !
        ↪  note_outside_harmonic_triad && !no_fifth_or_no_third) >> (
        ↪  triadCostArray[i] == double_thirds_cost));
```

Listing 6.3: Modification of double thirds case in 1.H8 implementation for four voices configuration

1.H10 (STRATUM_1H10_UPPER)

In three voice composition, tenths are prohibited in the last chord.
This rule applies to all species in three voices.

$$H_{brut}[0, m-1] > 12 \implies H[0, m-1] \notin \{3, 4\}$$

Fux remarks:

> "One feels that the degree of perfection and repose which is required of the final chord does not become sufficiently positive with this imperfect consonance" speaking about a tenth [8, p.77].

He then advises:

> "Therefore, it is advisable to omit the third altogether" referring to the final measure [8, p.80].

However, he later nuances this directive by adding:

> "What I told you regarding the omission of the third holds true only in cases in which it is possible"[8, p.82]

This indicates that the rule is not absolute and may be disregarded in certain contexts, making it a good candidate to be reformulated as a preference with an associated cost in the solver.

As already said in the section 6.2 about the rule 1.H12, Fux consistently favors the major third for the final chord, regardless of the mode of the counterpoint which is not the case of the some other composers.

In the present work, this rule has been disabled in its strict form but could be reintroduced as a soft preference, thereby leaving the final choice to the composer. However, since Fux himself does not always respect this rule (e.g., figure 112), I ultimately decided to suspend it.

1.H13 (V4_U2)

For every pair of upper voices, the thesis note of each measure must never form a minor second interval.
This rule applies to all species in three- and four-voice counterpoint.

The unknown rule **V4_U2** was already introduced in section 6.2, where its formalization is presented.

In the previous implementation, apparent violations of this rule occurred because delayed notes were not handled correctly. The implementation has now been updated to use the revised *Stratum* framework. As a result, the rule is now enforced consistently without spurious violations.

2.H2 (SP2_2H2)

Arsis harmonies cannot be dissonant, except in the case of a diminution.
This rule applies to second-species counterpoints for all numbers of voices.

$$\forall j \in [0, m-1], \forall p : \text{species}(p) = 2$$

$$IsDim(p)[j] = \begin{cases} \top & \text{if } M^2(p)[0,j] \in \{3,4\} \wedge M^1(p)[0,j] \in \{1,2\} \wedge M^1(p)[2,j] \in \{1,2\} \\ \bot & \text{otherwise} \end{cases}$$

$$\forall j \in [0, m-1] \quad \neg IsCons(p)[2,j] \implies IsDim(p)[j]$$

In the updated implementation, this rule is no longer violated. When the second species is in an upper voice, past violations were due to incorrect handling of delayed notes—a problem now resolved through the improved *Stratum* mechanism.

2.H3 (SP2_2H3_2V, SP2_2H3_3V, SP2_2H3_4V)

When composing in two voices, 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.
This rule applies to all two voice configurations in which the counterpoint is of the second species.

$$H[0, m-2] \in \{0, 7, 8, 9\}$$

$$\text{penulthesis}_{cost} = \begin{cases} \text{cost}_{\text{penulthesis}} & \text{if } H[0, m-2] \neq 7 \\ 0 & \text{otherwise} \end{cases}$$

In its textual explanation, the rule is stated as applying only to two-voice counterpoint. However, in the previous implementation, it was applied to all configurations, regardless of the number of voices. This inconsistency has been corrected by removing the constraint for counterpoint with more than two voices.

3.H1 (SP3_3H1)

If five notes follow each other by joint degrees in the same direction, then the harmonic interval of the third note must be consonant
This rule applies in all configurations, to third species counterpoints.

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

The current implementation of this rule is overly strict. Immediately after introducing it, Fux specifies an important exception:

> "This does not hold if, firstly, the second and the fourth notes are consonant, in which case the third note may be dissonant." [Mann, p. 50]

This exception is not currently handled in the model, which results in the rule being applied in situations where Fux would have allowed a violation. For this reason, the rule has been disabled in the present version. It should be revised to incorporate Fux's stated exception before being reintroduced into the solver.

3.H2 (SP3_3H2)

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
This rule applies in all configurations, to third species counterpoints.

$$\forall j \in [0, m-1]$$
$$IsCons[2, j] \vee (IsCons[1, j] \wedge IsCons[3, j] \wedge IsDim[j])$$
where $IsDim[j] = \top$ when the 3^{rd} note of the measure j is a diminution.

It translates a very important concept in counterpoint (which was already present in the second species) : all dissonances are surrounded by consonances. But this rule shouldn't be applied to the penultimate measure where the second and the third can be dissonant with the first and the fourth consonant. The formalization is slightly modified to become:

$$\forall j \in [0, m-3]$$
$$IsCons[2, j] \vee (IsCons[1, j] \wedge IsCons[3, j] \wedge IsDim[j])$$
where $IsDim[j] = \top$ when the 3^{rd} note of the measure j is a dimension. (6.30)

3.H4 (SP3_3H4)

In two voice composition, in the penultimate measure, if the counterpoint is the lowest stratum, then the harmonic interval of the first note should be a minor third. This rule applies in two voices when the counterpoint is of the third species.

$$A(cp)[m-2] \implies H[0, m-2] = 3$$

In the previous implementation, this rule was placed in `ThirdSpeciesCounterpoint.cpp`, even though it is a harmonic constraint that should be applied to the *cantus firmus*. Since the harmonic interval is computed from the lowest voice (here, the third-species voice), the rule needed to be moved. I removed the original implementation from `ThirdSpeciesCounterpoint.cpp` and created a new version in `TwoVoiceCounterpoint.cpp` that check that the part is a third species counterpoint to correctly compute the harmonic interval from the cantus firmus and ensure the rule only applies to two-voice counterpoint.

The rule remains violated in Figure 58. Nevertheless, the constraint is musically coherent and has therefore been retained in the implementation. Although it could be reconsidered as a preference in the future, in my view it should remain a strict rule.

1.P6 (V3_1P6, V4_1P6)

It is prohibited that all parts move in the same direction.
This rule applies to all configurations in three or four voices.

$$\forall j \in [0, m-1] : \bigvee_{p \in \{cf, cp_1, cp_2, cp_3\}} P(p)[0, j] \in \{0, 1\}$$

In the previous version of the code, the same implementation of the rule was used for both three- and four-voice configurations. As a result, when the rule was called in four-voice counterpoint, it only checked the motion of three voices instead of all four. To correct this, I created a dedicated version of the rule for four voices, which is now called whenever the configuration involves four voices.

During testing, I also discovered that the melodic interval of the lowest stratum was incorrectly computed in Figure 173. This error caused the solver to report a derogation of the rule. The issue was traced to the fact that the melodic interval of the lowest stratum wasn't well calculated. I fixed this bug, and with the correction, the rule is now applied correctly.

2.P1 (SP2_2P1_2V, SP2_2P1_3V)

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.
This rule applies to all counterpoints of the second species.

Note: This rule is used to evaluate all other motion rules in the context of second-species counterpoint. The special case of the battuta octave (rule 1.P3) always perceives motion from the last note of the previous measure, but this is already explicitly formulated in its definition. As such, it does not affect this rule and requires no reformulation.

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

In the code, this constraint redefined rule **1.P1** in order to correctly account for motion in the second species. In other words, when the leap from the thesis to the arsis exceeds a third, the harmonic evaluation is based on the arsis rather than the thesis, so as to prevent approaching a perfect consonance by direct motion.

However, the original implementation did not correctly reflect this principle: motion was mistakenly computed from *arsis to arsis*, whereas the correct computation must be from *arsis to thesis*. I corrected this error, and the rule is no longer derogated in the tested figures.

For clarity, I slightly reformulated the statement of the rule in the complete set of rules as follows: *"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 to the next thesis note."*

4.P1 (SP4_4P1, SP5_4P1)

Dissonant harmonies must be followed by the next lower consonant harmony.
This rule applies in all configurations to fourth species counterpoints.

$$\begin{gathered} \forall j \in [1, m-1] \\ \neg isCons[0, j] \implies M_{brut}[0, j] \in \{1, 2\} \end{gathered} \tag{6.31}$$

In the implementation, this rule was also applied in the fifth species, even though this was not explicitly stated in the original description. Nevertheless, the extension to the fifth species proved both logically consistent and error-free, since it applies naturally to tied half notes that create syncopation. For the fifth species, however, the mathematical formalisation differs slightly, as the rule must only be applied when the note is part of a fourth-species pattern (tied half notes):

$$\begin{gathered} \forall j \in [1, m-1] \\ \neg isCons[0, j] \wedge S[0, j] = 4 \implies M^2_{brut}[0, j] \in \{1, 2\} \end{gathered} \tag{6.32}$$

The use of M^2 is justified because, in the fifth species, every beat can potentially contain a note. In this case, it is necessary to check the melodic interval specifically between the thesis and the arsis note.

In the previous implementation, this constraint was not applied correctly for fourth-species counterpoint due to an incorrect index being used for the melodic interval. Even after correcting this index, the rule continued to be derogated whenever the fourth-species voice appeared in the lowest stratum. This was because harmonic intervals in fourth species were computed from the arsis (resolution) rather than the thesis (suspension), which incorrectly classified many thesis notes as dissonant.

This problem has now been resolved with the new *Strata* implementation, which assigns each beat to the note that carries the actual harmonic weight. By correctly handling delayed notes, harmonic intervals are now computed consistently, and the rule is enforced properly in both fourth- and fifth-species counterpoint.

4.P5 (SP4_4P5_3V, SP4_4P5_4V)

In three and four part composition, stationary movement in the bass implies dissonance in the fourth species part.
This rule applies to all fourth species counterpoints in three or four voice configurations.

$$\forall j \in [0, m-1] :$$
$$M(a)[0, j] \neq 0 \Leftrightarrow H[2, j] \in Cons$$
$$M(a)[0, j] = 0 \Leftrightarrow H[2, j] \in Dis$$

In the previous thesis, this constraint was deactivated in the code with the comment: "after careful testing, Fux does not seem to follow this rule in many of his examples." My own analysis confirms this conclusion.

Evidence: In Figures 146 and 147 (measure 6), the bass line shows stationary motion, yet the arsis note of the fourth-species voice is consonant.

These examples demonstrate that Fux explicitly tolerates stationary motion in the bass while maintaining consonance in the fourth-species line, contrary to the stricter stance of some musicologists. As a result, this rule should not be treated as a hard constraint. Instead, it would be more appropriate to reformulate it as a preference, implemented as a soft constraint with an associated cost, so that the solver favours but does not strictly enforce this behaviour.

5.M2 (SP5_M4)

For each measure from the second to the second-to-last, if both the thesis and the arsis belong to the fourth species (i.e., tied half notes forming a syncopation), then these two notes must not be the same.

This rule has already been discussed in section 6.2, where both the problem and its resolution were detailed.

To summarise, the original implementation incorrectly relied on a "constrained note" array, which did not properly account for the arsis in the fifth species. I reformulated the rule to explicitly require that, in measures where both thesis and arsis are tied half notes, these notes must not be identical. With this correction, no tested figure derogates the rule. For the moment, the rule is retained in the complete model set under the designation **5.M2**, though it may be reconsidered as a stylistic preference in future work.

Chapter 7

Full Validated Model of Fux Counterpoint

This chapter presents the complete set of contrapuntal rules implemented in the solver. The content is based on the work of Lamotte (2024), Wafflard (2023), and Sprockeels (2022), revised and expanded by Cleenewerk & de Patoul (2024). The goal here is to provide a unified and consistent reference: rules have been reorganised, reformulated for clarity, and their scopes made explicit. In some cases, redundant or outdated rules from previous versions were removed, while others were refined to align more closely with the intended Fuxian style.

Each rule is identified by a unique code. The first number indicates the species concerned (0 = general rule, 1 = first species, etc.), the letter indicates the category (H = harmonic, M = melodic, P = motion, R = rhythmic), and the second number is the index of the rule within that category. When a rule applies to several species, the code reflects the main one but its scope is clarified in the text.

For a detailed explanation of these rules, their underlying logic, and their origin in *Gradus ad Parnassum*, the reader may refer to the corresponding theses: T. Wafflard for two-voice rules[18], A. Lamotte for three-voice rules[13] and L. Cleenewerk & D. de Patoul for four-voice rules[5].

7.1 General Rules (G)

The rules **G1** to **G3** do not have explicit mathematical formalizations, since they are not implemented as separate constraints but are instead embedded directly into the structure of the program itself. They apply universally, regardless of the number of voices or the species considered.

G1

Harmonic intervals are calculated from the lowest note.

G2

The number of measures of the counterpoints must be the same as the number of measures of the cantus firmus.

$$m_{\text{cp}} = m_{\text{cf}} \tag{7.1}$$

This guarantees structural alignment between the cantus firmus and the counterpoint.

G3

The counterpoints must have the same time signature and the same tempo as the cantus firmus.

G4

The counterpoints must be in the same key as the cantus firmus.

G5

The range of the counterpoints must be consistent with the instruments used.

This prevents writing notes that would be impractical or impossible to perform.

G6

Chromatic melodies are forbidden for two and three voice composition, and to be avoided for four voice composition.

`2V & 3V:`

$$\forall \rho \in positions(m-2) \quad \begin{cases} (M_{brut}[\rho] = 1 \wedge M_{brut}[\rho+1] = 1) \iff \bot \\ (M_{brut}[\rho] = -1 \wedge M_{brut}[\rho+1] = -1) \iff \bot \end{cases} \tag{7.2}$$

For four voices, the preference of not having chromatic melodies is already taken into account by the cost of using borrowed notes.

G7

Melodic intervals should be small.

$$\forall \rho \in positions(m-1)$$
$$M_{deg\_costs}[\rho] = \begin{cases} cost_{secondMdeg} & \text{if } M[\rho] \in \{0,1,2\} \\ cost_{thirdMdeg} & \text{if } M[\rho] \in \{3,4\} \\ cost_{fourthMdeg} & \text{if } M[\rho] = 5 \\ cost_{tritoneMdeg} & \text{if } M[\rho] = 6 \\ cost_{fifthMdeg} & \text{if } M[\rho] = 7 \\ cost_{sixthMdeg} & \text{if } M[\rho] \in \{8,9\} \\ cost_{seventhMdeg} & \text{if } M[\rho] \in \{10,11\} \\ cost_{octaveMdeg} & \text{if } M[\rho] = 12 \end{cases} \tag{7.3}$$

7.2 Constraints of the First Species

Harmonic Rules

1.H1

All notes on the downbeat are consonant with the notes (on the downbeat) of the lowest stratum.
This rule applies in all configurations.

$$\begin{gathered} \forall j \in [0, m-1] \\ H[0,j] \in Cons \end{gathered} \tag{7.4}$$

1.H2

In two voices, the first harmony must be a perfect consonance.
This rule applies to all species in two voices.

$$H[0,0] \in Consp \tag{7.5}$$

1.H3

In two voice composition, the last harmonic interval must be a perfect consonance. When composing for three or four voices, the last chord should be composed only of notes of the harmonic triad.
This rule applies in all configurations.

$$\begin{aligned} \texttt{2V:} \quad & H[0, m-1] \in Consp \\ \texttt{3V \& 4V:} \quad & \forall s \in \{b, c, d\}, \quad H[s, m-1] \in Consh_{\text{triad}} \end{aligned} \tag{7.6}$$

1.H4

The key tone is tuned according to the first note of the cantus firmus.
This rule applies in all configurations.

$$\begin{aligned} N(a)[0,0] \mod 12 &= N(cf)[0,0] \mod 12 \\ N(a)[0,m-1] \mod 12 &= N(cf)[0,0] \mod 12 \end{aligned} \tag{7.7}$$

1.H5

The voices cannot play the same note at the same time except in the first and last measure
This rule applies to thesis notes in all species with two and three voices.

$$\begin{gathered} \forall p_1, p_2 \in \{cf, cp_1, cp_2\} \text{ with } p_1 \neq p_2 \quad \forall i \in \{0,1,2,3\} \quad \forall j \in [1, m-2] \\ N(p_1)[i,j] \neq N(p_2)[i,j] \end{gathered} \tag{7.8}$$

1.H6

On thesis notes, imperfect consonances are preferred to perfect ones; fifths preferred to octaves.
This rule applies in all configurations.

$$\begin{gathered} \forall j \in [0, m-1] \\ Pconscosts[j] = \begin{cases} cost_{Pcons\_oct} & \text{if } H[0,j] = 0, \\ cost_{Pcons\_fifth} & \text{if } H[0,j] = 7, \\ 0 & \text{otherwise.} \end{cases} \end{gathered} \tag{7.9}$$

1.H7

In two voice composition, the harmonic interval of the penultimate note must be a major sixth or a third (minor or major) depending on whether or not the cantus firmus is the lowest stratum. In three voice composition, that harmonic interval must be either a minor third, a major third, a perfect fifth, a major sixth or an octave.
This rule applies to the penultimate *note* for all species with two and three voices

`2V:`

$$\rho := \max(positions(m)) - 1$$
$$H[\rho] = \begin{cases} 9 & if\ A(cf)[0, m-2] \\ 3 \vee 4 & \text{otherwise} \end{cases} \tag{7.10}$$

`3V:`

$$H[0, m-2] \in \{0, 3, 4, 7, 9\} \tag{7.11}$$

1.H8

The harmonic triad should be used as much as possible, and in four voices it should be used in its natural order with the octave if possible.
This rule applies to thesis chords for three and four voices composition.

`3V:`

$$\begin{gathered} \forall j \in [0, m-1] \\ (H(b)[0, j] \notin 3, 4) \vee (H(c)[0, j] \neq 7) \iff C_{prefer\_harmonic\_triad}[j] = 1 \end{gathered} \tag{7.12}$$

`4V:`

`Worst case:` Not using the harmonic triad : either there is no third or fifth, or there is a note that does not belong to the harmonic triad.

$$\begin{aligned} &\forall j \in [0, m-1] \\ &\big(\forall h \in \{H_b[0, j], H_c[0, j], H_d[0, j]\} \quad (h \notin \{3, 4\})\big) \\ &\vee\ \big(\forall h \in \{H_b[0, j], H_c[0, j], H_d[0, j]\} \quad (h \notin \{7\})\big) \\ &\vee\ \big(\exists h \in \{H_b[0, j], H_c[0, j], H_d[0, j]\} \quad (h \notin \text{Cons}_{\text{h-triad}})\big) \\ &\implies C_{prefer\_harmonic\_triad}[j] = not\_harmonic\_triad\_cost \end{aligned} \tag{7.13}$$

`Better:` no octave, doubling the fifth.

$$\begin{gathered} \forall j \in [0, m-1] \\ \left(\begin{aligned} &\left(\sum_{h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\}} \mathbf{1}_{h=7} = 2 \right) \\ &\wedge \neg(\forall h \in \{H_b[0, j], H_c[0, j], H_d[0, j]\} \quad (h \notin \{3, 4\})) \\ &\wedge \neg\big(\exists h \in \{H_b[0, j], H_c[0, j], H_d[0, j]\} \quad (h \notin \text{Cons}_{\text{h-triad}})\big) \end{aligned} \right) \\ \implies C_{prefer\_harmonic\_triad}[j] = double\_fifth\_cost \end{gathered} \tag{7.14}$$

`Better yet:` no octave, doubling the third. In this case, we must also check that the two thirds are of the same nature (minor or major).

$$
\begin{aligned}
&\forall j \in [0, m-1] \\
&\left(\begin{array}{l}
\left(\sum_{h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\}} \mathbf{1}_{h \in \{3,4\}} = 2\right) \\
\wedge \neg(\forall h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \{7\})) \\
\wedge \neg\big(\exists h \in \{H_b[0,j], H_c[0,j], H_d[0,j]\} \quad (h \notin \text{Cons}_{\text{h-triad}})\big)
\end{array}\right) \\
&\qquad \Longrightarrow (C_{prefer\_harmonic\_triad}[j] = double\_thirds\_cost) \\
&(H_b[0,j] \in \{3,4\} \wedge H_c[0,j] \in \{3,4\}) \Longrightarrow (H_b[0,j] = H_c[0,j]) \\
&(H_b[0,j] \in \{3,4\} \wedge H_d[0,j] \in \{3,4\}) \Longrightarrow (H_b[0,j] = H_d[0,j]) \\
&(H_c[0,j] \in \{3,4\} \wedge H_d[0,j] \in \{3,4\}) \Longrightarrow (H_c[0,j] = H_d[0,j])
\end{aligned}
\tag{7.15}
$$

`Even better:` octave used, but not in the natural order.

$$
\begin{aligned}
&\forall j \in [0, m-1] \\
&(H_b[0,j] = 7) \wedge (H_c[0,j] \in \{3,4\}) \wedge (H_d[0,j] = 0) \\
&\qquad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost \\
&(H_b[0,j] \in \{3,4\}) \wedge (H_c[0,j] = 0) \wedge (H_d[0,j] = 7) \\
&\qquad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost \\
&(H_b[0,j] \in \{3,4\}) \wedge (H_c[0,j] = 7) \wedge (H_d[0,j] = 0) \\
&\qquad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost \\
&(H_b[0,j] = 0) \wedge (H_c[0,j] \in \{3,4\}) \wedge (H_d[0,j] = 7) \\
&\qquad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost \\
&(H_b[0,j] = 0) \wedge (H_c[0,j] = 7) \wedge (H_d[0,j] \in \{3,4\}) \\
&\qquad \Longleftrightarrow (C_{prefer\_harmonic\_triad}[j] = triad\_with\_octave\_cost
\end{aligned}
\tag{7.16}
$$

`Best:` Harmonic triad with the octave, in natural order.

$$
\begin{aligned}
&\forall j \in [0, m-1] \\
&(H_b[j] = 7) \wedge (H_c[j] = 0) \wedge (H_d[j] \in \{3,4\}) \Longleftrightarrow C_{\text{prefer-harmonic-triad}}[j] = 0
\end{aligned}
\tag{7.17}
$$

1.H11

Octaves should be preferred over unisons.
This rule applies to all configurations.

This rule is taken into account by **1.H5** for two and three voices, and by **1.H8** for four voices.

1.H12

In three- and four-voice counterpoint, the final chord must not contain a minor third.
This rule applies to any species in three and four voice counterpoints.

$$H[0, m-1] \notin 3 \tag{7.18}$$

1.H13

For every pair of upper voices, the thesis note of each measure must never form a minor second interval.
This rule is applied to every species of counterpoint in three and four voices configuration.

$$\begin{gathered} \forall j \in [0, m-1], \quad s_1, s_2 \in \{b, c, d\} \\ H(s_1, s_2)[0, j] \neq 1 \end{gathered} \tag{7.19}$$

Melodic Rules

1.M2

Intervals cannot exceed a minor sixth interval, but octave leaps are allowed in three and four voices
This rule applies to all first-species counterpoints.

`2V:`

$$\begin{gathered} \forall j \in [0, m-2] \\ M[0, j] \leq 8 \end{gathered} \tag{7.20}$$

`3V & 4V:`

$$\begin{gathered} \forall j \in [0, m-2] \\ \big(M[0, j] \leq 8 \ \vee \ M[0, j] = 12\big). \end{gathered} \tag{7.21}$$

1.M4

The notes of each part should be as diverse as possible.
This rule applies to all configurations in three and four voices.

$$\begin{gathered} \forall p \in \{cp1, cp2, cp3\}, \ \forall j \in [0, m-1], \ \forall k \in [j+1, \min(j+3, m-1)] : \\ \big(N(p)[0, j] = N(p)[0, j * k]\big) \iff C_{variety}[j + m \cdot k] = 2. \end{gathered} \tag{7.22}$$

Motion Rules

1.P1 & 3.P2

Reaching a perfect consonance by direct motion is forbidden in two voices, and should be avoided when composing for three and four voices. In four voice composition, it is to be tolerated between inner voices rather than outer voices.
This rule applies to all configurations.

2V:

$$\begin{gathered}\forall j \in [0, m-2] \\ H[0, j+1] \in Consp \;\Rightarrow\; P[0, j] \neq 2\end{gathered} \tag{7.23}$$

3V:

$$\begin{gathered}\forall j \in [0, m-2] \\ \big(P[0, j] = 2 \wedge H[0, j+1] \in Cons_p\big) \iff C_{direct\_move\_to\_p\_cons}[j] = 8\end{gathered} \tag{7.24}$$

4V:

$$\begin{aligned}&P(b)[0, j] = 2 \wedge H(b)[0, j] \in Cons_p \wedge H(b)[0, j+1] \in Cons_p \iff C_{direct\_move\_to\_p\_cons}[j] = 2,\\ &P(c)[0, j] = 2 \wedge H(c)[0, j] \in Cons_p \wedge H(c)[0, j+1] \in Cons_p \iff C_{direct\_move\_to\_p\_cons}[j] = 2,\\ &P(d)[0, j] = 2 \wedge H(d)[0, j] \in Cons_p \wedge H(d)[0, j+1] \in Cons_p \iff C_{direct\_move\_to\_p\_cons}[j] = 8.\end{aligned} \tag{7.25}$$

1.P2

Contrary motions are preferred to oblique motions which are preferred to direct motions.
This rule applies in all configurations.

$$\begin{gathered}\text{With } cost_{con} = 0 \quad \text{and} \quad cost_{con} < cost_{obl} < cost_{dir} \\ \forall j \in [0, m-1] \\ P_{costs}[j] = \begin{cases} cost_{con} & \text{if } P[0, j] = 0, \\ cost_{obl} & \text{if } P[0, j] = 1, \\ cost_{dir} & \text{if } P[0, j] = 2. \end{cases}\end{gathered} \tag{7.26}$$

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 applies in all configurations.

$$\begin{gathered}i := \max(B),\ \forall j \in [0, m-1],\ \forall p \in \{cf, cp1, cp2, cp3\} \\ H(p)[0, j+1] = 0 \wedge P(p)[i, j] = 0 \wedge M_{brut}(p)[i, j] < -4 \wedge \neg A(p) \iff \bot,\end{gathered} \tag{7.27}$$

1.P4

Successive perfect consonances should be avoided.
This rule applies in all configurations, except second species in three and four voices (see 2.P3).

$$\begin{gathered}\forall p_1 \neq p_2 \in \{cf, cp1, cp2, cp3\},\ \forall j \in [0, m-1] \\ \big(H(p_1,p_2)[0,j] \in Cons_p \wedge H(p_1,p_2)[0,j+1] \in Cons_p\big)\ \Rightarrow\ C_{succ\_p\_cons} = 2.\end{gathered} \tag{7.28}$$

1.P5

Each part starts distant from the lowest stratum.

This is a general indication for the composer (to make it easier for parts to move towards each other in contrary motion), but no explicit constraint is implemented.

1.P6

It is prohibited that all parts move in the same direction.
This rule applies in three and four voices configurations.

$$\begin{gathered}\forall j \in [0, m-1] \\ \bigvee_{p \in \{cf, cp1, cp2, cp3\}} P(p)[0,j] \in \{0,1\}.\end{gathered} \tag{7.29}$$

1.P7

It is prohibited to use successive ascending sixths on a direct upwards motion.
This rule applies to all configurations in three or four voices.

$$\begin{gathered}\forall j \in [1, m-1],\ \forall p_1 \neq p_2,\ \text{with } sixth := \{8, 9\} \\ H(p_1,p_2)[0,j-1] \notin sixth \\ \vee\ H(p_1,p_2)[0,j] \notin sixth \\ \vee\ M_{brut}(p_1)[0,j] \leq 0 \\ \vee\ M_{brut}(p_2)[0,j] \leq 0\end{gathered} \tag{7.30}$$

7.3 Constraints of the Second Species

Harmonic Rules

2.H2

Arsis harmonies cannot be dissonant except if there is a diminution[1].
This rule applies to all numbers of voices, on second species counterpoints.

$$\forall j \in [0, m-2],\ \forall p:\ species(p) = 2$$

$$IsDim(p)[j] = \begin{cases} \top & \text{if } M^2(p)[0,j] \in \{3,4\} \wedge M^1(p)[0,j] \in \{1,2\} \wedge M^1(p)[2,j] \in \{1,2\} \\ \bot & \text{otherwise} \end{cases}$$

$$\forall j \in [0, m-2] \quad \neg IsCons(p)[2,j] \ \Rightarrow\ IsDim(p)[j] \tag{7.31}$$

2.H3 & 2.H4

When composing in two voices, 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.
This rule applies to all two voice configurations in which the counterpoint is of the second species.

$$H[0, m-2] \in \{7, 8, 9\}$$

$$penulthesiscost = \begin{cases} cost_{penulthesis} & \text{if } H[0,m-2] \neq 7 \\ 0 & \text{otherwise} \end{cases} \tag{7.32}$$

2.H6 & 3.H5

The half notes (or quarter notes in the third species) must be coherent ("concordant") with respect to the whole notes.

This is not really a constraint, but more of an indication. It was decided to leave it in this complete set of rules because it is a redundant concept in *Gradus ad Parnassum*.

[1] Remember that a diminution consists of an intermediate note that exists between two notes separated by a skip of a third.

Melodic Rules

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.
This rule applies to all two voice configurations in which the counterpoint is of the second species.

$$\begin{gathered} \forall j \in [0, m-2],\ \forall M_{cf}[j] \neq 0 \\ M[0,j] = 12 \ \Rightarrow\ (H[0,j] \leq 4) \ \wedge\ (\neg A(cp)[j] \ \Leftrightarrow\ M_{cf}[j] > 0) \end{gathered} \tag{7.33}$$

2.M2

In two-voice composition, two consecutive notes cannot be the same. When writing a three or four voice composition, the 4th-to-last, the 3rd-to-last and the 2nd-to-last may be the same.
This rule applies to all configurations, to second species counterpoints.

`2V:`

$$\begin{gathered} \forall \rho \in \mathit{positions}(m) \\ N[\rho] \neq N[\rho+1] \end{gathered} \tag{7.34}$$

`3V & 4V:`

$$\begin{gathered} \forall p:\ \mathit{species}(p) = 2,\ \forall j \in [1, m-3] \\ \big((N[2,j-1] \neq N[0,j]) \wedge (N[0,j] \neq N[2,j])\big) \ \wedge \\ \big((N[2,m-3] \neq N[0,m-2]) \vee (N[0,m-2] \neq N[2,m-2])\big) \end{gathered} \tag{7.35}$$

Motion Rules

2.P1 & 1.P2

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.
This rule applies to all counterpoints of the second species.

$$\begin{gathered} \forall j \in [0, m-2] \\ \mathit{Preal}[j] = \begin{cases} P[2,j] & \text{if } M[0,j] > 4 \\ P[0,j] & \text{otherwise} \end{cases} \end{gathered} \tag{7.36}$$

Note: The rule concerning the battuta octave (1.P3) always perceives motion from the last note of the previous measure, but this is already explicitly for-

mulated in the formalization of the rule, so it does not impact this rule or need a new formulation.

2.P3

For three and four part composition, successive fifths on the downbeat are only allowed when they are separated by a third on the upbeat.
This rule applies to all configurations in three and four voice composition in which second species counterpoints are involved.

$$\forall p_1, p_2 \in \{cf, cp1, cp2, cp3\} \text{ with } p_1 \neq p_2 \text{ and } (species(p_1) = 2 \vee species(p_2) = 2), \quad \forall j \in [0, m-3]$$

$$Cost_{succ\_p\_cons} = \begin{cases} 0 & \text{if } (H(p_1,p_2)[0,j] \notin Cons_p) \ \vee \ (H(p_1,p_2)[0,j+1] \notin Cons_p) \\ 0 & \text{if } (H(p_1,p_2)[0,j] = 5) \ \wedge \ (H(p_1,p_2)[0,j+1] = 5) \\ & \quad \wedge \big(H(p_1,p_2)[2,j] \in \{3,4\}\big) \\ 2 & \text{otherwise} \end{cases} \tag{7.37}$$

7.4 Constraints of the Third Species

Harmonic Rules

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.
This rule applies in all configurations, to third species counterpoints.

As discussed in section 6.3, this rule is currently disabled and must be modified in a future work.

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.

$$\forall j \in [0, m-3]$$
$$IsCons[2,j] \vee (IsCons[1,j] \wedge IsCons[3,j] \wedge IsDim[j])$$
where $IsDim[j] = \top$ when the 3^{rd} note of the measure j is a dimension. (7.38)

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.
This rule applies in all configurations, to third species counterpoints.

$$\forall j \in [0, m-2]$$
$$Cambiatacosts[j] = \begin{cases} cost_{Cambiata} & \text{if } IsCons[1,j] \wedge IsCons[2,j] \wedge M[1,j] \leq 2 \\ 0 & \text{otherwise} \end{cases} \tag{7.39}$$

3.H4

In two voice composition, in the penultimate measure, if the counterpoint is the lowest stratum, then the harmonic interval of the first note should be a minor third.

$$A(cp)[m-2] \implies H[0, m-2] = 3 \tag{7.40}$$

3.H6

If the harmonic triad could not be used on the downbeat, it should be used on the second or third beat.
This rule applies in all three and four voice configurations in which there is a third species counterpoint.

$$\forall j \in [0, m-2]$$
$$(H[0,j] \notin Cons_{h\_triad}) \wedge (H[1,j] \notin Cons_{h\_triad}) \wedge (H[2,j] \notin Cons_{h\_triad})$$
$$\iff cost_{harmonic\_triad\_3rd\_species}[j] = 1 \tag{7.41}$$

3.H7

On the downbeat of a four-part composition, it is now preferred to double the third or the fifth, rather than to use a unison.
This rule applies only in four voice composition, when a third species counterpoint is involved.

This rule is handled by 1.H8 by setting *triad_with_unison_cost* superior to the *double_thirds_cost* and the $double\_fifths\_cost$.

Melodic Rules

3.M1

Each note and its two beats further peer are preferred to be different.
This rule applies in all configurations, to third species counterpoints.

$$\begin{gathered}\forall \rho \in positions(m-2) \\ MtwoSame_{costs}[i,j] = \begin{cases} cost_{MtwoSame} & \text{if } M^2[\rho] = 0 \\ 0 & \text{otherwise} \end{cases}\end{gathered} \tag{7.42}$$

3.M2

No melodic interval between 9 and 11 semitones.
This rule applies to all third-species counterpoints.

$$\begin{gathered}\forall i \in \mathcal{B}, \quad \forall j \in [0, m-2], \\ M[i,j] \notin \{9, 10, 11\}\end{gathered} \tag{7.43}$$

3.M3

The second note of the penultimate measure must be more than a semitone away from the fourth note.
This rule applies to all third-species counterpoints.

$$M_{brut}[1, m-2] + M_{brut}[2, m-2] > 1 \tag{7.44}$$

Motion Rules

3.P1

The motion is perceived based on the fourth note.
This rule applies in all configurations, to third species counterpoints.

$$\begin{gathered}\forall j \in [0, m-2] \\ P_{real}[j] = P[3, j]\end{gathered} \tag{7.45}$$

7.5 Constraints of the Fourth Species

Harmonic Rules

4.H1

Arsis harmonies must be consonant.
This rule applies in all configurations, to fourth species counterpoints.

$$\begin{array}{c} \forall j \in [0, m-2] \\ H[2, j] \in Cons \end{array} \tag{7.46}$$

4.H2

In two part composition, if the fourth species counterpoint is the lowest stratum, then no harmonic seventh interval can occur.
This rule applies to two voice compositions in which the counterpoint is of the fourth species.

$$\begin{array}{c} \forall j \in [1, m-2] \\ A(cp)[j] \;\Rightarrow\; H[0, j] \notin \{10, 11\} \end{array} \tag{7.47}$$

4.H3

In the penultimate measure, the harmonic interval of the thesis note must be a major sixth or a minor or major third depending on the cantus firmus pitch.
This rule applies to two voice compositions in which the counterpoint is of the fourth species.

$$H[0, m-2] = \begin{cases} 9 & \text{if } A(cf)[m-2] \\ 3 \vee 4 & \text{otherwise} \end{cases} \tag{7.48}$$

Melodic Rules

4.M1

Arsis half notes should be the same as their next halves in thesis.
This rule applies in all configurations, to fourth species counterpoints.

$$\begin{array}{c} \forall j \in [0, m-2] \\ NoSync_{costs} = \begin{cases} cost_{NoSync} & \text{if } M[2, j] \neq 0 \\ 0 & \text{otherwise} \end{cases} \end{array} \tag{7.49}$$

The cost should be set high, because this syncopation is the very nature of the species.

4.M2

Each arsis note and its two measures further peer are preferred to be different.
This rule applies in all configurations, to fourth species counterpoints.

$$\forall j \in [0, m-2]$$
$$MtwomSame_{costs} = \begin{cases} cost_{MtwomSame} & \text{if } N[2,j] = N[2,j+2] \\ 0 & \text{otherwise} \end{cases} \tag{7.50}$$

Motion Rules

4.P1

Dissonant harmonies must be followed by the next lower consonant harmony.
This rule applies in all configurations, to fourth and fifth species counterpoints.

`4sp:`

$$\begin{gathered} \forall j \in [1, m-1] \\ \neg isCons[0,j] \implies M_{brut}[0,j] \in \{1,2\} \end{gathered} \tag{7.51}$$

`5sp:`

$$\begin{gathered} \forall j \in [1, m-1] \\ \neg isCons[0,j] \wedge S[0,j] = 4 \implies M^2_{brut}[0,j] \in \{1,2\} \end{gathered} \tag{7.52}$$

The use of M^2 is justified because, in the fifth species, every beat can potentially contain a note. In this case, it is necessary to check the melodic interval specifically between the thesis and the arsis note.

4.P2

If the fourth species counterpoint is not the lowest stratum then no second harmony can be preceded by a unison/octave harmony.
This rule applies in all configurations, to fourth species counterpoints.

$$\begin{gathered} \forall j \in [1, m-2] \\ \neg A[j+1] \;\Rightarrow\; H[2,j] \neq 0 \;\wedge\; H[0,j+1] \notin \{1,2\} \end{gathered} \tag{7.53}$$

7.6 Constraints of the Fifth Species

Melodic Rules

5.M1

No unison between two consecutive notes.
This rule applies to all fifth species counterpoints.

$$\begin{gathered}\forall j \in [0, m-2], \\ N[0,j] \neq N[1,j], \\ N[1,j] \neq N[2,j], \\ N[2,j] \neq N[3,j] \\ N[3,j] \neq N[0,j+1]\end{gathered} \tag{7.54}$$

5.M2

For each measure from the second to the second-to-last, if a note is part of the fourth species and the corresponding subsequent note is constrained, then these two notes must not be the same.
This rule applies to all fifth species counterpoints.

$$\begin{gathered}\forall j \in [1, m-2] \\ (S[0,j] = 4 \wedge S[2,j] = 4) \implies N[0,j] \neq N[2,j]\end{gathered} \tag{7.55}$$

Rhythmic Rules

The fifth species does not operate in the same manner as the previous four. In this case, the S array is used to indicate the species assigned to each note, describing both its rhythmic value and the rules that govern it. The rhythmic constraints outlined below are defined on this S array, and from it one can deduce, via the equations of this section, which set of rules applies to every note.

To make the formalization more readable, we define $IsSx$:

$$\begin{gathered}\forall x \in \{0,1,2,3,4\},\ \forall \rho \in positions(m) \\ IsSx[\rho] = \begin{cases} \top & \text{if } S[\rho] = x \\ \bot & \text{otherwise} \end{cases}\end{gathered} \tag{7.56}$$

5.R1

There must always be a note in thesis and in arsis, except the very first thesis and the very last arsis.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall j \in [0, m-1] \\ \neg IsS_0[0,j] \quad \text{where } j \neq 0\end{gathered} \tag{7.57}$$

$$\forall j \in [0, m-1]: \quad \neg IsS_0[2,j] \quad \text{where } j \neq m-1 \tag{7.58}$$

5.R2

The 4th species can only exist in first and third beat.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall i \in \{1,3\},\ \forall j \in [0, m-1] \\ \neg IsS_4[i,j]\end{gathered} \tag{7.59}$$

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 rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall j \in [0, m-2] \\ IsS_4[2,j] \iff IsS_4[0,j+1]\end{gathered} \tag{7.60}$$

$$\begin{gathered}\forall j \in [0, m-2] \\ IsS_4[2,j] \implies IsS_0[3,j]\end{gathered} \tag{7.61}$$

5.R4

A 3rd species cannot be followed by no note.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall \rho \in positions(m-1) \\ IsS_3[\rho] \implies \neg IsS_0[\rho+1]\end{gathered} \tag{7.62}$$

5.R5

Only 3rd species and 4th species are used.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered}\forall \rho \in positions(m) \\ \neg IsS_1[\rho] \ \wedge \ \neg IsS_2[\rho]\end{gathered} \tag{7.63}$$

5.R6

The first and penultimate measures are linked to the 4th species.
This rule applies in all configurations, to fifth species counterpoints.

$$IsS_0[0,0] \wedge IsS_0[1,0] \wedge IsS_4[2,0] \tag{7.64}$$

$$IsS_4[0, m-2] \wedge IsS_0[1, m-2] \wedge IsS_4[2, m-2] \tag{7.65}$$

5.R7

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 rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered} \forall j \in [1, m-2] \\ \neg IsCons[0, j] \wedge IsS_4[0, j] \implies M^2_{brut}[0, j] \in \{-1, -2\} \wedge IsCons[2, j] \end{gathered} \tag{7.66}$$

5.R8

To avoid multiple same final solutions, the non-displayed notes (whose values in S are 0) must be of the same value as the note on the next beat.
This rule applies in all configurations, to fifth species counterpoints.

$$\begin{gathered} \forall \rho \in positions(m-1) \\ IsS_0[\rho] \implies (N[\rho] = N[\rho + 1]) \end{gathered} \tag{7.67}$$

5.R9

When composing two counterpoints of the fifth species, the counterpoints must use different species for at least half of the beats. If there are three counterpoints of the fifth species, this rule applies between the fifth species counterpoint with the minimum range and the one with the maximum range. The middle one remains unconstrained by this rule.
This rule applies when there are multiple counterpoints of the fifth species.

$$species(cp_1) = species(cp_2) = 5 \iff \sum_{i=0}^{3} \sum_{j=0}^{m-1} (S(cp_1)[i, j] = S(cp_2)[i, j]) < \frac{sm}{2} \tag{7.68}$$

Determining the Constraints that Apply to Each Note

$$\begin{gathered} \forall x \in \{3, 4\},\ \forall cst_x \in Constraints(x),\ \forall V \in Variables(cst_x) \\ \Big(\bigwedge_{v \in V} IsS_x[v_{pos}] \Big) \implies cst_x(V) \end{gathered} \tag{7.69}$$

where $Constraints(x)$ is the set of constraints of the species x, and $Variables(cst_x)$ is the set of variables concerned by the constraint cst_x, and v_{pos} is the position of the v related note in the array N.

Special case for 1.H1 and 4.H1

$$\forall j \in [0, m-1]: \quad IsS_3[0,j] \implies (H[0,j] \in Cons) \tag{7.70}$$

$$\begin{gathered} \forall j \in [0, m-2] \\ IsS_4[2,j] \implies (H[2,j] \in Cons) \end{gathered} \tag{7.71}$$

7.7 Tables of rules scope

	All species	**First sp.**	**Second sp.**	**Third sp.**	**Fourth sp.**	**Fifth sp.**
nV	G1–5, G7, 1H1, 1H4, 1H6, 1P2, 1P4		2H2, 2P1	3H1, 3H2, 3H3, 3M1, 3M2, 3M3, 3P1	4H1, 4M1, 4M2, 4P1, 4P2	5.R1–R8, 5M1, 5M2
2V	1H2, 1H3, 1H7, 1P1	1M2	2H3, 2M1, 2M2	3H4	4H2, 4H3	
2V3V	G6, 1H5					
3V	1H7, 1H8, 1H10, 1P1					
3V4V	1H3, 1H12, 1H13, 1M4, 1P6, 1P7	1M2	2M2, 2P3	3H6	4P3, 4P5	5.R9
4V	1H8, 1P1			3H7		

Table 7.1: Table of Scopes

Chapter 8

Limitations and Future Work

This work consolidates and validates a significant part of the Fux counterpoint model, highlighting areas where the implementation and formal rule set fall short of current practice. Below, I summarise the main limitations encountered during the project and outline concrete next steps that remain faithful to Fux while leveraging the clarity of Gallon–Bitsch.

8.1 Limitations

Code readability and duplicated logic

A very practical obstacle was the codebase itself. Many files lack comments or brief rationales at the point where constraints are posted. Combined with this, each species counterpoint class often maintains its own interval arrays—sometimes several—with different sizes and even different implementations. In practice this makes it hard to know which array is "the" authoritative one to use when debugging or extending a rule. This duplication also clashes with the conceptual unification already present in the model through the core arrays N, H, M, P and the species grid S, and with the shared class structure described earlier.

Feature gaps surfaced by figure-driven validation

When I validated the solver against Fux's figures using MUS, I had to adapt many fifth-species examples: the current program does not yet cover several fifth-species rhythms and ornaments. In effect, I restricted fifth species to quarter notes (third species behaviour) and tied half notes (fourth species syncopations) to make the tests executable. This means the solver cannot yet reproduce Fux's florid variety verbatim.

Relatedly, certain tolerances documented during the audit are not encoded yet. Examples include the use of a second-species syncopation in the penultimate bar, which I had to test without syncopation because it is not currently

supported, and cadential tolerances (e.g., the diminished fifth in the penultimate bar when the bass delays the leading tone) that appear in the literature but are not fully parameterized in the solver.

Incomplete rule coverage across contexts

Some rules required by analysis are disabled or only partially applied across species/voice counts due to where they are posted in the class hierarchy (e.g., implementation tied to a specific species constructor instead of the voice-configuration layer). This surfaced during MUS analysis and refactoring plans (e.g., for penultimate-bar cadence rules).

Test coverage imbalance

I began with unit tests but only implemented about ten constraint-level tests before switching to figure-driven MUS diagnostics, which proved far more informative for real examples. As a result, low-level unit coverage remains thin.

Limited interactivity/integration at this stage

The implementation is now a standalone C++ library (the OpenMusic dependency is gone), and a native C++ UI is being developed separately. Until that UI work lands, day-to-day composition workflows remain code-centric and less interactive than desired.

8.2 Future Work

Bring in Gallon–Bitsch—explicitly, but keep Fux the default

A clear, high-leverage next step is to encode the rules from Noël Gallon & Marcel Bitsch. Their treatise is numbered, explicit, and exhaustive, which maps naturally to constraints and helps resolve remaining ambiguities. I want to use it as a clarifying layer while preserving Fux style as the goal and default—i.e., provide a "Gallon–Bitsch profile" that can be toggled or intersected with Fux, not a replacement. The thesis already uses Gallon–Bitsch as an authoritative cross-check; formalizing those rules directly will make the solver clearer, more complete, and easier to extend—without drifting from Fux.

Refactor to a single "theory kernel"

To eliminate confusion and duplication:

- **Centralize interval/motion helpers** (harmonic, melodic, motion categories) into one shared module parameterized by species and voice count.
- **Retire per-species interval arrays** (with divergent sizes/implementations) in favour of a single authoritative representation that all species use.

- **Add lightweight comments** where constraints are posted: what rule (and source) is being encoded, scope (species/voices), and any tolerated exceptions.

This aligns the code with the already unified conceptual arrays N, H, M, P, S and the class hierarchy documented earlier.

Complete missing behaviours exposed by validation

- **Fifth species:** implement the absent rhythmic figures/ornaments so Fux's florid examples can run without simplification. At least allowed the use of first and second species type note in fifth species part.
- **Second-species penultimate syncopation:** add the allowed syncopated cadence option used in the sources.
- **Cadential tolerances:** encode the documented, tightly scoped tolerances (e.g., diminished fifth with delayed leading tone in the bass) behind explicit switches.
- **Constructor placement of global rules:** move cadence rules that must hold across species (e.g., penultimate-bar intervals) to the voice-configuration layer so they're always active in the right contexts.

Strengthen tests without losing MUS benefits

Keep MUS for figure-level diagnosis, and complement it with:

- **Property-style unit tests** for low-level helpers (interval reductions, motion detection, stratum mapping).
- **Rule-scope smoke tests** per species/voice count to catch constructor-scope regressions early.

Integrate the upcoming UI

Once the native C++ UI is available, expose:

- **Constraint toggles and profiles** (Fux vs. Gallon–Bitsch) and
- **MUS-based explanations** ("which minimal rule set broke this edit?")

so users can lock notes, make local edits, and re-solve with immediate feedback.

Chapter 9

Conclusion

This thesis set out to turn the pedagogy of species counterpoint into an explicit, testable computational model: a clarified rule system faithful to Fux, implemented over a constraint solver, and validated by systematic tests. In doing so, it connects three strata that are often kept apart—historical sources, a formal theory, and working software—so that each can inform and check the others.

9.1 What was delivered

Conceptually, the work consolidates and corrects the formal model of Fuxian counterpoint, distinguishing necessities (hard rules) from tendencies (soft preferences) and using Minimal Unsatisfiable Subsets (MUS) to pinpoint tensions between the specification and canonical examples. Practically, it extends and fixes the Gecode-based implementation, adds a modular on/off mechanism for constraints, integrates MUS diagnostics into the testing loop, and unifies the rule library and its documentation into a reproducible framework. The core arrays N, H, M, P and the species grid S provide a common vocabulary to express melodic, harmonic and rhythmic structure in a way that is both human-legible and solver-friendly.

The project also completes an engineering transition begun in prior work: FuxCP no longer depends on OpenMusic and is maintained as a standalone C++ library on Gecode, with a native C++ UI being developed separately. This decoupling preserves transparency and makes broader integration easier.

9.2 What we learned

Figure-driven MUS analysis proved decisive. In practice, every failing example was explained by a single constraint—MUS sets never exceeded size one—supporting the model's modularity and making fixes tractable. At the same time, several fifth-species figures had to be simplified because some rhythmic patterns and ornaments are not yet handled; restricted tests used third-

species quarters and fourth-species ties for syncopation. During diagnostics, previously undocumented rules surfaced in code and were tracked explicitly. Finally, the audit highlighted duplicated interval/motion tables scattered across species classes—an obstacle to clarity and extension.

9.3 Why it matters

Constraint Programming remains a fitting foundation: rules are explicit and traceable; preferences encode stylistic latitude; and solutions are inspectable rather than opaque. In context, FuxCP is designed to assist composition and is positioned for pedagogical use—provided the solver's explanations and interfaces continue to mature.

9.4 Where this naturally goes next

With the Fux corpus now reviewed and corrected where needed, the next tractable step is to encode the numbered, explicit rules of Noël Gallon and Marcel Bitsch as an auxiliary source—resolving ambiguities while preserving a Fux-first "style profile." In parallel, a refactor to a single "theory kernel" (one authoritative set of interval/lookup utilities shared by all species/voice counts) should eliminate duplication and lower cognitive load. Completing missing fifth-species behaviours, targeting mixed-species textures such as the four-voice *grand mélange*, and integrating the forthcoming UI with explainable feedback and standard I/O (MusicXML/MIDI) round out the roadmap, alongside empirical classroom studies.

9.5 Closing

The combined outcome is a clearer, more faithful executable reading of species counterpoint: a validated catalogue of rules, an implementation that explains its own failures, and a workflow that reproduces canonical figures where supported and identifies precisely where the model must evolve. These foundations make the next extensions—Gallon–Bitsch, cleaner internals, richer rhythms, and denser textures—both feasible and well-motivated.

Author's Note on Language Assistance

All ideas, structure, and academic argumentation in this thesis—including the formulation of the research questions, the design of the methods, the analysis of the results, and the final interpretations—are entirely my own. To improve clarity, fluency, and grammar in English, I used ChatGPT as an editing aid for phrasing and style. The tool did not generate new content, nor did it determine the substance of the work. Responsibility for all content and any remaining errors is mine alone.

Bibliography

[1] Aiva Technologies SARL. Aiva — ai music generation assistant. URL: https://www.aiva.ai/.

[2] Anton Belov, Inês Lynce, and Joao Marques-Silva. Towards efficient MUS extraction. January 2012. URL: https://researchrepository.ul.ie/articles/journal_contribution/Towards_efficient_MUS_extraction/19854478.

[3] Jean-Pierre Briot, Gaëtan Hadjeres, and François Pachet. Deep learning techniques for music generation - A survey. *CoRR*, abs/1709.01620, 2017. URL: http://arxiv.org/abs/1709.01620, arXiv:1709.01620.

[4] Simonne Chevalier. *Gradus ad Parnassum: Manuel pour la composition régulière de la musique*. Editions Gabriel Foucou, January 2020.

[5] Louis Cleenewerk and Denis de Patoul. Fuxcp : a constraint programming formalization of fux's musical theory of four-voice counterpoint. Master's thesis, École polytechnique de Louvain, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2024. URL: https://hdl.handle.net/2078.2/41477.

[6] Kemal Ebcioğlu. Computer counterpoint. In *Proceedings of the 1980 International Computer Music Conference*, pages 534–543, 1980. Accessed 2025-08-18. URL: https://quod.lib.umich.edu/cgi/p/pod/dod-idx/computer-counterpoint.pdf?c=icmc;idno=bbp2372.1980.041;format=pdf.

[7] Johann Joseph Fux. *Gradus ad Parnassum, sive Manuductio ad Compositionem Musicae regularem*. Joannis Petri van Ghelen, Vienna, 1725.

[8] Johann Joseph Fux, Alfred Mann, and John Edmunds. *The study of counterpoint from Johann Joseph Fux's Gradus ad parnassum*. Number N277 in The Norton library. W. W. Norton, New York, rev. ed., 32. print edition, 1971.

[9] Noël Gallon and Marcel Bitsch. *Traité de contrepoint*. Durand, Paris, 1964.

[10] Google. A.i. duet - a piano that responds to you. URL: https://experiments.withgoogle.com/ai/ai-duet/view/.

[11] Google. Magenta — make music and art using machine learning. URL: https://magenta.withgoogle.com/.

[12] IRCAM. Openmusic — ircam forum. https://forum.ircam.fr/projects/detail/openmusic/. Accessed 2024-11-27.

[13] Anton Lamotte. Fuxcp: Constraint programming formalisation of three-voice counterpoint according to fux. Master's thesis, École polytechnique de Louvain, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2024. URL: https://hdl.handle.net/2078.2/38126.

[14] Bill Schottstaedt. Automatic species counterpoint. Technical Report STAN-M-19, Center for Computer Research in Music and Acoustics, Stanford University, Stanford, CA, 1984. Accessed 2025-08-18. URL: https://ccrma.stanford.edu/files/papers/stanm19.pdf.

[15] Christian Schulte, Guido Tack, and Mikael Z. Lagerkvist. *Modeling and Programming with Gecode*, version 6.2.0 edition, 2019. Published May 28, 2019. URL: https://www.gecode.org/doc/6.2.0/MPG.pdf.

[16] Sony Computer Science Laboratories, Inc. Flow machines — augmenting creativity with ai. URL: https://www.flow-machines.com/.

[17] Damien Sprockeels. Melodizer : a constraint programming tool for computer-aided musical composition. Master's thesis, École polytechnique de Louvain, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2022. URL: https://hdl.handle.net/2078.2/26512.

[18] Thibault Wafflard. Fuxcp: a constraint programming based tool formalizing fux's musical theory of counterpoint. Master's thesis, École polytechnique de Louvain, Université catholique de Louvain, Louvain-la-Neuve, Belgium, 2023. URL: https://hdl.handle.net/2078.2/33371.

Appendix A

Complete FuxCP Code

It is required to include all of the code of FuxCP in the appendix of master thesis. Everything is available on GitHub (https://github.com/TomLaiUCL/FuxCP5). This appendix serves to leave an additional trace of the code.

A.1 Counterpoint header files

CounterpointProblem.hpp

```
1  //
2  // Created by Luc Cleenewerk and Diego de Patoul.
3  //
4
5  #ifndef MYPROJECT_COUNTERPOINTPROBLEM_HPP
6  #define MYPROJECT_COUNTERPOINTPROBLEM_HPP
7
8  #include "../Utilities.hpp"
9  #include "../Parts/FirstSpeciesCounterpoint.hpp"
10 #include "../Parts/SecondSpeciesCounterpoint.hpp"
11 #include "../Parts/CantusFirmus.hpp"
12
13 /** Types of search engines */
14 enum {
15     dfs_solver, //0
16     bab_solver, //1
17 };
18
19
20 /**
21  * This (abstract) class gives a general model for a counterpoint problem.
22  */
23 class CounterpointProblem : public IntLexMinimizeSpace{
24 protected:
25     CantusFirmus* cantusFirmus;
26
27     int nMeasures;      /// the number of measures in the score to generate
28     int n_unique_costs;
29     Stratum* lowest;
30     IntVarArray successiveCostArray;
31     IntVarArray triadCostArray;
32     Part* counterpoint_1;
33     Part* counterpoint_2;
34     Part* counterpoint_3;
35     Stratum* upper_1;
```

```
36      Stratum* upper_2;
37      Stratum* upper_3;
38      IntVarArray unitedCosts;
39      IntVarArray sortedCosts;
40      IntVarArray solutionArray;
41      vector<int> importance;
42      vector<string> importanceNames;
43      vector<string> unitedCostNames;
44      IntVarArray orderedFactors;
45      IntVarArray finalCosts;
46      vector<IntVarArray> sorted_voices;
47      vector<IntVarArray> measures_order;
48      unordered_map<string, int> prefs;
49      vector<vector<string>> costLevels;
50
51      IntVar globalCost;
52      // vector<int> species;        /// the species of the counterpoint to
          ↪ generate
53
54      IntVarArray combinedCosts;
55
56  public:
57      /**
58       * Constructor of the class.
59       * @param cf a vector<int> representing the cantus firmus.
60       * @param k the key of the score. it takes values from the notes in headers/
            ↪ Utilities.hpp
61       * @param lb the lowest note possible for the counterpoints in MIDI
62       * @param ub the highest note possible for the counterpoints in MIDI
63       */
64      CounterpointProblem(vector<int> cf, int v_type, vector<int> m_costs, vector<
          ↪ int> g_costs, vector<int> s_costs, vector<int> imp, int nV);
65
66      CounterpointProblem(CounterpointProblem& s);
67      virtual IntLexMinimizeSpace* copy();
68
69      virtual string to_string() const;
70
71      /**
72       * Constrain method for bab search
73       * @todo modify this function if you want to use branch and bound
74       * @param _b a space to constrain the current instance of the Problem class
            ↪ with upon finding a solution
75       */
76      void constrain(const IntLexMinimizeSpace& _b);
77
78      IntVarArgs cost() const;
79
80      /// Getters
81      //Part* getCounterpoint(){ return counterpoint_1; }
82      ///destructor
83      virtual ~CounterpointProblem(){ delete cantusFirmus; delete lowest; delete
          ↪ counterpoint_1; delete counterpoint_2; delete counterpoint_3; delete
          ↪ upper_1;
84          delete upper_2, delete upper_3;}
85
86      Home getHome();
87
88      void setPreferenceMap(vector<string> importance_names);
89
90      void orderCosts();
91
92      int getSize();
93
94      int* return_solution();
95
96      IntVarArray getSolutionArray();
97
```

```
    Stratum* getLowest();

    // --- Added getters for counterpoint_2 and counterpoint_3 (Modified by Tom Lai)
    Part* getCantusFirmus();
    Part* getCounterpoint_1();
    Part* getCounterpoint_2();
    Part* getCounterpoint_3();
    // ---

    Stratum* getUpper_1(){ return upper_1; }
    Stratum* getUpper_2(){ return upper_2; }
    Stratum* getUpper_3(){ return upper_3; }

    int* get_species_array_5sp(int ctp_index);
    int* get_extended_cp_domain(int ctp_index);
    int  get_ext_cp_domain_size(int ctp_index);

    vector<IntVarArray> getMeasuresOrder() {
        return measures_order;
    }
    // Helper functions for setStrata
    void setVoiceNote(IntVarArray& voices, int voiceIndex, Part* part, int measureIndex);
    void setStrataAtPosition(int measureIndex, int position, int nVoices, IntVarArray& );
    void setVoiceLowestFlags(int measureIndex, int nVoices, int size);
    void setMelodicIntervalConstraints(int measureIndex, int nVoices);

    void setStrata();

    IntVarArray get_combinedCosts(){ return combinedCosts; }
    void computeCombinedCosts();
};


/**
 * Creates a search engine for the given problem
 * Should only be used when using OM, otherwise you can create the solver etc in the main file
 * @todo Modify this function to add search options etc
 * @param pb an instance of the Problem class representing a given problem
 * @param type the type of search engine to create (see enumeration in headers/gecode_problem.hpp)
 * @return a search engine for the given problem
 */
Search::Base<CounterpointProblem>* make_solver(CounterpointProblem* pb, int type) ;


/**
 * Returns the next solution space for the problem
 * Should only be used when using OM
 * @param solver a solver for the problem
 * @return an instance of the Problem class representing the next solution to the problem
 */
CounterpointProblem* get_next_solution_space(Search::Base<CounterpointProblem>* solver);

#endif //MYPROJECT_COUNTERPOINTPROBLEM_HPP
```

FourVoiceCounterpoint.hpp

```
    //
```

```cpp
// Created by Luc Cleenewerk and Diego de Patoul.
//

#ifndef MYPROJECT_FOURVOICECOUNTERPOINT_HPP
#define MYPROJECT_FOURVOICECOUNTERPOINT_HPP

#include "../Utilities.hpp"
#include "CounterpointProblem.hpp"
#include "../Parts/FirstSpeciesCounterpoint.hpp"
#include "../Parts/SecondSpeciesCounterpoint.hpp"
#include "../Parts/ThirdSpeciesCounterpoint.hpp"
#include "../Parts/CantusFirmus.hpp"

/**
 * This class models a counterpoint problem with 4 voices.
 */
class FourVoiceCounterpoint : public CounterpointProblem{
protected:

    vector<Species> species;        /// the species of the counterpoints to generate

public:
    /**
     * Constructor of the class.
     * @param cf a vector<int> representing the cantus firmus.
     * @param sp the species of the counterpoint. it takes values from the enum "species" in headers/Utilities.hpp
     * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
     * @param lb the lowest note possible for the counterpoint in MIDI
     * @param ub the highest note possible for the counterpoint in MIDI
     */
    FourVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
      vector<int> s_costs, vector<int> imp, int bm);

    FourVoiceCounterpoint(FourVoiceCounterpoint& s);
    IntLexMinimizeSpace* copy() override;

    string to_string() const override;

    /// Getters
    // Part* getCounterpoint(){ return counterpoint; }

    /// destructor : mother class destructor is sufficient, nothing to add

    void uniteCounterpoints();

    void uniteCosts();
};

#endif
```

ThreeVoiceCounterpoint.hpp

```cpp
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#ifndef MYPROJECT_THREEVOICECOUNTERPOINT_HPP
#define MYPROJECT_THREEVOICECOUNTERPOINT_HPP

#include "../Utilities.hpp"
#include "CounterpointProblem.hpp"
```

```cpp
#include "../Parts/FirstSpeciesCounterpoint.hpp"
#include "../Parts/SecondSpeciesCounterpoint.hpp"
#include "../Parts/ThirdSpeciesCounterpoint.hpp"
#include "../Parts/CantusFirmus.hpp"

/**
 * This class models a counterpoint problem with 3 voices.
 */
class ThreeVoiceCounterpoint : public CounterpointProblem{
protected:

    vector<Species> species;        /// the species of the counterpoints to generate

public:
    /**
     * Constructor of the class.
     * @param cf a vector<int> representing the cantus firmus.
     * @param sp the species of the counterpoint. it takes values from the enum "species" in headers/Utilities.hpp
     * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
     * @param lb the lowest note possible for the counterpoint in MIDI
     * @param ub the highest note possible for the counterpoint in MIDI
     */
    ThreeVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
       vector<int> s_costs, vector<int> imp, int bm);

    ThreeVoiceCounterpoint(ThreeVoiceCounterpoint& s);
    IntLexMinimizeSpace* copy() override;

    string to_string() const override;

    /// Getters
    // Part* getCounterpoint(){ return counterpoint; }

    ///destructor : mother class destructor is sufficient, nothing to add

    void uniteCounterpoints();

    void uniteCosts();
};

#endif //MYPROJECT_THREEVOICECOUNTERPOINT_HPP
```

TwoVoiceCounterpoint.hpp

```cpp
    //
// Created by Damien Sprockeels on 13/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#ifndef MYPROJECT_TWOVOICECOUNTERPOINT_HPP
#define MYPROJECT_TWOVOICECOUNTERPOINT_HPP

#include "../Utilities.hpp"
#include "CounterpointProblem.hpp"
#include "../Parts/FirstSpeciesCounterpoint.hpp"
#include "../Parts/SecondSpeciesCounterpoint.hpp"
#include "../Parts/CantusFirmus.hpp"
#include "../constraints.hpp"

/**
```

```
 * This class models a counterpoint problem with 2 voices.
 */
class TwoVoiceCounterpoint : public CounterpointProblem{
protected:
    Species species;        /// the species of the counterpoint to generate

public:
    /**
     * Constructor of the class.
     * @param cf a vector<int> representing the cantus firmus.
     * @param sp the species of the counterpoint. it takes values from the enum "species" in headers/Utilities.hpp
     * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
     * @param lb the lowest note possible for the counterpoint in MIDI
     * @param ub the highest note possible for the counterpoint in MIDI
     */
    TwoVoiceCounterpoint(vector<int> cf, Species sp, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, vector<int> imp, int bm);

    TwoVoiceCounterpoint(TwoVoiceCounterpoint& s);
    IntLexMinimizeSpace* copy() override;

    string to_string() const override;



    ///destructor : mother class destructor is sufficient, nothing to add

};

#endif //MYPROJECT_TWOVOICECOUNTERPOINT_HPP
```

A.2 Voice header files

Voice.hpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#ifndef FUXCP_BASE_VOICE_HPP
#define FUXCP_BASE_VOICE_HPP

#include "Utilities.hpp"

#include "gecode/kernel.hh"
#include "gecode/int.hh"
#include "gecode/search.hh"
#include "gecode/minimodel.hh"
#include "gecode/set.hh"

using namespace Gecode;
using namespace Gecode::Search;
using namespace std;

/// abstract class!!! Should not be instanciated
/// This class represents a Voice, so it creates all the variables associated to that part and posts the constraints that are species independent
class Voice{
    protected:
        int nMeasures;
        int size;
```

```

        int lowerBound;
        int upperBound;

        IntVarArray notes;
        IntVarArray h_intervals;  // with respect to lowest stratum. TODO set to 0 if isLowest
        IntVarArray m_intervals_brut;
        IntVarArray motions;


    public:
        Voice(Home home, int nMes, int lb, int ub);

        Voice(Home home, Voice& s);  // clone constructor

        int getNMeasures() { return nMeasures; }
        int getSize() { return size; }

        int getLowerBound() { return lowerBound; }
        int getUpperBound() { return upperBound; }

        IntVarArray getNotes() { return notes; }
        IntVarArray getMelodicIntervals();

        /// must be implemented in the child classes, returns the variables to branch on
        // virtual IntVarArray getBranchingNotes();

        virtual Voice* clone(Home home);

        virtual string to_string() const;

        virtual IntVarArgs getFirstNotes();

        IntVarArray getHIntervals();

        IntVarArgs getSecondHInterval();

        IntVarArray getMotions();
};


#endif //FUXCP_BASE_VOICE_HPP
```

Stratum.hpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#ifndef STRATUM_HPP
#define STRATUM_HPP

#include "Utilities.hpp"
#include "Voice.hpp"

#include "gecode/kernel.hh"
#include "gecode/int.hh"
#include "gecode/search.hh"
#include "gecode/minimodel.hh"
#include "gecode/set.hh"

using namespace Gecode;
using namespace Gecode::Search;
```

```
using namespace std;

/// This class represents a Stratum
class Stratum : public Voice{
    protected:
        // intvararray with species of each measure?

    public:
        Stratum(Home home, int nMes, int lb, int ub);

        Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes);

        Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int
          ↪  nV);

        Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int
          ↪  nV1, int nV2);

        Stratum(Home home, Stratum& s);  // clone constructor

        Stratum* clone(Home home) override;

        string to_string() const override;

        void setNote(Home home, int index, IntVar note);

};


#endif //STRATUM_HPP
```

Part.hpp

```
    //
// Created by Damien Sprockeels on 11/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August
  ↪  2024.
//

#ifndef FUXCP_BASE_COUNTERPOINT_HPP
#define FUXCP_BASE_COUNTERPOINT_HPP

#include "../Utilities.hpp"
#include "../Voice.hpp"
#include "../Stratum.hpp"
#include "../constraints.hpp"

#include "gecode/kernel.hh"
#include "gecode/int.hh"
#include "gecode/search.hh"
#include "gecode/minimodel.hh"
#include "gecode/set.hh"

using namespace Gecode;
using namespace Gecode::Search;
using namespace std;

/// abstract class!!! Should not be instanciated
/// This class represents a part, so it creates all the variables associated to
  ↪  that part and posts the constraints that are species independent
class Part : public Voice {
    protected:
        Species species;
        int nVoices;
        int borrowMode;
```

```
31          int voice_type;
32          //Stratum* lowest;
33          IntVarArray melodicDegreeCost;
34          IntVarArray fifthCostArray;
35          IntVarArray octaveCostArray;
36          BoolVarArray is_off;
37          IntVarArray offCostArray;
38          IntVarArray costs;
39          IntVarArray varietyCostArray;
40          IntVarArray directCostArray;
41          BoolVarArray isConsonance;
42          IntVarArray speciesArray;
43          BoolVarArray isNotLowest;
44          BoolVarArray isHighest;
45
46          vector<int> borrowed_scale;
47          vector<int> scale;
48          vector<int> chromatic_scale;
49
50          vector<int> cp_range;
51
52          vector<int> extended_domain;
53          vector<int> off_domain;
54
55          vector<string> cost_names;
56
57          int secondCost;
58          int thirdCost;
59          int fourthCost;
60          int tritoneCost;
61          int fifthCost;
62          int sixthCost;
63          int seventhCost;
64          int octaveCost;
65
66          int borrowCost;
67          int h_fifthCost;
68          int h_octaveCost;
69          int succCost;
70          int varietyCost;
71          int triadCost;
72          int directMoveCost;
73          int penultCost;
74
75          int penultSixthCost;
76          int cambiataCost;
77          int mSkipCost;
78          int triad3rdCost;
79          int m2ZeroCost;
80          int syncopationCost;
81          int prefSlider;
82
83          int directCost;
84          int obliqueCost;
85          int contraryCost;
86
87          IntVarArray toCombineCosts;
88          vector<string> toCombineCostNames;
89
90          //First Species specific variables
91          IntVarArray firstSpeciesNotesCp;
92          IntVarArray firstSpeciesHarmonicIntervals;
93          IntVarArray firstSpeciesMelodicIntervals;
94          IntVarArray firstSpeciesMotions;
95          IntVarArray firstSpeciesMotionCosts;
96
97          //Second species specific variables
98          BoolVarArray isDiminution;
```

```
99          IntVarArray penultCostArray;
100         IntVarArray secondSpeciesMotions;
101         IntVarArray secondSpeciesMelodicIntervals;
102         IntVarArray secondSpeciesRealMotions;
103
104         //Third species specific variables
105         IntVarArray thirdSpeciesHarmonicIntervals;
106         IntVarArray thirdSpeciesMelodicIntervals;
107         BoolVarArray is5QNArray;
108         IntVarArray cambiataCostArray;
109
110         //Fourth species specific variables
111         BoolVarArray isNoSyncopeArray;
112         IntVarArray snycopeCostArray;
113         IntVarArray fourthSpeciesMelodicIntervals;
114
115     public:
116         Part(Home home, int nMes, Species sp, vector<int> cf, int lb, int ub, int
              ↪   v_type, vector<int> m_costs, vector<int> g_costs,
117           vector<int> s_costs, int nV, int bm);
118
119         // Part(Part& s); (no longer copy constructor since not a space anymore.
              ↪  Now just a clone constructor to deep copy the object (called by
              ↪  the Space's copy constructor))
120         Part(Home home, Part& s);  // clone constructor
121
122         /// must be implemented in the child classes, returns the variables to
              ↪  branch on
123         // virtual IntVarArray getBranchingNotes();
124
125         virtual Part* clone(Home home) override;
126
127         virtual string to_string() const override;
128
129         virtual IntVarArray getBranchingNotes();
130
131         virtual IntVarArray getFirstHInterval();
132
133         virtual IntVarArray getFirstMInterval();
134
135         virtual IntVarArray getMotions();
136
137         int getSpecies() { return species; }
138
139         IntVarArray getPartNotes();
140
141         IntVarArray getCosts();
142
143         BoolVarArray getIsHighest();
144
145         int getSuccCost();
146
147         int getTriadCost();
148
149         int getVarietyCost();
150
151         IntVar getVarietyArray(int idx);
152
153         virtual int getHIntervalSize();
154
155         BoolVarArray getConsonance();
156
157         BoolVarArray getIsNotLowest();
158
159         int getSecondCost();
160         int getThirdCost();
161         int getFourthCost();
162         int getTritoneCost();
```

```
163         int getFifthCost();
164         int getSixthCost();
165         int getSeventhCost();
166         int getOctaveCost();
167         int getHFifthCost();
168         int getHOctaveCost();
169         int getDirectCost();
170         int getBorrowCost();
171         int getPenultCost();
172         int getDirectMoveCost();
173         int getCambiataCost();
174
175         IntVarArray getMelodicDegreeCost();
176
177         IntVarArray getFirstSpeciesHIntervals();
178
179         IntVarArray getFirstSpeciesNotes();
180
181         IntVarArray getFifthCostArray();
182
183         IntVarArray getOctaveCostArray();
184
185         IntVarArray getFirstSpeciesMIntervals();
186
187         IntVarArray getFirstSpeciesMotions();
188
189         IntVarArray getDirectCostArray();
190
191         BoolVarArray getIsOffArray();
192
193         vector<int> getOffDomain();
194
195         vector<int> getExtendedDomain();
196
197         IntVarArray getOffCostArray();
198
199         BoolVarArray getIsDiminution();
200
201         IntVarArray getPenultCostArray();
202
203         IntVarArray getSecondSpeciesMotions();
204
205         IntVarArray getSecondSpeciesMIntervals();
206
207         IntVarArray getSecondSpeciesRealMotions();
208
209         BoolVarArray getIs5QNArray();
210
211         IntVarArray getThirdSpeciesHIntervals();
212
213         IntVarArray getThirdSpeciesMIntervals();
214
215         IntVarArray getCambiataCostArray();
216
217         BoolVarArray getNoSyncope();
218
219         IntVarArray getSyncopeCostArray();
220
221         IntVarArray getFourthSpeciesMIntervals();
222
223         IntVarArray getSpeciesArray();
224
225         void add_cost(Home home, int idx, IntVarArray to_be_added, IntVarArray
              ↪ costs);
226
227         vector<string> getCostNames();
228
229         IntVarArray getToCombineCosts();
```

```
230
231         vector<string> getToCombineCostNames();
232
233         void add_toCombineCost(Home home, int idx, IntVarArray to_be_added,
              ↪ IntVarArray costs);
234 };
235
236
237 #endif //FUXCP_BASE_COUNTERPOINT_HPP
```

CantusFirmus.hpp

```
1     //
2 // Created by Damien Sprockeels on 12/06/2024.
3 // Extended and developed by Luc Cleenewerk and Diego de Patoul up to August
    ↪ 2024.
4 //
5
6 #ifndef MYPROJECT_CANTUSFIRMUS_HPP
7 #define MYPROJECT_CANTUSFIRMUS_HPP
8
9 #include "../Utilities.hpp"
10 #include "Part.hpp"
11 class CantusFirmus : public Part {
12 private:
13     vector<int> cf_vector;
14     IntVarArray disArray; // Array of dissonances
15
16 public:
17     CantusFirmus(Home home, int size, vector<int> cf, Stratum* low, int v_type,
        ↪ vector<int> m_costs, vector<int> g_costs, vector<int> s_costs,
18         int nV);
19
20     string to_string() const override;
21
22     CantusFirmus(Home home, CantusFirmus& s); // clone constructor
23     CantusFirmus* clone(Home home) override;
24
25     IntVarArray getFirstHInterval() override;
26
27     IntVarArray getFirstMInterval() override;
28
29     IntVarArray getMotions() override;
30
31     IntVarArgs getFirstNotes() override;
32
33     int getHIntervalSize() override;
34
35 };
36
37
38 #endif //MYPROJECT_CANTUSFIRMUS_HPP
```

FifthSpeciesCounterpoint.hpp

```
1     //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #ifndef FUXCP_BASE_FIFTHSPECIESCOUNTERPOINT_HPP
6 #define FUXCP_BASE_FIFTHSPECIESCOUNTERPOINT_HPP
7
```

```
8  #include "Part.hpp"
9  #include "CantusFirmus.hpp"
10 #include "../constraints.hpp"
11
12 class FifthSpeciesCounterpoint : public Part{
13
14 protected:
15
16     int solutionLength;
17     int m2Len;
18     CantusFirmus* cantus;
19     IntVarArray fifthSpeciesNotesCp;
20     IntVarArray fifthSpeciesHIntervals;
21     IntVarArray firstHInterval;
22     IntVarArray fifthSpeciesSuccMIntervals;
23     IntVarArray fifthSpeciesMIntervals;
24     IntVarArray fifthSpeciesMTAIntervals;
25     IntVarArray fifthSpeciesM2Intervals;
26     IntVarArray fifthSpeciesMAllIntervals;
27     IntVarArray fifthSpeciesMotions;
28     IntVarArray fifthSpeciesMotionsCosts;
29     BoolVarArray isNthSpeciesArray;
30     BoolVarArray isConstrainedArray;
31     BoolVarArray isMostlyThirdArray;
32     BoolVarArray isThirdSpeciesArray;
33     BoolVarArray isFourthSpeciesArray;
34     BoolVarArray isNotCambiata;
35     IntVarArray m2ZeroCostArray;
36     IntVarArray thirdHTriadArray;
37
38 public:
39
40     /**
41      * General constructor. It takes the mother species as an argument and calls
          ↪ the super constructor from the part class.
42      * Additionally, it sets all the 1st species specific variables as well as
          ↪ the general rules that have to be applied
43      * regardless of the species. It does not apply 1st species specific rules
          ↪ and does not post branching.
44      * todo maybe add a Stratum object or IntVarArray for the lowest voice
45      * @param nMes the number of measures in the composition
46      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
47      * @param lb the lower bound for the counterpoint
48      * @param ub the upper bound for the counterpoint
49      * @param k the key of the composition
50      * @param mSpecies the species from which this is called.
51      * @param low the lowest stratum
52      * @param c the cantus firmus
53      * @param v_type the voice type of the counterpoint
54      * @param m_costs the user-defined melodic costs
55      * @param g_costs the user-defined general costs
56      * @param s_costs the user-defined specific costs
57      * @param bm parameter specifying if borrow Mode is enabled or not
58      * @param nV the number of voices - as it is a 2 voice constructor, this
          ↪ parameter contains the number of voices
59      */
60     FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub,
          ↪  Species mSpecies, Stratum* low, CantusFirmus* c,  int v_type
61     , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
          ↪ nV);
62
63     FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub,
          ↪  Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
64     , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
65
66     FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub,
          ↪  Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
67     , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
```

```
68
69      FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub,
          ↪   Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
70      , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3
          ↪  );
71
72      /**
73       * This function returns a string with the characteristics of the
            ↪  counterpoint. It calls the to_string() method from
74       * the Part class and adds 1st species specific characteristics.
75       * @return a string representation of the current instance of the
            ↪  FirstSpeciesCounterpoint class.
76       */
77      string to_string() const override;
78
79      /// Copy constructor. This needs to copy all useful attributes and update
            ↪  variables. Must call the super copy constructor       NO LONGER NEEDED
80      // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
81      /// Copy function
82      // virtual Space *copy() override;
83
84      FifthSpeciesCounterpoint(Home home, FifthSpeciesCounterpoint& s); // clone
            ↪  constructor
85      FifthSpeciesCounterpoint* clone(Home home) override;
86
87      virtual IntVarArray getBranchingNotes() override;
88
89      IntVarArray getFirstHInterval() override;
90
91      IntVarArray getMotions() override;
92
93      IntVarArray getFirstMInterval() override;
94
95      int getHIntervalSize() override;
96
97      void createSpeciesArrays(Home home);
98
99  };
100
101 #endif
```

FourthSpeciesCounterpoint.hpp

```
1       //
2   // Created by Luc Cleenewerk and Diego de Patoul.
3   //
4
5   #ifndef FUXCP_BASE_FOURTHSPECIESCOUNTERPOINT_HPP
6   #define FUXCP_BASE_FOURTHSPECIESCOUNTERPOINT_HPP
7
8   #include "Part.hpp"
9   #include "CantusFirmus.hpp"
10  #include "../constraints.hpp"
11
12  /**
13   * This class represents a counterpoint of the first species. It inherits from
        ↪  the Part class.
14   * todo modify it so it also works for 3 and 4 voices. Add the appropriate
        ↪  constraints by making a constructor that takes the number of voices as a
        ↪  parameter
15   * todo maybe it should take a Stratum (object or just IntVarArray) for the
        ↪  lowest voice or something like that depending on the formalization
16   */
17  class FourthSpeciesCounterpoint : public Part{
18  protected:
```

```
19
20      CantusFirmus* cantus;
21      IntVarArray fourthSpeciesNotesCp;
22      IntVarArray fourthSpeciesHIntervals;
23      IntVarArray m2IntervalsArray;
24      IntVarArray firstHInterval;
25      IntVarArray m2ZeroArray;
26      IntVarArray sol;
27
28  public:
29      /**
30       * General constructor. It takes the mother species as an argument and calls
           ↪ the super constructor from the part class.
31       * Additionally, it sets all the 1st species specific variables as well as
           ↪ the general rules that have to be applied
32       * regardless of the species. It does not apply 1st species specific rules
           ↪ and does not post branching.
33       * todo maybe add a Stratum object or IntVarArray for the lowest voice
34       * @param nMes the number of measures in the composition
35       * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
36       * @param lb the lower bound for the counterpoint
37       * @param ub the upper bound for the counterpoint
38       * @param k the key of the composition
39       * @param mSpecies the species from which this is called.
40       * @param low the lowest stratum
41       * @param c the cantus firmus
42       * @param v_type the voice type of the counterpoint
43       * @param m_costs the user-defined melodic costs
44       * @param g_costs the user-defined general costs
45       * @param s_costs the user-defined specific costs
46       * @param bm parameter specifying if borrow Mode is enabled or not
47       * @param nV the number of voices - as it is a 2 voice constructor, this
           ↪ parameter contains the number of voices
48       */
49      FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub
           ↪ , Species mSpecies, Stratum* low, CantusFirmus* c,  int v_type
50      , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
           ↪ nV);
51
52      FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub
           ↪ , Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
53      , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
54
55      FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub
           ↪ , Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
56      , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
57
58      FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub
           ↪ , Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
59      , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3
           ↪ );
60
61      /**
62       * This function returns a string with the characteristics of the
           ↪ counterpoint. It calls the to_string() method from
63       * the Part class and adds 1st species specific characteristics.
64       * @return a string representation of the current instance of the
           ↪ FirstSpeciesCounterpoint class.
65       */
66      string to_string() const override;
67
68      /// Copy constructor. This needs to copy all useful attributes and update
           ↪ variables. Must call the super copy constructor       NO LONGER NEEDED
69      // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
70      /// Copy function
71      // virtual Space *copy() override;
72
```

```
    FourthSpeciesCounterpoint(Home home, FourthSpeciesCounterpoint& s); // clone constructor
    FourthSpeciesCounterpoint* clone(Home home) override;

    virtual IntVarArray getBranchingNotes() override;

    IntVarArray getFirstHInterval() override;

    IntVarArray getMotions() override;

    IntVarArray getFirstMInterval() override;

    int getHIntervalSize() override;

};

#endif //FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP
```

ThirdSpeciesCounterpoint.hpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#ifndef MYPROJECT_THIRDSPECIESCOUNTERPOINT_HPP
#define MYPROJECT_THIRDSPECIESCOUNTERPOINT_HPP


#include "FirstSpeciesCounterpoint.hpp"
#include "../Utilities.hpp"

class ThirdSpeciesCounterpoint : public FirstSpeciesCounterpoint {
protected :
    IntVarArray thirdSpeciesNotesCp;              /// The notes of the counterpoint that have to follow the rules for the 2nd species
    IntVarArray thirdSpeciesMotions;              /// This array is the array of REAL motions
    IntVarArray thirdSpeciesMotionCosts;
    IntVarArray m2IntervalsArray;
    IntVarArray m2ZeroArray;
    IntVarArray thirdHTriadArray;
    IntVarArray thirdSpeciesAbsMelodic;

public:

    ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV);

    ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV);

    ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);

    ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3 );

    string to_string() const override;

```

```
    ThirdSpeciesCounterpoint(Home home, ThirdSpeciesCounterpoint &s); // clone constructor

    ThirdSpeciesCounterpoint* clone(Home home) override;

    IntVarArray getFirstHInterval() override;

    IntVarArray getMotions() override;

    IntVarArray getFirstMInterval() override;

    IntVarArray getBranchingNotes() override;

    int getHIntervalSize() override;
};

#endif
```

SecondSpeciesCounterpoint.hpp

```
    //
// Created by Damien Sprockeels on 12/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#ifndef MYPROJECT_SECONDSPECIESCOUNTERPOINT_HPP
#define MYPROJECT_SECONDSPECIESCOUNTERPOINT_HPP


#include "FirstSpeciesCounterpoint.hpp"
#include "../Utilities.hpp"

///This class extends the first species class because most rules of the first species are also valid for the second species
/**
 * This class represents a counterpoint of the second species. It inherits from the FirstSpeciesCounterpoint class.
 */
class SecondSpeciesCounterpoint : public FirstSpeciesCounterpoint {
protected:
    IntVarArray secondSpeciesNotesCp;               /// The notes of the counterpoint that have to follow the rules for the 2nd species
    IntVarArray secondSpeciesHarmonicIntervals;     /// The harmonic intervals between the notes that have to follow the 2nd species rules and the cantus firmus
    IntVarArray secondSpeciesMotionCosts;
    IntVarArray secondSpeciesRealMotionCosts;
    IntVarArray secondSpeciesArsisArray;

public:
    /**
     *
     * @param size
     * @param cf
     * @param lb
     * @param ub
     * @param k
     */

    SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec, Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
    , vector<int> g_costs, vector<int> s_costs, int bm, int nV);

```

```
38      SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub,
          ↪   Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
39      , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
40
41      SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub,
          ↪   Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
42      , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
43
44      SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub,
          ↪   Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
45      , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3
          ↪  );
46
47      string to_string() const override;
48
49      // SecondSpeciesCounterpoint(SecondSpeciesCounterpoint &s);    No longer a
          ↪  copy, now a clone (since it is not a space, and copy constructor are
          ↪  for spaces, while cloning functions are just used to create a deep
          ↪  copy of an object)
50
51      // virtual Space *copy() override;
52
53      SecondSpeciesCounterpoint(Home home, SecondSpeciesCounterpoint &s); // clone
          ↪  constructor
54      SecondSpeciesCounterpoint* clone(Home home) override;
55
56      IntVarArray getFirstHInterval() override;
57
58      IntVarArray getMotions() override;
59
60      IntVarArray getFirstMInterval() override;
61
62      IntVarArray getBranchingNotes() override;
63
64      int getHIntervalSize() override;
65
66  };
67
68
69  #endif //MYPROJECT_SECONDSPECIESCOUNTERPOINT_HPP
```

FirstSpeciesCounterpoint.hpp

```
1       //
2   // Created by Damien Sprockeels on 11/06/2024.
3   // Extended and developed by Luc Cleenewerk and Diego de Patoul up to August
      ↪  2024.
4   //
5
6   #ifndef FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP
7   #define FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP
8
9   #include "Part.hpp"
10  #include "CantusFirmus.hpp"
11  #include "../constraints.hpp"
12
13  /**
14   * This class represents a counterpoint of the first species. It inherits from
        ↪  the Part class.
15   * todo modify it so it also works for 3 and 4 voices. Add the appropriate
        ↪  constraints by making a constructor that takes the number of voices as a
        ↪  parameter
16   * todo maybe it should take a Stratum (object or just IntVarArray) for the
        ↪  lowest voice or something like that depending on the formalization
17   */
```

```
18 class FirstSpeciesCounterpoint : public Part
19 {
20 protected:
21     Species motherSpecies;                          /// The species from which
       ↪  this is called.
22     CantusFirmus* cantus;
23     IntVarArray disArray; // Array of dissonances
24
25 public:
26     /**
27      * General constructor. It takes the mother species as an argument and calls
         ↪  the super constructor from the part class.
28      * Additionally, it sets all the 1st species specific variables as well as
         ↪  the general rules that have to be applied
29      * regardless of the species. It does not apply 1st species specific rules
         ↪  and does not post branching.
30      * todo maybe add a Stratum object or IntVarArray for the lowest voice
31      * @param nMes the number of measures in the composition
32      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
33      * @param lb the lower bound for the counterpoint
34      * @param ub the upper bound for the counterpoint
35      * @param k the key of the composition
36      * @param mSpecies the species from which this is called.
37      * @param low the lowest stratum
38      * @param c the cantus firmus
39      * @param v_type the voice type of the counterpoint
40      * @param m_costs the user-defined melodic costs
41      * @param g_costs the user-defined general costs
42      * @param s_costs the user-defined specific costs
43      * @param bm parameter specifying if borrow Mode is enabled or not
44      * @param nV the number of voices - as it is a 2 voice constructor, this
         ↪  parameter contains the number of voices
45      */
46     FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub,
       ↪   Species mSpecies, Stratum* low, CantusFirmus* c,  int v_type
47     , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
       ↪  nV);
48
49     /**
50      * 2 VOICES CONSTRUCTOR. This constructor is only used when creating a
         ↪  counterpoint of the first species. It calls the other constructor
         ↪  with
51      * FIRST_SPECIES as the mother species. Additionally, it posts 1st species
         ↪  specific constraints.
52      * @param nMes the number of measures in the composition
53      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
54      * @param lb the lower bound for the counterpoint
55      * @param ub the upper bound for the counterpoint
56      * @param k the key of the composition
57      * @param mSpecies the species from which this is called.
58      * @param low the lowest stratum
59      * @param c the cantus firmus
60      * @param v_type the voice type of the counterpoint
61      * @param m_costs the user-defined melodic costs
62      * @param g_costs the user-defined general costs
63      * @param s_costs the user-defined specific costs
64      * @param bm parameter specifying if borrow Mode is enabled or not
65      * @param nV the number of voices - as it is a 2 voice constructor, this
         ↪  parameter is passed to the general constructor
66      */
67     FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub,
       ↪   Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
68     , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
69
70     /**
71      * 3 VOICES CONSTRUCTOR. This constructor is only used when creating a
         ↪  counterpoint of the first species. It calls the other constructor
         ↪  with
```

```
72      * FIRST_SPECIES as the mother species. Additionally, it posts 1st species specific constraints.
73      * @param nMes the number of measures in the composition
74      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
75      * @param lb the lower bound for the counterpoint
76      * @param ub the upper bound for the counterpoint
77      * @param k the key of the composition
78      * @param mSpecies the species from which this is called.
79      * @param low the lowest stratum
80      * @param c the cantus firmus
81      * @param v_type the voice type of the counterpoint
82      * @param m_costs the user-defined melodic costs
83      * @param g_costs the user-defined general costs
84      * @param s_costs the user-defined specific costs
85      * @param bm parameter specifying if borrow Mode is enabled or not
86      * @param nV1 the number of voices - this parameter is there to differenciate it from other number of voices
87      * @param nV2 the number of voices - as it is a 3 voice constructor, this parameter is passed to the general constructor
88      */
89     FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
90     , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
91
92     /**
93      * 4 VOICES CONSTRUCTOR. This constructor is only used when creating a counterpoint of the first species. It calls the other constructor with
94      * FIRST_SPECIES as the mother species. Additionally, it posts 1st species specific constraints.
95      * @param nMes the number of measures in the composition
96      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
97      * @param lb the lower bound for the counterpoint
98      * @param ub the upper bound for the counterpoint
99      * @param k the key of the composition
100     * @param mSpecies the species from which this is called.
101     * @param low the lowest stratum
102     * @param c the cantus firmus
103     * @param v_type the voice type of the counterpoint
104     * @param m_costs the user-defined melodic costs
105     * @param g_costs the user-defined general costs
106     * @param s_costs the user-defined specific costs
107     * @param bm parameter specifying if borrow Mode is enabled or not
108     * @param nV1 the number of voices - this parameter is there to differenciate it from other number of voices
109     * @param nV2 the number of voices - this parameter is there to differenciate it from other number of voices
110     * @param nV3 the number of voices - as it is a 4 voice constructor, this parameter is passed to the general constructor
111     */
112    FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type, vector<int> m_costs
113    , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3 );
114
115    /**
116     * This function returns a string with the characteristics of the counterpoint. It calls the to_string() method from
117     * the Part class and adds 1st species specific characteristics.
118     * @return a string representation of the current instance of the FirstSpeciesCounterpoint class.
119     */
120    string to_string() const override;
121
122    /// Copy constructor. This needs to copy all useful attributes and update variables. Must call the super copy constructor       NO LONGER NEEDED
123    // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
124    /// Copy function
```

```
125     // virtual Space *copy() override;
126
127     FirstSpeciesCounterpoint(Home home, FirstSpeciesCounterpoint& s); // clone
          ↪ constructor
128     FirstSpeciesCounterpoint* clone(Home home) override;
129
130     virtual IntVarArray getBranchingNotes() override;
131
132     IntVarArray getFirstHInterval() override;
133
134     IntVarArray getMotions() override;
135
136     IntVarArray getFirstMInterval() override;
137
138     int getHIntervalSize() override;
139
140 };
141
142 #endif //FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP
```

A.3 Other header files

constraints.hpp

```
1     //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file contains the declarations of the functions that post the constraints
    ↪ .
4 //
5
6 #ifndef CONSTRAINTS_HPP
7 #define CONSTRAINTS_HPP
8
9 class Part;
10
11 #include "Parts/Part.hpp"
12
13 /* ================================================
14  *               GENERAL CONSTRAINTS
15  * ================================================
16  */
17
18 /**
19  * G4 : if a note is borrowed, then a cost must be set
20  */
21 void G4_counterpointMustBeInTheSameKey(Home home, Part* part);
22
23 /**
24  * G6 : chromatic melodies are prohibited
25  */
26 void G6_noChromaticMelodies(Home home, Part* part, int mSpec);
27
28 /**
29  * G7 : prefer smaller melodic intervals (and set tritones to be last resort)
30  */
31 void G7_melodicIntervalsShouldBeSmall(Home home, Part* part, int mSpec);
32
33 /**
34  * G9 : the last chord must be in the scale of the composition
35  */
36 void G9_lastChordSameAsFundamental(Home home, Stratum* lowest, Part* cantusFirmus
    ↪ );
37
38 /* ================================================
```

```
39  *              HARMONIC CONSTRAINTS
40  * ================================================
41  */
42
43 /**
44  * 1.H1 : the harmonic intervals of the first note of each measure must be
        ↪ consonances
45  */
46 void H1_1_harmonicIntervalsAreConsonances(Home home, Part* part);
47
48 void H1_3_fiveConsecutiveNotesByJointDegree(Home home, Part* part);
49
50 /**
51  * 1.H2 : The first harmonic interval must be a perfect consonance in 2 voice
        ↪ compositions
52  */
53 void H2_1_startWithPerfectConsonance(Home home, Part* part);
54
55 void H2_2_arsisHarmoniesCannotBeDisonnant(Home home, Part* part);
56
57 void H2_3_disonanceImpliesDiminution(Home home, Part* part);
58
59 /**
60  * 1.H3 : The last chord must be a perfect consonance. This applies to 2 voices,
        ↪ the 3 and 4 voice version is applied in the stratum constructors
61  */
62 void H3_1_endWithPerfectConsonance(Home home, Part* part);
63
64 void H3_2_penultimateNoteDomain(Home home, Part* part);
65
66 void H3_3_cambiataCost(Home home, Part* part);
67
68 //note : H5 was divided into 2 different ocnstraints since the cantusFirmus has a
      ↪  smaller notes array than the other parts
69
70 /**
71  * 1.H5 : Voices cannot play the same notes (except for the first and last note)
        ↪ (cantusFirmus version)
72  */
73 void H5_1_cpAndCfDifferentNotes(Home home, Part* part, Part* cf);
74
75 /**
76  * 1.H5 : Voices cannot play the same notes (except for the first and last note)
        ↪ (inter-voices version)
77  */
78 void H5_1_differentNotes(Home home, vector<Part*> part);
79
80 /**
81  * 1.H6 : Imperfect consonances are preferred (sets costs for fifth and octave)
82  */
83 void H6_1_preferImperfectConsonances(Home home, Part* part);
84
85 /**
86  * 1.H7 : the penultimate note must be a major sixth or a minor third depending
        ↪ on if it is the lowest stratum or not (counterpoint version)
87  */
88 void H7_1_2v_penultimateSixthOrThird(Home home, Part* part);
89
90 /**
91  * 1.H8 : the harmonic triad should be used as much as possible (3 voices)
92  */
93 void H8_3v_preferHarmonicTriad(Home home, Part* part, IntVarArray triadCostArray,
      ↪  Stratum* upper1, Stratum* upper2);
94
95 /**
96  * 1.H8 : the harmonic triad should be used as much as possible (4 voices)
97  */
```

```
98  void H8_4v_preferHarmonicTriad(Home home, IntVarArray triadCostArray, Stratum*
      ↪ upper1, Stratum* upper2, Stratum* upper3);
99
100 /* ================================================
101  *              MELODIC CONSTRAINTS
102  * ================================================
103  */
104
105
106 /**
107  * 1.M1 : melodic intervals cannot exceed a minor sixth
108  */
109 void M1_1_2v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part);
110
111 /**
112  * 1.M1 : melodic intervals cannot exceed a minor sixth but may include an octave
113  */
114 void M1_1_3v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part);
115
116 /**
117  * 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
118  * the counterpoint can be an octave leap.
119  */
120 void M1_2_octaveLeap(Home home, Part* part, Stratum* low);
121
122 void M2_2_2v_twoConsecutiveNotesAreNotTheSame(Home home, Part* part);
123
124 void M2_2_3v_melodicIntervalsNotExceedMinorSixth(Home home, vector<Part*> parts,
      ↪ bool containsThirdSpecies);
125
126 /**
127  * 1.M2 : The notes of each part should be as diverse as possible
128  */
129 void M2_1_varietyCost(Home home, vector<Part*> parts);
130
131 void P1_1_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
132
133 void P1_1_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
134
135 void P1_2_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
136
137 void P1_2_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
138
139 void P1_2_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
140
141 void P3_0_noBattuta(Home home, Part* part);
142
143 void P3_1_noBattuta(Home home, Part* part);
144
145 void P3_2_noBattuta(Home home, Part* part);
146
147 void P4_successiveCost(Home home, vector<Part*> parts, int scc_cz, IntVarArray
      ↪ successiveCostArray, vector<Species> species);
148
149 void P6_3v_noMoveInSameDirection(Home home, vector<Part*> parts);
150
151 void P6_4v_noMoveInSameDirection(Home home, vector<Part*> parts);
152
153 void P7_noSuccessiveAscendingSixths(Home home, vector<Part*> parts);
154
155 void P1_1_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
156
157 void H7_1_3v_penultimateSixthOrThird(Home home, Part* part);
158
159 /* ================================================
160  *               OTHER CONSTRAINTS
161  * ================================================
```

```
 */

/**
 * This function creates the isOff array to be able to know if a note is borrowed or not
 */
void initializeIsOffArray(Home home, Part* part);

/**
 * Two fifth species rhythms should be as diverse as possible
 */
void R9_5_twoFifthSpeciesDiversity_3v(Home home, Part* cp1, Part* cp2);

/**
 * For the upper strata, there shouldn't be a minor second interval between the thesis notes
 */
void noMinorSecondBetweenUpper(Home home, vector<Stratum*> strata);

#endif
```

Utilities.hpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
// This file contains the declarations of general utility functions.
//

#ifndef UTILITIES
#define UTILITIES

#include "gecode/kernel.hh"
#include "gecode/int.hh"
#include "gecode/search.hh"
#include "gecode/minimodel.hh"
#include "gecode/set.hh"

#include <iostream>
#include <vector>
#include <string>
#include <map>
#include <iostream>
#include <fstream>
#include <set>

using namespace std;
using namespace Gecode;

/*****************************************************
 *                                                   *
 *                 Useful constants                  *
 *                                                   *
 *****************************************************/

/** Files */
const string LOG_FILE = "log.txt";
const string STATISTICS_FILE = "statistics.txt";
const string STATISTICS_CSV = "statistics";

/** Types of search engines */
enum solver_types{
    DFS_SOLVER, //0
    BAB_SOLVER, //1
    LDS_SOLVER  //2
};
```

```
43
44  // /** Branching strategies */
45  // enum variable_selection{
46  //     DEGREE_MAX,                     //0
47  //     DOM_SIZE_MIN,                   //1
48  //     RIGHT_TO_LEFT,                  //2
49  //     LEFT_TO_RIGHT_SOPRANO_TO_BASS,  //3
50  //     AFC_MAX,                        //4
51  // };
52
53  // /// go <-- soprano->bass: 4-3-2-1-8-7-6-5 etc
54  // auto right_to_left = [](const Space& home, const IntVar& x, int i) {
55  //     return i;
56  // };
57
58  // /// go --> soprano->bass
59  // auto left_to_right_soprano_to_bass = [](const Space& home, const IntVar& x,
      ↪  int i) {
60  //     return (i/4) * 4 + (4 - i%4);
61  // };
62
63  // const vector<IntVarBranch> variable_selection_heuristics = {INT_VAR_DEGREE_MAX
      ↪  (),
64  //                                                             INT_VAR_SIZE_MIN()
      ↪  ,
65  //                                                             INT_VAR_MERIT_MAX(
      ↪  right_to_left),
66  //                                                             INT_VAR_MERIT_MIN(
      ↪  left_to_right_soprano_to_bass)};
67
68  const vector<string> variable_selection_heuristics_names = {"Degree␣max", "Domain
      ↪  ␣size␣min", "Left␣to␣right",
69                                                                "Right␣to␣left", "AFC
                                                                    ↪  ␣max"};
70
71  enum value_selection{
72      VAL_MIN,            //0
73      VAL_MAX,            //1
74      VAL_MED,            //2
75      VAL_RND,            //3
76  };
77
78  // /// value selection heuristic
79  // auto branchVal = [](const Space& home, IntVar x, int i) {
80  //     return x.min();
81  // };
82
83  // /// commit function (EQ and DIFF)
84  // auto branchCommit = [](Space& home, unsigned int a, IntVar x, int i, int n){
85  //     if (a == 0U){
86  //         rel(home, x, IRT_EQ, n);
87  //     } else {
88  //         rel(home, x, IRT_NQ, n);
89  //     }
90  // };
91
92  const vector<IntValBranch> value_selection_heuristics = {INT_VAL_MIN(),
      ↪  INT_VAL_MAX(), INT_VAL_MED(), INT_VAL_RND(1U)};
93
94  const vector<string> value_selection_heuristics_names = {"Value␣min", "Value␣max"
      ↪  , "Median␣value", "Value␣random"};
95
96  /** Voice ranges */
97  const int BASS_MIN = 40;
98  const int BASS_MAX = 60;
99  const int TENOR_MIN = 48;
100 const int TENOR_MAX = 69;
101 const int ALTO_MIN = 55;
```

```
const int ALTO_MAX = 75;
const int SOPRANO_MIN = 60;
const int SOPRANO_MAX = 84;

/** Melodic costs */
const int UNISON_COST = 0;
const int SECOND_COST = 1;
const int THIRD_COST = 3;
const int FOURTH_COST = 6;
const int FIFTH_COST = 6;
const int SIXTH_COST = 12;
const int SEVENTH_COST = 18;
const int OCTAVE_COST = 6;

const int MAX_MELODIC_COST = SEVENTH_COST;


/** Notes */
const int B_SHARP = 0;
const int C = 0;
const int C_SHARP = 1;
const int D_FLAT = 1;
const int D = 2;
const int D_SHARP = 3;
const int E_FLAT = 3;
const int E = 4;
const int F_FLAT = 4;
const int E_SHARP = 5;
const int F = 5;
const int F_SHARP = 6;
const int G_FLAT = 6;
const int G = 7;
const int G_SHARP = 8;
const int A_FLAT = 8;
const int A = 9;
const int A_SHARP = 10;
const int B_FLAT = 10;
const int B = 11;
const int C_FLAT = 11;

const vector<std::string> noteNames = {"C", "Csharp", "D", "Eb", "E", "F", "
    ↪ Fsharp", "G", "Ab", "A", "Bb", "B"};

/** Voice positions */
enum voices{
    BASS,       //0
    TENOR,      //1
    ALTO,       //2
    SOPRANO     //3
};

const vector<std::string> voiceNames = {"Bass", "Tenor", "Alto", "Soprano"};

/** scale degrees */
enum degrees{
    FIRST_DEGREE,       //0
    SECOND_DEGREE,      //1
    THIRD_DEGREE,       //2
    FOURTH_DEGREE,      //3
    FIFTH_DEGREE,       //4
    SIXTH_DEGREE,       //5
    SEVENTH_DEGREE      //6
};

const vector<std::string> degreeNames = {"First␣degree", "Second␣degree", "Third␣
    ↪ degree", "Fourth␣degree", "Fifth␣degree",
                                         "Sixth␣degree", "Seventh␣degree"};

```

```
168 /** Intervals */
169 // "classic" intervals
170 enum intervals{
171     UNISSON,            //0
172     MINOR_SECOND,       //1
173     MAJOR_SECOND,       //2
174     MINOR_THIRD,        //3
175     MAJOR_THIRD,        //4
176     PERFECT_FOURTH,     //5
177     TRITONE,            //6
178     PERFECT_FIFTH,      //7
179     MINOR_SIXTH,        //8
180     MAJOR_SIXTH,        //9
181     MINOR_SEVENTH,      //10
182     MAJOR_SEVENTH,      //11
183     PERFECT_OCTAVE      //12
184 };
185
186 // augmented/diminished intervals
187 const int AUGMENTED_SECOND = 3;
188 const int AUGMENTED_FOURTH = TRITONE;
189 const int DIMINISHED_FIFTH = TRITONE;
190
191 /** Chords */
192 enum chordTypes{
193     MAJOR_CHORD,                //0
194     MINOR_CHORD,                //1
195     DIMINISHED_CHORD,           //2
196     AUGMENTED_CHORD,            //3
197     DOMINANT_SEVENTH_CHORD,     //4
198     MAJOR_SEVENTH_CHORD,        //5
199     MINOR_SEVENTH_CHORD         //6
200 };
201
202 /// Types of chords represented by the intervals between their notes in root
      ↪ position up to an octave
203 const vector<int> MAJOR_CHORD_INTERVALS = {MAJOR_THIRD, MINOR_THIRD,
      ↪ PERFECT_FOURTH};
204 const vector<int> MINOR_CHORD_INTERVALS = {MINOR_THIRD, MAJOR_THIRD,
      ↪ PERFECT_FOURTH};
205 const vector<int> DIMINISHED_CHORD_INTERVALS = {MINOR_THIRD, MINOR_THIRD, TRITONE
      ↪ };
206 const vector<int> AUGMENTED_CHORD_INTERVALS = {MAJOR_THIRD, MAJOR_THIRD,
      ↪ MAJOR_THIRD};
207 const vector<int> DOMINANT_SEVENTH_CHORD_INTERVALS = {MAJOR_THIRD, MINOR_THIRD,
      ↪ MINOR_THIRD, MAJOR_SECOND};
208 const vector<int> MAJOR_SEVENTH_CHORD_INTERVALS = {MAJOR_THIRD, MINOR_THIRD,
      ↪ MAJOR_THIRD, MINOR_SECOND};
209 const vector<int> MINOR_SEVENTH_CHORD_INTERVALS = {MINOR_THIRD, MAJOR_THIRD,
      ↪ MINOR_THIRD, MAJOR_SECOND};
210
211 const vector<vector<int>> chordQualitiesIntervals = {MAJOR_CHORD_INTERVALS,
      ↪ MINOR_CHORD_INTERVALS,
212                                                      DIMINISHED_CHORD_INTERVALS,
                                                           ↪ AUGMENTED_CHORD_INTERVALS
                                                           ↪ ,
213                                                      DOMINANT_SEVENTH_CHORD_INTERVALS
                                                           ↪ ,
                                                           ↪ MAJOR_SEVENTH_CHORD_INTERVALS
                                                           ↪ ,
214                                                      MINOR_SEVENTH_CHORD_INTERVALS
                                                           ↪ };
215
216 const vector<std::string> chordQualityNames = {"Major", "Minor", "Diminished", "
      ↪ Augmented", "Dominant␣seventh",
217                                                 "Major␣seventh", "Minor␣seventh"};
218
219 // Chord states
```

```
enum chordStates{
    FUNDAMENTAL_STATE,  //0
    FIRST_INVERSION,    //1
    SECOND_INVERSION,   //2
    THIRD_INVERSION     //3
};

const vector<std::string> stateNames = {"Fundamental state", "First inversion", "Second inversion", "Third inversion"};

/** Modes */
enum Mode {
    IONIAN,     //0 , major mode
    DORIAN,     //1
    PHRYGIAN,   //2
    LYDIAN,     //3
    MIXOLYDIAN, //4
    AEOLIAN,    //5 , natural minor mode
    LOCRIAN     //6
};

// syntactic sugar for more commonly used modes
const int MAJOR_MODE = IONIAN;
const int MINOR_MODE = AEOLIAN;

const vector<std::string> modeNames = {"Major", "Dorian", "Phrygian", "Lydian", "Mixolydian", "Minor", "Locrian"};

/** Scales */
/// defined by the intervals between their notes

//@todo turn this into a dictionary with the name of the scale as key and the vector of intervals as value

const vector<int> MAJOR_SCALE = {MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND};
const vector<int> NATURAL_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND};
const vector<int> HARMONIC_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND, AUGMENTED_SECOND, MINOR_SECOND};
const vector<int> MELODIC_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND};
const vector<int> BORROWED_SCALE = {PERFECT_FOURTH, MAJOR_THIRD, MAJOR_SECOND, MINOR_SECOND};
const vector<int> CHROMATIC_SCALE = {MINOR_SECOND,MINOR_SECOND,MINOR_SECOND, MINOR_SECOND,MINOR_SECOND,MINOR_SECOND,MINOR_SECOND,MINOR_SECOND, MINOR_SECOND,MINOR_SECOND,MINOR_SECOND};

/** Part related */
enum Species{
    FIRST_SPECIES,  //0
    SECOND_SPECIES, //1
    THIRD_SPECIES,  //2
    FOURTH_SPECIES, //3
    FIFTH_SPECIES,  //4
    CANTUS_FIRMUS   //5
};

const map<int,int> notesPerMeasure = {{FIRST_SPECIES, 1}, {SECOND_SPECIES, 2},
                                      {THIRD_SPECIES, 4}, {FOURTH_SPECIES, 2},
                                      {FIFTH_SPECIES, 4}, {CANTUS_FIRMUS, 1}};

enum nVoices{
    TWO_VOICES,     //0
    THREE_VOICES,   //1
    FOUR_VOICES     //2
};
```

```
278
279 enum motions{
280     CONTRARY_MOTION,        //0
281     OBLIQUE_MOTION,         //1
282     PARALLEL_MOTION,        //2
283 };
284
285 const vector<int> PERFECT_CONSONANCES = {UNISSON, PERFECT_FIFTH, PERFECT_OCTAVE};
286 const vector<int> IMPERFECT_CONSONANCES = {MAJOR_THIRD, MINOR_THIRD, MAJOR_SIXTH,
      ↪  MINOR_SIXTH};
287 const vector<int> CONSONANCES = {UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH
      ↪  , MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE};
288 const vector<int> TRIAD = {UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH};
289 const vector<int> DISONANCE = {MINOR_SECOND, MAJOR_SECOND, PERFECT_FOURTH,
      ↪  TRITONE, MINOR_SEVENTH, MAJOR_SEVENTH};
290
291 const int not_harmonic_triad_cost  = 16;
292 const int double_fifths_cost       = 8;
293 const int double_thirds_cost       = 6;
294 const int triad_with_octave_cost   = 2;
295
296 enum costValues{
297     NO_COST,        //0
298     LOW_COST,       //1
299     MEDIUM_COST,    //2
300     HIGH_COST = 4,
301     LAST_RESORT = 8,
302     PROPORTIONAL_TO_SIZE = 2,   /// To multiply by size
303     FORBIDDEN = 64              /// To multiply by size
304 };
305
306 //** Max Step between two notes */
307 const int MAX_STEP = 19; // Tom Lai
308
309 //** Constraints */
310 // constraints enum
311 enum constraints{
312     CF_1H1,
313     CF_1H2_2V,
314     CF_1H3_2V,
315     CF_1H7_2V,
316     CF_1P1,
317     CF_1H7_3V,
318     CF_1P3,
319     STRATUM_UPPER_1H3,
320     STRATUM_UPPER_1H10,
321     STRATUM_UPPER_1H12,
322     STRATUM_1H3,
323     STRATUM_1H12,
324     V2_G6,
325     V2_G9,
326     V2_1H2,
327     V2_1H3,
328     V2_1H5,
329     V3_G6,
330     V3_1H4,
331     V3_1H5,
332     V3_1H8,
333     V3_1M4,
334     V3_1P4,
335     V3_1P6,
336     V3_1P7,
337     V3_2M2,
338     V3_5R9,
339     V4_G6,
340     V4_1H4,
341     V4_1H8,
342     V4_1M4,
```

```
    V4_1P4,
    V4_1P6,
    V4_1P7,
    V4_2M2,
    V4_5R9,
    V4_U2,
    SP1_G4,
    SP1_G7,
    SP1_1H1,
    SP1_1H6,
    SP1_1H7_2V,
    SP1_1M2_2V,
    SP1_1P1_2V,
    SP1_1P3_2V,
    SP1_1H7_3V,
    SP1_1M2_3V,
    SP1_1P1_3V,
    SP1_1P3_3V,
    SP1_1M2_4V,
    SP1_1P1_4V,
    SP1_1P3_4V,
    SP2_2H2,
    SP2_2M1,
    SP2_2P3,
    SP2_2H3_2V,
    SP2_2M2_2V,
    SP2_2P1_2V,
    SP2_2H3_3V,
    SP2_1P1_3V,
    SP2_2H3_4V,
    SP2_1P1_4V,
    SP3_3H1,
    SP3_3H2,
    SP3_3H3,
    SP3_3M1,
    SP3_1P3,
    SP3_U1,
    SP3_U2,
    SP3_U3,
    SP3_3H4_2V,
    SP3_1H7_3V,
    SP3_3H6_3V,
    SP3_1P1_3V,
    SP3_1H7_4V,
    SP3_3H6_4V,
    SP3_1P1_4V,
    SP4_G6,
    SP4_G7,
    SP4_1H6,
    SP4_4H1,
    SP4_4M1,
    SP4_4M2,
    SP4_4P1,
    SP4_4P2,
    SP4_1M2,
    SP4_U2,
    SP4_4H2_2V,
    SP4_4H3_2V,
    SP4_1H2_2V,
    SP4_4P5_3V,
    SP4_4P5_4V,
    SP5_3M3,
    SP5_3M4,
    SP5_H3,
    SP5_H4,
    SP5_H5,
    SP5_H6,
    SP5_M1,
```

```
    SP5_M2,
    SP5_M3,
    SP5_M4,
    SP5_P1,
    SP5_4P1,
    SP5_P3,
    SP5_P4,
    SP5_2V_1,
    SP5_2V_2,
    SP5_2V_3,
    SP5_2V_4,
    SP5_3V_1,
    SP5_3V_2,
    SP5_4V_1,
    SP5_4V_2
};

const int consSize = static_cast<int>(SP5_4V_2+1); // Number of constraints
extern vector<bool> activeConstraints;

enum toCombineConstraints{
    H1_1,
};

const vector<string> constraintNames = {
    "CF_1H1", "CF_1H2_2V", "CF_1H3_2V", "CF_1H7_2V", "CF_1P1", "CF_1H7_3V", "CF_1P3",
    "STRATUM_UPPER_1H3", "STRATUM_UPPER_1H10", "STRATUM_UPPER_1H12", "STRATUM_1H3", "STRATUM_1H12",
    "V2_G6", "V2_G9", "V2_1H2", "V2_1H3", "V2_1H5",
    "V3_G6", "V3_1H4", "V3_1H5", "V3_1H8", "V3_1M4", "V3_1P4", "V3_1P6", "V3_1P7", "V3_2M2", "V3_5R9",
    "V4_G6", "V4_1H4", "V4_1H8", "V4_1M4", "V4_1P4", "V4_1P6", "V4_1P7", "V4_2M2", "V4_5R9","V4_U2",
    "SP1_G4", "SP1_G7", "SP1_1H1", "SP1_1H6", "SP1_1H7_2V", "SP1_1M2_2V", "SP1_1P1_2V", "SP1_1P3_2V", "SP1_1H7_3V", "SP1_1M2_3V", "SP1_1P1_3V", "SP1_1P3_3V", "SP1_1M2_4V", "SP1_1P1_4V", "SP1_1P3_4V",
    "SP2_2H2", "SP2_2M1", "SP2_2P3", "SP2_2H3_2V", "SP2_2M2_2V", "SP2_2P1_2V", "SP2_2H3_3V", "SP2_1P1_3V", "SP2_2H3_4V", "SP2_1P1_4V",
    "SP3_3H1", "SP3_3H2", "SP3_3H3", "SP3_3M1", "SP3_1P3", "SP3_U1", "SP3_U2", "SP3_U3", "SP3_3H4_2V", "SP3_1H7_3V", "SP3_3H6_3V", "SP3_1P1_3V", "SP3_1H7_4V", "SP3_3H6_4V", "SP3_1P1_4V",
    "SP4_G6", "SP4_G7", "SP4_1H6", "SP4_4H1", "SP4_4M1", "SP4_4M2", "SP4_4P1", "SP4_4P2", "SP4_1M2", "SP4_U2", "SP4_4H2_2V", "SP4_4H3_2V", "SP4_1H2_2V", "SP4_4P5_3V", "SP4_4P5_4V",
    "SP5_3M3", "SP5_3M4", "SP5_H3", "SP5_H4", "SP5_H5", "SP5_H6", "SP5_M1", "SP5_M2", "SP5_M3", "SP5_M4", "SP5_P1", "SP5_4P1", "SP5_P3", "SP5_P4", "SP5_2V_1", "SP5_2V_2", "SP5_2V_3", "SP5_2V_4", "SP5_3V_1", "SP5_3V_2", "SP5_4V_1", "SP5_4V_2"
};

const vector<string> combinedCostNames = {"1H1"};

/*********************************************
 *                                           *
 *                Functions                  *
 *                                           *
 *********************************************/

/**
 * Returns the name of a constraint based on its integer value
 * @param constraint an integer representing the constraint
 * @return string the name of the constraint
 */
string get_constraint_name(int constraint);

/**
```

```
 * For a given set of intervals between notes that loops and a starting note, returns all the possible notes
 * @param note the starting note
 * @param intervals the set of intervals between notes. It must make a loop. For example, to get all notes from a major
 * scale from note, use {2, 2, 1, 2, 2, 2, 1}. To get all notes from a minor chord, use {3, 4, 5}.
 * @return vector<int> all the notes
 */
vector<int> get_all_notes_from_interval_loop(int n, vector<int> intervals);

/**
 * For a given tonality (root + mode), returns all the possible notes
 * @param root the root of the tonality (in [0,11])
 * @param scale the set of tones and semitones that define the scale
 * @return vector<int> all the possible notes from that tonality
 */
vector<int> get_all_notes_from_scale(int root, vector<int> scale);

/**
 * For a given chord (root + mode), returns all the possible notes
 * @param root the root of the chord
 * @param quality the set of tones and semitones that define the chord
 * @return vector<int> all the possible notes from that chord
 */
vector<int> get_all_notes_from_chord(int root, vector<int> quality);

/**
 * Get all values for a given note
 * @param note a note
 * @return vector<int> a vector containing all the given notes
 */
vector<int> get_all_given_note(int note);

/**
 * Transforms an int* into a vector<int>
 * @param ptr an int* pointer
 * @param size the size of the array
 * @return a vector<int> containing the same values as the array
 */
vector<int> int_pointer_to_vector(int* ptr, int size);

/**
 * Union algorithm found and adapted from GeeksForGeeks.com
 */
vector<int> vector_union(vector<int> v1, vector<int> v2);

/**
 * Intersection algorithm found and adapted from GeeksForGeeks.com
 */
vector<int> vector_intersection(vector<int> v1, vector<int> v2);

/**
 * Difference algorithm found and adapted from GeeksForGeeks.com
 */
vector<int> vector_difference(vector<int> v1, int lb, int ub);

/**
 * Transforms a vector of integers into a string
 * @param vector a vector of integers
 * @return string the string representation of the vector
 */
string int_vector_to_string(vector<int> vector);

/**
 * Prints the Search::Statistics object into a readable format
 * @param stats a Search::Statistics object representing the statistics of a search
```

```
528  * @return The string representation of the statistics object
529  */
530 string statistics_to_string(Search::Statistics stats);
531
532 /**
533  * Prints the Search::Statistics object into a csv format (coma separated)
534  * @param stats a Search::Statistics object representing the statistics of a
       ↪ search
535  * @return The string representation of the statistics object
536  */
537 string statistics_to_csv_string(Search::Statistics stats);
538
539 /**
540  * Returns the value of a variable as a string
541  * @param var an integer variable
542  * @param absolute a boolean indicating if the value should be returned as an
       ↪ absolute value (default is false)
543  * @return a string representing the value of the variable
544  */
545 string intVar_to_string(const IntVar &var, bool absolute = false);
546
547 /**
548  * Returns the values of an array of variables as a string. Calls the
       ↪ intVar_to_string function
549  * @param vars an array of integer variables
550  * @return a string representing the values of the variables
551  */
552 string intVarArray_to_string(IntVarArray vars);
553
554 /**
555  * Returns the value of a variable as a string
556  * @param var an integer variable
557  * @param absolute a boolean indicating if the value should be returned as an
       ↪ absolute value (default is false)
558  * @return a string representing the value of the variable
559  */
560 string boolVar_to_string(const BoolVar &var, bool absolute = false);
561
562 /**
563  * Returns the values of an array of variables as a string
564  * @param vars an array of integer variables
565  * @return a string representing the values of the variables
566  */
567 string boolVarArray_to_string(BoolVarArray vars);
568
569 /**
570  * Returns the values of an array of variables as a string
571  * @param vars an array of integer variables
572  * @return a string representing the values of the variables, ignoring unassigned
       ↪ variables
573  */
574 string cleanIntVarArray_to_string(IntVarArray vars);
575
576 /**
577  * Returns the values of an IntVarArgs as a string
578  * @param args an IntVarArgs
579  * @return a string representing the values
580  */
581 string intVarArgs_to_string(IntVarArgs args);
582
583 /**
584  * Returns the name of a note based on its MIDI value
585  * @param note an integer
586  */
587 string midi_to_letter(int note);
588
589 /**
590  * Returns the name of a mode based on its integer value
```

```
 * @param mode an integer
 * @return a string representing the name of the mode
 */
string mode_int_to_name(int mode);

/**
 * returns a string with the time
 * @return a string with the time
 */
string time();

/**
 * Write a text into a log file
 * @param message the text to write
 */
void write_to_log_file(const char *message, const string& filename);

void writeToLogFile(const char* message);


#endif
```

CounterpointUtils.hpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
// This file contains the declarations of the main functions to create problems and counterpoints.
//

#ifndef COUNTERPOINTUTILS_HPP
#define COUNTERPOINTUTILS_HPP

#include "Utilities.hpp"
#include "CounterpointProblems/TwoVoiceCounterpoint.hpp"
#include "CounterpointProblems/ThreeVoiceCounterpoint.hpp"
#include "CounterpointProblems/FourVoiceCounterpoint.hpp"
#include "Parts/FirstSpeciesCounterpoint.hpp"
#include "Parts/SecondSpeciesCounterpoint.hpp"
#include "Parts/ThirdSpeciesCounterpoint.hpp"
#include "Parts/FourthSpeciesCounterpoint.hpp"
#include "Parts/FifthSpeciesCounterpoint.hpp"

using namespace std;
using namespace Gecode;


/**
 * This function creates the appropriate Part given the species of the counterpoint requested.
 * @return a pointer to a Part object
*/
Part* create_counterpoint(Home home, int species, int nMeasures, vector<int> cantusFirmus, int lowerBound, int upperBound, Stratum* low,
    CantusFirmus* c, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV);

/**
 * This function creates the appropriate counterpoint problem given the number of counterpoints (size of the species list) requested.
 * @return a pointer to a counterpoint problem object
*/
CounterpointProblem* create_problem(vector<int> cf, vector<Species> sp, vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
    vector<int> s_costs, vector<int> imp, int bm);
```

```
36
37 bool notInt(char* argv);
38
39 bool has_solution(CounterpointProblem* problem);
40
41
42
43 #endif
```

fuxTest.hpp

```
1     //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file is the header file of the testing framework implementation.
4 //
5
6 #ifndef FUX_TESTS_HPP
7 #define FUX_TESTS_HPP
8
9 #include "Utilities.hpp"
10 #include "Parts/Part.hpp"
11 #include "CounterpointUtils.hpp"
12 #include "CounterpointProblems/CounterpointProblem.hpp"
13 #include "Parts/CantusFirmus.hpp"
14
15 using namespace Gecode;
16 using namespace std;
17
18 class FuxTest{
19
20 protected:
21     vector<Species> spList;
22     vector<int> cantusFirmus;
23     vector<int> v_type;
24     int borrowMode;
25     vector<int> cp;
26     int idx;
27     int cfSize;
28     vector<int> melodic_params;
29     vector<int> general_params;
30     vector<int> specific_params;
31     vector<int> importance;
32
33 public:
34
35 // ========== Modified by Tom ====================
36
37     void test_G6();
38     void test_G6_2v_1sp();
39
40     void test_1H1();
41     void test_1H1_2v_1sp();
42     void test_1H1_3v_1sp();
43     void test_1H1_4v_1sp();
44     void test_1H1_2v_2sp();
45     void test_1H1_3v_2sp();
46     void test_1H1_4v_2sp();
47     void test_1H1_2v_3sp();
48     void test_1H1_3v_3sp();
49     void test_1H1_4v_3sp();
50     void test_1H1_2v_4sp();
51     void test_1H1_3v_4sp();
52     void test_1H1_4v_4sp();
53     void test_1H1_2v_5sp();
54     void test_1H1_3v_5sp();
```

```
55      void test_1H1_4v_5sp();
56
57      void test_1H2();
58      void test_1H2_2v_1sp();
59      void test_1H2_2v_2sp();
60      void test_1H2_2v_3sp();
61      void test_1H2_2v_4sp();
62
63      void test_1H3();
64      void test_1H3_2v_1sp();
65      void test_1H3_3v_1sp();
66      void test_1H3_4v_1sp();
67      void test_1H3_2v_2sp();
68      void test_1H3_3v_2sp();
69      void test_1H3_4v_2sp();
70      void test_1H3_2v_3sp();
71      void test_1H3_3v_3sp();
72      void test_1H3_4v_3sp();
73      void test_1H3_2v_4sp();
74      void test_1H3_3v_4sp();
75      void test_1H3_4v_4sp();
76
77      void test_1H4();
78      void test_1H4_2v_1sp();
79      void test_1H4_3v_1sp();
80      void test_1H4_4v_1sp();
81      void test_1H4_2v_2sp();
82      void test_1H4_3v_2sp();
83      void test_1H4_4v_2sp();
84      void test_1H4_2v_3sp();
85      void test_1H4_3v_3sp();
86      void test_1H4_4v_3sp();
87      void test_1H4_2v_4sp();
88      void test_1H4_3v_4sp();
89      void test_1H4_4v_4sp();
90
91      void test_1H5();
92      void test_1H5_2v_1sp();
93      void test_1H5_3v_1sp();
94      void test_1H5_2v_2sp();
95      void test_1H5_3v_2sp();
96      void test_1H5_2v_3sp();
97      void test_1H5_3v_3sp();
98      void test_1H5_2v_4sp();
99      void test_1H5_3v_4sp();
100
101     void test_1H6();
102     void test_1H6_2v_1sp();
103
104     void test_1H7();
105     void test_1H7_2v_1sp();
106     void test_1H7_3v_1sp();
107     void test_1H7_2v_2sp();
108     void test_1H7_3v_2sp();
109     void test_1H7_2v_3sp();
110     void test_1H7_3v_3sp();
111     void test_1H7_2v_4sp();
112     void test_1H7_3v_4sp();
113
114     void test_1H10();
115     void test_1H10_3v_1sp();
116     void test_1H10_3v_2sp();
117     void test_1H10_3v_3sp();
118     void test_1H10_3v_4sp();
119
120     void test_1M2();
121     void test_1M2_2v_1sp();
122     void test_1M2_3v_1sp();
```

```
123     void test_1M2_4v_1sp();
124
125     void test_1P1_2v_1sp();
126
127     void test_2H2();
128     void test_2H2_2v_2sp();
129     void test_2H2_3v_2sp();
130     void test_2H2_4v_2sp();
131
132     void test_2M2();
133     void test_2M2_2v_2sp();
134
135     void test_2v_1sp_fig22();
136     void test_2v_1sp_fig23();
137     void test_2v_2sp_fig38();
138     void test_2v_2sp_fig39();
139     void test_2v_2sp_fig40();
140     void test_2v_2sp_fig41();
141     void test_2v_2sp_fig42();
142     void test_2v_2sp_fig43();
143     void test_2v_2sp_fig44();
144     void test_2v_2sp_fig45();
145     void test_2v_3sp_fig55();
146     void test_2v_3sp_fig56();
147     void test_2v_3sp_fig57();
148     void test_2v_3sp_fig58();
149     void test_2v_3sp_fig59();
150     void test_2v_3sp_fig60();
151     void test_2v_4sp_fig74();
152     void test_2v_4sp_fig75();
153     void test_2v_4sp_fig76();
154     void test_2v_4sp_fig77();
155     void test_2v_4sp_fig78();
156     void test_3v_1sp_fig108();
157     void test_3v_1sp_fig109();
158     void test_3v_1sp_fig110();
159     void test_3v_1sp_fig111();
160     void test_3v_1sp_fig112();
161     void test_3v_1sp_fig113();
162     void test_3v_1sp_fig114();
163     void test_3v_1sp_fig115();
164     void test_3v_1sp_fig116();
165     void test_3v_1sp_fig117();
166     void test_3v_1sp_fig118();
167     void test_3v_1sp_fig119();
168     void test_3v_2sp_fig125();
169     void test_3v_2sp_fig128();
170     void test_3v_2sp_fig129();
171     void test_3v_3sp_fig132();
172     void test_3v_3sp_fig133();
173     void test_3v_4sp_fig150();
174     void test_3v_4sp_fig151();
175     void test_4v_1sp_fig171();
176     void test_4v_1sp_fig172();
177     void test_4v_2sp_fig175();
178     void test_4v_2sp_fig176();
179     void test_4v_3sp_fig184();
180     void test_4v_3sp_fig186();
181     void test_4v_4sp_fig193();
182     void test_4v_4sp_fig196();
183
184
185     int get_motions(CounterpointProblem* problem, int i);
186
187 // ================================================
188
189     FuxTest(char* test);
190
```

```
191     FuxTest(int testNumber);
192
193     FuxTest(int testNumber, int i);
194
195     vector<Species> getSpList();
196     vector<int> getCf();
197     vector<int> getVType();
198     int getBMode();
199     vector<int> getCp();
200     int getIdx();
201
202     CounterpointProblem* test_1sp_H1();
203     CounterpointProblem* test_1sp_H2();
204     CounterpointProblem* test_1sp_H3();
205     CounterpointProblem* test_1sp_H3_2();
206     CounterpointProblem* test_1sp_H4_1();
207     CounterpointProblem* test_1sp_H4_2();
208     CounterpointProblem* test_1sp_H5();
209     CounterpointProblem* test_1sp_H7();
210     CounterpointProblem* test_1sp_H7_2();
211     CounterpointProblem* test_2sp_H2();
212     CounterpointProblem* test_3sp_H1();
213     CounterpointProblem* test_4sp_H2();
214
215     CounterpointProblem* dispatcher(char* test);
216
217     void test_2v_1sp_fig22_setter(int i);
218     void test_2v_1sp_fig23_setter(int i);
219
220     void test_2v_2sp_fig38_setter(int i);
221     void test_2v_2sp_fig39_setter(int i);
222     void test_2v_2sp_fig40_setter(int i);
223
224     void test_2v_3sp_fig55_setter(int i);
225
226     void test_2v_4sp_fig74_setter(int i);
227
228     void test_2v_5sp_fig82_setter(int i);
229
230     void test_3v_1sp_fig108_setter(int i);
231     void test_3v_1sp_fig109_setter(int i);
232     void test_3v_1sp_fig110_setter(int i);
233     void test_3v_1sp_fig111_setter(int i);
234
235     void test_3v_2sp_fig125_setter(int i);
236
237     void test_3v_4sp_fig146_setter(int i);
238
239     void test_4v_1sp_fig166_setter(int i);
240     void test_4v_1sp_fig167_setter(int i);
241
242     void test_4v_2sp_fig176_setter(int i);
243
244     void test_configuration();
245 };
246
247 void printVector(const std::vector<int>& array);
248
249 void printIntVarArray(const IntVarArray& array);
250
251 std::vector<CounterpointProblem*> get_all_solutions(CounterpointProblem* problem)
      ↪ ;
252
253
254 #endif
```

figureTests.hpp

```
    #ifndef FIGURE_TESTS_HPP
#define FIGURE_TESTS_HPP

#include "fuxTest.hpp"

class FigureTests {
private:
    std::vector<Species> spList;
    std::vector<int> cantusFirmus;
    std::vector<int> cp;
    std::vector<int> v_type;
    std::vector<int> melodic_params;
    std::vector<int> general_params;
    std::vector<int> specific_params;
    std::vector<int> importance;
    std::vector<int> notesSpeciesFor5sp; // species for each notes in the 5th species
    int borrowMode;
    void test_configuration();
    CounterpointProblem* set_configuration();
    std::vector<CounterpointProblem*> get_all_solutions();
    bool is_unsat(const set<int>& cons_set);
    set<int> minimize (const std::set<int>& activeCons);
    void findAllMUSes();
    void find_unsat_constraints();

public:
    FigureTests();

    // Two voice first species figures
    void test_2v_1sp_fig22();
    void test_2v_1sp_fig23();

    // Two voice second species figures
    void test_2v_2sp_fig38();
    void test_2v_2sp_fig39();
    void test_2v_2sp_fig40();
    void test_2v_2sp_fig41();
    void test_2v_2sp_fig42();
    void test_2v_2sp_fig43();
    void test_2v_2sp_fig44();
    void test_2v_2sp_fig45();

    // Two voice third species figures
    void test_2v_3sp_fig55();
    void test_2v_3sp_fig56();
    void test_2v_3sp_fig57();
    void test_2v_3sp_fig58();
    void test_2v_3sp_fig59();
    void test_2v_3sp_fig60();

    // Two voice fourth species figures
    void test_2v_4sp_fig74();
    void test_2v_4sp_fig75();
    void test_2v_4sp_fig76();
    void test_2v_4sp_fig77();
    void test_2v_4sp_fig78();

    // Two voice fifth species figures
    void test_2v_5sp_fig82();
    void test_2v_5sp_fig83();
    void test_2v_5sp_fig87_1();

    // Three voice first species figures
    void test_3v_1sp_fig108();
```

```
65      void test_3v_1sp_fig109();
66      void test_3v_1sp_fig110();
67      void test_3v_1sp_fig111();
68      void test_3v_1sp_fig112();
69      void test_3v_1sp_fig113();
70      void test_3v_1sp_fig114();
71      void test_3v_1sp_fig115();
72      void test_3v_1sp_fig116();
73      void test_3v_1sp_fig117();
74      void test_3v_1sp_fig118();
75      void test_3v_1sp_fig119();
76
77      // Three voice second species figures
78      void test_3v_2sp_fig125();
79      void test_3v_2sp_fig126();
80      void test_3v_2sp_fig127();
81      void test_3v_2sp_fig128();
82      void test_3v_2sp_fig129();
83
84      // Three voice third species figures
85      void test_3v_3sp_fig130();
86      void test_3v_3sp_fig131();
87      void test_3v_3sp_fig132();
88      void test_3v_3sp_fig133();
89
90      // Three voice fourth species figures
91      void test_3v_4sp_fig146();
92      void test_3v_4sp_fig147();
93      void test_3v_4sp_fig148();
94      void test_3v_4sp_fig149();
95      void test_3v_4sp_fig150();
96      void test_3v_4sp_fig151();
97
98      // Three voice fifth species figures
99      void test_3v_5sp_fig154();
100     void test_3v_5sp_fig155();
101     void test_3v_5sp_fig156();
102     void test_3v_5sp_fig157();
103
104     // Four voice first species figures
105     void test_4v_1sp_fig166();
106     void test_4v_1sp_fig167();
107     void test_4v_1sp_fig168();
108     void test_4v_1sp_fig169();
109     void test_4v_1sp_fig170();
110     void test_4v_1sp_fig171();
111     void test_4v_1sp_fig172();
112
113     // Four voice second species figures
114     void test_4v_2sp_fig173();
115     void test_4v_2sp_fig174();
116     void test_4v_2sp_fig175();
117     void test_4v_2sp_fig176();
118
119     // Four voice third species figures
120     void test_4v_3sp_fig183();
121     void test_4v_3sp_fig184();
122     void test_4v_3sp_fig185();
123     void test_4v_3sp_fig186();
124
125     // Four voice fourth species figures
126     void test_4v_4sp_fig196();
127
128     // Four voice fifth species figures
129     void test_4v_5sp_fig200();
130     void test_4v_5sp_fig201();
131
132     // Multispecies figures
```

```
    void test_4v_Xsp_fig204();

    // Run all figure tests
    void run_all_tests();

    void run_twoVoice_tests();
    void run_threeVoice_tests();
    void run_fourVoice_tests();

    void run_thirdSpecies_tests();
    void run_fourthSpecies_tests();
    void run_fifthSpecies_tests();

    void MUSTest();

    void quickTest();

    // Add new method for testing cost summing
    void test_cost_summing();
};

#endif // FIGURE_TESTS_HPP
```

problem_wrapper.hpp

```
    //
// This file contains the declarations of the C++ interface functions.
//

#ifndef gecode_wrapper_hpp
#define gecode_wrapper_hpp

#include <stdlib.h>

#ifdef __cplusplus
extern "C" {
#endif

/**
 * Wraps the Problem constructor.
 * @todo modify this to include any parameters your Problem constructor requires
 * @param size an integer representing the size of the problem
 * @param lower_bound_domain an integer representing the lower bound of the domain of the variables
 * @param upper_bound_domain an integer representing the upper bound of the domain of the variables
 * @return A pointer to a Problem object casted as a void*
 */
void* create_new_problem(int* cantusFirmus, int size, int n_cp, int* splist, int* v_types, int b_mode, int min_skips, int* general_params,
        int* motion_params, int* melodic, int* specific, int* importance, int t_off, int* scle, int scale_size,
        int* chromatic, int chrom_size, int* borrow, int borrow_size);

/**
 * returns the size of the problem
 * @param sp a void* pointer to a Problem object
 * @return an integer representing the size of the problem
 */
int get_size(void* sp);

void delete_pointer(void* p);

void delete_solver_pointer(void* p);

```

```
37
38  /**
39   * returns the values of the variables for a solution
40   * @param sp a void* pointer to a Problem object
41   * @return an int* pointer representing the values of the variables
42   */
43  int* return_solution(void* sp);
44
45  /**
46   * creates a search engine for Problem objects
47   * @param sp a void* pointer to a Problem object
48   * @return a void* cast of a Base<Problem>* pointer
49   */
50  void* create_solver(void* sp, int type);
51
52  /**
53   * returns the next solution space, it should be bound. If not, it will return
       ↪ NULL.
54   * @param solver a void* pointer to a Base<Problem>* pointer for the search
       ↪ engine of the problem
55   * @return a void* cast of a Problem* pointer
56   */
57  void* return_next_solution_space(void* solver);
58
59
60  int search_stopped(void* solver);
61
62
63  int* return_species_array_5sp(void* sp, int ctp_index);
64  int* return_extended_cp_domain(void* sp, int ctp_index);
65  int get_extended_cp_domain_size(void* sp, int ctp_index);
66
67
68  #ifdef __cplusplus
69  };
70  #endif
71
72  #endif
```

A.4 Counterpoint source files

CounterpointProblem.cpp

```
1     //
2   // Created by Luc Cleenewerk and Diego de Patoul.
3   //
4
5   #include "../../headers/CounterpointProblems/CounterpointProblem.hpp"
6   #include "../../headers/CounterpointUtils.hpp"
7
8   /**
9    * Constructor of the class.
10  * @param cf a vector<int> representing the cantus firmus.
11  * @param k the key of the score. it takes values from the notes in headers/
      ↪ Utilities.hpp
12  * @param lb the lowest note possible for the counterpoint in MIDI
13  * @param ub the highest note possible for the counterpoint in MIDI
14  */
15  CounterpointProblem::CounterpointProblem(vector<int> cf, int v_type, vector<int>
      ↪ m_costs, vector<int> g_costs, vector<int> s_costs,
16      vector<int> imp, int nV){
17      nMeasures = cf.size();
18      lowest = new Stratum(*this, nMeasures, 0, 127);
```

```
19      cantusFirmus = new CantusFirmus(*this, nMeasures, cf, lowest, v_type, m_costs
          ↪ , g_costs, s_costs, nV);
20      importance = imp;
21      n_unique_costs = 0;
22      importanceNames = {"borrow", "fifth", "octave", "succ", "variety", "triad", "
          ↪ direct", "motion", "penult", "cambiata", "triad3" ,"m2", "syncopation"
          ↪ , "melodic"};
23
24      //creating the map with the names of the costs and their importance
25
26      prefs = {};
27
28      setPreferenceMap(importanceNames);
29
30      orderedFactors = IntVarArray(*this, 14, 0, 1000);   //orderedFactors will
          ↪ contain the costs in order of their importance
31      for(int i = 0; i < 14; i++){
32          vector<string> tmp = {};
33          costLevels.push_back(tmp);                      //costLevels will contain
              ↪  all the costs at a specific level of importance (1 to 14)
34      }
35
36      for(const auto& entry : prefs){
37          int val = entry.second-1;
38          costLevels[val].push_back(entry.first);         //initialize the
              ↪ costLevels
39      }
40
41      globalCost = IntVar(*this, 0, 2000000);             //contains the global
          ↪ cost
42
43      writeToLogFile("counterpointproblem_constructor");
44
45      // Constraints Cost
46      combinedCosts = IntVarArray(*this, combinedCostNames.size(), 0, 10000);
          ↪          //combinedCosts will contain the costs of the different
          ↪ counterpoints
47
48      // 1H1
49      // combined constraints has to be respect user preferences
50      int respect_percent_1H1 = 50; //50 allowed every Fux figures tested (figures
          ↪ 60 is the worst)
51      //TODO: should be replaced by the user preference
52
53      int nCP = 1; // number of counterpoints
54      if (counterpoint_2 != nullptr) {
55          nCP ++;
56      }
57      if (counterpoint_3 != nullptr) {
58          nCP ++;
59      }
60      int n_thesis_notes = this->nMeasures * nCP;
61      int threshold = n_thesis_notes - (respect_percent_1H1 * n_thesis_notes) /
          ↪ 100;
62      rel(*this, combinedCosts[H1_1], IRT_LQ, threshold); // 1H1
63  }
64
65  // COPY CONSTRUCTOR
66  CounterpointProblem::CounterpointProblem(CounterpointProblem& s) :
      ↪ IntLexMinimizeSpace(s){
67      if (s.cantusFirmus) {
68          cantusFirmus = s.cantusFirmus->clone(*this);
69      } else {
70          cantusFirmus = nullptr;
71      }
72      if (s.lowest) {
73          lowest = s.lowest->clone(*this);
74      } else {
```

```
75          lowest = nullptr;
76      }
77      if (s.counterpoint_1) {
78          counterpoint_1 = s.counterpoint_1->clone(*this);
79      } else {
80          counterpoint_1 = nullptr;
81      }
82      if (s.counterpoint_2) {
83          counterpoint_2 = s.counterpoint_2->clone(*this);
84      } else {
85          counterpoint_2 = nullptr;
86      }
87      if (s.counterpoint_3) {
88          counterpoint_3 = s.counterpoint_3->clone(*this);
89      } else {
90          counterpoint_3 = nullptr;
91      }
92      if(s.upper_1){
93          upper_1 = s.upper_1->clone(*this);
94      } else {
95          upper_1 = nullptr;
96      }
97      if(s.upper_2){
98          upper_2 = s.upper_2->clone(*this);
99      } else {
100         upper_2 = nullptr;
101     }
102     if(s.upper_3){
103         upper_3 = s.upper_3->clone(*this);
104     } else {
105         upper_3 = nullptr;
106     }
107     nMeasures = s.nMeasures;
108     importance = s.importance;
109     prefs = s.prefs;
110     sorted_voices = s.sorted_voices;
111     combinedCosts = s.combinedCosts;
112     unitedCostNames = s.unitedCostNames;
113     costLevels = s.costLevels;
114     n_unique_costs = s.n_unique_costs;
115     importanceNames = s.importanceNames;
116     combinedCosts.update(*this, s.combinedCosts);
117     successiveCostArray.update(*this, s.successiveCostArray);
118     triadCostArray.update(*this, s.triadCostArray);
119     unitedCosts.update(*this, s.unitedCosts);
120     solutionArray.update(*this, s.solutionArray);
121     sortedCosts.update(*this, s.sortedCosts);
122     orderedFactors.update(*this, s.orderedFactors);
123     finalCosts.update(*this, s.finalCosts);
124     for(int i = 0; i < sorted_voices.size(); i++){
125         sorted_voices[i].update(*this, s.sorted_voices[i]);
126     }
127     globalCost.update(*this, s.globalCost);
128 }
129
130 IntLexMinimizeSpace* CounterpointProblem::copy(){
131     return new CounterpointProblem(*this);
132 }
133
134 void CounterpointProblem::constrain(const IntLexMinimizeSpace& _b){
135
136     const CounterpointProblem &b = dynamic_cast<const CounterpointProblem &>(_b);
137
138 }
139
140 IntVarArgs CounterpointProblem::cost() const{
141
142     return IntVarArgs(finalCosts);
```

```
143
144 }
145
146 string CounterpointProblem::to_string() const {
147     string text = "Counterpoint problem : \n";
148     text += cantusFirmus->to_string();
149     text += "\n";
150     text += "All costs : \n";
151     text += intVarArray_to_string(unitedCosts);
152     text += "\nLowest : \n";
153     text += lowest->to_string();
154     text += "\n";
155     text += "Upper : \n";
156     text += upper_1->to_string();
157     text += "\n";
158     text += "Final Costs : \n";
159     text += intVarArray_to_string(finalCosts);
160     text += "\n";
161     return text;
162 }
163
164 Home CounterpointProblem::getHome(){
165     return *this;
166 }
167
168 void CounterpointProblem::setPreferenceMap(vector<string> importance_names){
169     for(int i = 0; i < importance.size(); i++){
170         prefs.insert({importance_names[i], importance[i]});
171     }
172 }
173
174 void CounterpointProblem::orderCosts(){
175     for(int i = 0; i < 14; i++){
176         if(!costLevels[i].empty()){
177             int sm_size = 0;
178             for(int k = 0; k < costLevels[i].size(); k++){
179                 //goes through every cost at the cost level
180                 for(int t = 0; t < unitedCostNames.size(); t++){
181                     if(unitedCostNames[t]==costLevels[i][k]){
182                         //adjusts the amount of costs at this level
183                         sm_size++;
184                     }
185                 }
186             }
187             //now cst_sm_size contains the size of all the cost array on this
                  ↪ level combined
188             IntVarArgs sm(sm_size); //sum of all the costs on this level
189             int idx = 0;
190             for(int k = 0; k < costLevels[i].size(); k++){
191                 //goes through every cost at the cost level
192                 for(int t = 0; t < unitedCostNames.size(); t++){
193                     if(unitedCostNames[t]==costLevels[i][k]){
194                         //add the cost to the intvarargs
195                         sm[idx] = unitedCosts[t];
196                         idx++;
197                     }
198                 }
199             }
200             //if there is at least one cost at this level
201             if(idx>0){
202                 //then sum all the intvarargs of this level together and add them
                      ↪   to the orderedFactors
203                 rel(*this, orderedFactors[n_unique_costs], IRT_EQ, expr(*this,
                      ↪  sum(sm)));
204                 //increase the amount of unique costs
205                 n_unique_costs++;
206             }
207         }
```

```
208     }
209
210     finalCosts = IntVarArray(*this, n_unique_costs, 0, 1000000);
211     for(int i = 0; i < n_unique_costs; i++){
212         //adds the cost/costs to the finalCosts list, which will be the one that
              ↪ will be minimized
213         //(this eliminates all the <not assigned> values of the intvararray for
              ↪ costs which are not set)
214         rel(*this, finalCosts[i], IRT_EQ, orderedFactors[i]);
215     }
216     //globalCost is the sum of all the finalCosts
217     rel(*this, globalCost, IRT_EQ, expr(*this, sum(finalCosts)));
218 }
219
220
221 void CounterpointProblem::setStrata(){
222     // Determine number of voices
223     int nVoices = 2;
224     if(upper_2 != nullptr) nVoices++;
225     if(upper_3 != nullptr) nVoices++;
226
227     int size = counterpoint_1->getNMeasures();
228
229     // Initialize measure_orders for each measure
230     sorted_voices = {};
231     measures_order = {};
232
233     // Process each measure
234     for(int i = 0; i < size; i++){
235         // For each measure, create voices array with the actual notes at that
              ↪ measure
236         IntVarArray voices = IntVarArray(*this, nVoices, 0, 127);
237
238         // Set cantus firmus note
239         rel(*this, voices[0], IRT_EQ, cantusFirmus->getNotes()[i]);
240
241         // Set counterpoint notes based on species-specific rules for this
              ↪ measure
242         setVoiceNote(voices, 1, counterpoint_1, i);
243         if(nVoices >= 3) setVoiceNote(voices, 2, counterpoint_2, i);
244         if(nVoices >= 4) setVoiceNote(voices, 3, counterpoint_3, i);
245
246         // Sort voices and create order array for this specific measure
247         measures_order.push_back(IntVarArray(*this, nVoices, 0, nVoices-1));
248         sorted_voices.push_back(IntVarArray(*this, nVoices, 0, 127));
249         sorted(*this, voices, sorted_voices[i], measures_order[i]);
250
251         // Set stratum constraints for each position in the measure
252         int maxPos = (i == size-1) ? 1 : 4;  // Only set first position for last
              ↪ measure
253         for(int pos = 0; pos < maxPos; pos++){
254             // Set stratum constraints using the position-specific ordering
255             setStrataAtPosition(i, pos, nVoices, measures_order[i]);
256         }
257
258         // Set voice flags and melodic interval constraints (only once per
              ↪ measure)
259         setVoiceLowestFlags(i, nVoices, size);
260         if(i > 0) setMelodicIntervalConstraints(i, nVoices);
261     }
262 }
263
264 // Helper function implementations for setStrata
265
266 void CounterpointProblem::setVoiceNote(IntVarArray& voices, int voiceIndex, Part*
      ↪   part, int measureIndex) {
267     // Set the appropriate note for a voice at a specific position within the
        ↪ measure
```

```
    if ((part->getSpecies() == FOURTH_SPECIES || part->getSpecies() ==
      ↪  FIFTH_SPECIES) && measureIndex == 0) {
        // First measure: use resolution note (index 2) for fourth/fifth species
        rel(*this, voices[voiceIndex], IRT_EQ, part->getNotes()[(measureIndex*4)
          ↪  +2]);
    } else if (part->getSpecies() == FOURTH_SPECIES && measureIndex != 0 &&
      ↪  measureIndex != nMeasures-1) {
        // Middle measures of fourth species: choose between thesis and arsis
          ↪  based on joined degree
        BoolVar isJoinedDegree(*this, 0, 1);
        rel(*this, expr(*this, abs(part->getNotes()[measureIndex*4+2] - part->
          ↪  getNotes()[measureIndex*4])), IRT_LQ, 2, Reify(isJoinedDegree));
        rel(*this, (isJoinedDegree == 0) >> (voices[voiceIndex] == part->getNotes
          ↪  ()[measureIndex*4]));      // thesis
        rel(*this, (isJoinedDegree == 1) >> (voices[voiceIndex] == part->getNotes
          ↪  ()[(measureIndex*4)+2])); // arsis
    } else if (part->getSpecies() == FIFTH_SPECIES && measureIndex != 0 &&
      ↪  measureIndex != nMeasures-1) {
        // Fifth species: use species at this specific position
        IntVar currentSpecies(*this, -1, 4);
        rel(*this, currentSpecies, IRT_EQ, part->getSpeciesArray()[measureIndex
          ↪  *4]);
        // Fourth species logic for fifth species
        BoolVar isJoinedDegree(*this, 0, 1);
        rel(*this, expr(*this, abs(part->getNotes()[measureIndex*4+2] - part->
          ↪  getNotes()[measureIndex*4])), IRT_LQ, 2, Reify(isJoinedDegree));
        rel(*this, (currentSpecies == FOURTH_SPECIES && isJoinedDegree == 0) >> (
          ↪  voices[voiceIndex] == part->getNotes()[measureIndex*4]));
        rel(*this, (currentSpecies == FOURTH_SPECIES && isJoinedDegree == 1) >> (
          ↪  voices[voiceIndex] == part->getNotes()[measureIndex*4+2]));
        // Third species logic for fifth species
        rel(*this, (currentSpecies != FOURTH_SPECIES) >> (voices[voiceIndex] ==
          ↪  part->getNotes()[measureIndex*4]));
    } else {
        // Default case: use first note of the measure
        rel(*this, voices[voiceIndex], IRT_EQ, part->getFirstNotes()[measureIndex
          ↪  ]);
    }
}

void CounterpointProblem::setStrataAtPosition(int measureIndex, int position, int
  ↪   nVoices, IntVarArray& order) {
    // The order array tells us: order[voice_index] = position_in_sorted_order
    // We need to find which voice is at each sorted position (0=lowest, 1=middle
      ↪  , etc.)

    // For the first beat of each measure, we already computed the note that
      ↪  really matter for each stratum
    if (position == 0) {
        rel(*this, lowest->getNotes()[measureIndex*4+position], IRT_EQ,
          ↪  sorted_voices[measureIndex][0]);
        rel(*this, upper_1->getNotes()[measureIndex*4+position], IRT_EQ,
          ↪  sorted_voices[measureIndex][1]);
        if(nVoices >= 3) rel(*this, upper_2->getNotes()[measureIndex*4+position],
          ↪   IRT_EQ, sorted_voices[measureIndex][2]);
        if(nVoices >= 4) rel(*this, upper_3->getNotes()[measureIndex*4+position],
          ↪   IRT_EQ, sorted_voices[measureIndex][3]);

    } else {
        // Lowest stratum
        BoolVar cfIsLowest(*this, 0, 1);
        BoolVar cp1IsLowest(*this, 0, 1);
        BoolVar cp2IsLowest(*this, 0, 1);
        BoolVar cp3IsLowest(*this, 0, 1);

        rel(*this, order[0], IRT_EQ, 0, Reify(cfIsLowest));      // cantus firmus
          ↪   is at position 0 (lowest)
```

```
313         rel(*this, order[1], IRT_EQ, 0, Reify(cp1IsLowest));     //
              ↪  counterpoint_1 is at position 0 (lowest)
314         if(nVoices >= 3) rel(*this, order[2], IRT_EQ, 0, Reify(cp2IsLowest)); //
              ↪  counterpoint_2 is at position 0 (lowest)
315         if(nVoices >= 4) rel(*this, order[3], IRT_EQ, 0, Reify(cp3IsLowest)); //
              ↪  counterpoint_3 is at position 0 (lowest)
316
317         // For cantus firmus
318         rel(*this, cfIsLowest >> (lowest->getNotes()[measureIndex*4+position] ==
              ↪  cantusFirmus->getNotes()[measureIndex]));
319
320         // For counterpoint_1
321         if (counterpoint_1->getSpecies() == FIRST_SPECIES) {
322             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position
                  ↪  ] == counterpoint_1->getFirstNotes()[measureIndex]));
323         } else if (counterpoint_1->getSpecies() == SECOND_SPECIES) {
324             int noteIndex = (position < 2) ? 0 : 2;  // pos 0,1 -> 0; pos 2,3 ->
                  ↪  2
325             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position
                  ↪  ] == counterpoint_1->getNotes()[measureIndex*4+noteIndex]));
326         } else if (counterpoint_1->getSpecies() == THIRD_SPECIES) {
327             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position
                  ↪  ] == counterpoint_1->getNotes()[measureIndex*4+position]));
328         } else if ((counterpoint_1->getSpecies() == FOURTH_SPECIES ||
              ↪  counterpoint_1->getSpecies() == FIFTH_SPECIES) && measureIndex ==
              ↪  0) {
329             // First measure: use arsis note (index 2) for fourth/fifth species
330             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position
                  ↪  ] == counterpoint_1->getNotes()[measureIndex*4+2]));
331         } else if (counterpoint_1->getSpecies() == FOURTH_SPECIES && measureIndex
              ↪   != 0 && measureIndex != nMeasures-1) {
332             if (position == 2) {
333                 rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                      ↪  position]));
334             } else {
335                 rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == lowest->getNotes()[measureIndex*4+position
                      ↪  -1]));
336             }
337         } else if (counterpoint_1->getSpecies() == FIFTH_SPECIES && measureIndex
              ↪  != 0 && measureIndex != nMeasures-1) {
338             if (position == 2) {
339                 rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                      ↪  position]));
340             } else {
341                  IntVar currentSpecies(*this, -1, 4);
342                 rel(*this, currentSpecies, IRT_EQ, counterpoint_1->
                      ↪  getSpeciesArray()[measureIndex*4+position]);
343                 rel(*this, (cp1IsLowest && currentSpecies == THIRD_SPECIES) >> (
                      ↪  lowest->getNotes()[measureIndex*4+position] ==
                      ↪  counterpoint_1->getNotes()[measureIndex*4+position]));
344                 rel(*this, (cp1IsLowest && currentSpecies != THIRD_SPECIES) >> (
                      ↪  lowest->getNotes()[measureIndex*4+position] == lowest->
                      ↪  getNotes()[measureIndex*4+position-1]));
345             }
346         } else {
347             // Default case: use first note of the measure
348             rel(*this, cp1IsLowest >> (lowest->getNotes()[measureIndex*4+position
                  ↪  ] == counterpoint_1->getNotes()[measureIndex*4]));
349         }
350
351         if(nVoices >= 3) {
352             // For counterpoint_2
353             if (counterpoint_2->getSpecies() == FIRST_SPECIES) {
354                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getFirstNotes()[measureIndex
```

```
                ↪  ]));
355             } else if (counterpoint_2->getSpecies() == SECOND_SPECIES) {
356                 int noteIndex = (position < 2) ? 0 : 2;
357                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+
                      ↪  noteIndex]));
358             } else if (counterpoint_2->getSpecies() == THIRD_SPECIES) {
359                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+
                      ↪  position]));
360             } else if ((counterpoint_2->getSpecies() == FOURTH_SPECIES ||
                  ↪  counterpoint_2->getSpecies() == FIFTH_SPECIES) && measureIndex
                  ↪   == 0) {
361                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+2])
                      ↪  );
362             } else if (counterpoint_2->getSpecies() == FOURTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
363                 if (position == 2) {
364                     rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+
                          ↪  position] == counterpoint_2->getNotes()[measureIndex
                          ↪  *4+position]));
365                 } else {
366                     rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+
                          ↪  position] == lowest->getNotes()[measureIndex*4+
                          ↪  position-1]));
367                 }
368             } else if (counterpoint_2->getSpecies() == FIFTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
369                 if (position == 2) {
370                     rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+
                          ↪  position] == counterpoint_2->getNotes()[measureIndex
                          ↪  *4+position]));
371                 } else {
372                     IntVar currentSpecies(*this, -1, 4);
373                     rel(*this, currentSpecies, IRT_EQ, counterpoint_2->
                          ↪  getSpeciesArray()[measureIndex*4+position]);
374                     rel(*this, (cp2IsLowest && currentSpecies == THIRD_SPECIES)
                          ↪  >> (lowest->getNotes()[measureIndex*4+position] ==
                          ↪  counterpoint_2->getNotes()[measureIndex*4+position]));
375                     rel(*this, (cp2IsLowest && currentSpecies != THIRD_SPECIES)
                          ↪  >> (lowest->getNotes()[measureIndex*4+position] ==
                          ↪  lowest->getNotes()[measureIndex*4+position-1]));
376                 }
377             } else {
378                 rel(*this, cp2IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4]));
379             }
380         }
381 
382         if(nVoices >= 4) {
383             // For counterpoint_3
384             if (counterpoint_3->getSpecies() == FIRST_SPECIES) {
385                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getFirstNotes()[measureIndex
                      ↪  ]));
386             } else if (counterpoint_3->getSpecies() == SECOND_SPECIES) {
387                 int noteIndex = (position < 2) ? 0 : 2;
388                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getNotes()[measureIndex*4+
                      ↪  noteIndex]));
389             } else if (counterpoint_3->getSpecies() == THIRD_SPECIES) {
390                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getNotes()[measureIndex*4+
                      ↪  position]));
391             } else if ((counterpoint_3->getSpecies() == FOURTH_SPECIES ||
                  ↪  counterpoint_3->getSpecies() == FIFTH_SPECIES) && measureIndex
                  ↪   == 0) {
```

```
392                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getNotes()[measureIndex*4+2])
                      ↪  );
393             } else if (counterpoint_3->getSpecies() == FOURTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
394                 if (position == 2) {
395                     rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+
                          ↪  position] == counterpoint_3->getNotes()[measureIndex
                          ↪  *4+position]));
396                 } else {
397                     rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+
                          ↪  position] == lowest->getNotes()[measureIndex*4+
                          ↪  position-1]));
398                 }
399             } else if (counterpoint_3->getSpecies() == FIFTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
400                 if (position == 2) {
401                     rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+
                          ↪  position] == counterpoint_3->getNotes()[measureIndex
                          ↪  *4+position]));
402                 } else {
403                     IntVar currentSpecies(*this, -1, 4);
404                     rel(*this, currentSpecies, IRT_EQ, counterpoint_3->
                          ↪  getSpeciesArray()[measureIndex*4+position]);
405                     rel(*this, (cp3IsLowest && currentSpecies == THIRD_SPECIES)
                          ↪  >> (lowest->getNotes()[measureIndex*4+position] ==
                          ↪  counterpoint_3->getNotes()[measureIndex*4+position]));
406                     rel(*this, (cp3IsLowest && currentSpecies != THIRD_SPECIES)
                          ↪  >> (lowest->getNotes()[measureIndex*4+position] ==
                          ↪  lowest->getNotes()[measureIndex*4+position-1]));
407                 }
408             } else {
409                 rel(*this, cp3IsLowest >> (lowest->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getNotes()[measureIndex*4]));
410             }
411         }
412
413         // Set upper strata notes based on voice mapping
414         // For upper_1 (sorted position 1)
415         BoolVar cfIsSecond(*this, 0, 1);
416         BoolVar cp1IsSecond(*this, 0, 1);
417         BoolVar cp2IsSecond(*this, 0, 1);
418         BoolVar cp3IsSecond(*this, 0, 1);
419
420         rel(*this, order[0], IRT_EQ, 1, Reify(cfIsSecond));      // cantus firmus
              ↪  is at position 1 (second)
421         rel(*this, order[1], IRT_EQ, 1, Reify(cp1IsSecond));     //
              ↪  counterpoint_1 is at position 1 (second)
422         if(nVoices >= 3) rel(*this, order[2], IRT_EQ, 1, Reify(cp2IsSecond)); //
              ↪  counterpoint_2 is at position 1 (second)
423         if(nVoices >= 4) rel(*this, order[3], IRT_EQ, 1, Reify(cp3IsSecond)); //
              ↪  counterpoint_3 is at position 1 (second)
424
425         // CF (whole note)
426         rel(*this, cfIsSecond >> (upper_1->getNotes()[measureIndex*4+position] ==
              ↪  cantusFirmus->getNotes()[measureIndex]));
427
428         // cp1: mirror lowest logic
429         if (counterpoint_1->getSpecies() == FIRST_SPECIES) {
430             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+
                  ↪  position] == counterpoint_1->getFirstNotes()[measureIndex]));
431         } else if (counterpoint_1->getSpecies() == SECOND_SPECIES) {
432             int noteIndex = (position < 2) ? 0 : 2;
433             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+
                  ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                  ↪  noteIndex]));
434         } else if (counterpoint_1->getSpecies() == THIRD_SPECIES) {
```

```
435             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+
                  ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                  ↪  position]));
436         } else if ((counterpoint_1->getSpecies() == FOURTH_SPECIES ||
                  ↪  counterpoint_1->getSpecies() == FIFTH_SPECIES) && measureIndex ==
                  ↪  0) {
437             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+
                  ↪  position] == counterpoint_1->getNotes()[measureIndex*4+2]));
438         } else if (counterpoint_1->getSpecies() == FOURTH_SPECIES && measureIndex
                  ↪   != 0 && measureIndex != nMeasures-1) {
439             if (position == 2) {
440                 rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                      ↪  position]));
441             } else {
442                 rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+
                      ↪  position] == upper_1->getNotes()[measureIndex*4+position
                      ↪  -1]));
443             }
444         } else if (counterpoint_1->getSpecies() == FIFTH_SPECIES && measureIndex
                  ↪  != 0 && measureIndex != nMeasures-1) {
445             if (position == 2) {
446                 rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                      ↪  position]));
447             } else {
448                 IntVar currentSpecies(*this, -1, 4);
449                 rel(*this, currentSpecies, IRT_EQ, counterpoint_1->
                      ↪  getSpeciesArray()[measureIndex*4+position]);
450                 rel(*this, (cp1IsSecond && currentSpecies == THIRD_SPECIES) >> (
                      ↪  upper_1->getNotes()[measureIndex*4+position] ==
                      ↪  counterpoint_1->getNotes()[measureIndex*4+position]));
451                 rel(*this, (cp1IsSecond && currentSpecies != THIRD_SPECIES) >> (
                      ↪  upper_1->getNotes()[measureIndex*4+position] == upper_1->
                      ↪  getNotes()[measureIndex*4+position-1]));
452             }
453         } else {
454             rel(*this, cp1IsSecond >> (upper_1->getNotes()[measureIndex*4+
                  ↪  position] == counterpoint_1->getNotes()[measureIndex*4]));
455         }
456 
457         if(nVoices >= 3) {
458             // cp2: mirror lowest logic
459             if (counterpoint_2->getSpecies() == FIRST_SPECIES) {
460                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getFirstNotes()[measureIndex
                      ↪  ]));
461             } else if (counterpoint_2->getSpecies() == SECOND_SPECIES) {
462                 int noteIndex = (position < 2) ? 0 : 2;
463                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+
                      ↪  noteIndex]));
464             } else if (counterpoint_2->getSpecies() == THIRD_SPECIES) {
465                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+
                      ↪  position]));
466             } else if ((counterpoint_2->getSpecies() == FOURTH_SPECIES ||
                      ↪  counterpoint_2->getSpecies() == FIFTH_SPECIES) && measureIndex
                      ↪   == 0) {
467                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+2])
                      ↪  );
468             } else if (counterpoint_2->getSpecies() == FOURTH_SPECIES &&
                      ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
469                 if (position == 2) {
470                     rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex
                          ↪  *4+position] == counterpoint_2->getNotes()[
                          ↪  measureIndex*4+position]));
```

```
471                 } else {
472                     rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex
                        ↪  *4+position] == upper_1->getNotes()[measureIndex*4+
                        ↪  position-1]));
473                 }
474             } else if (counterpoint_2->getSpecies() == FIFTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
475                 if (position == 2) {
476                     rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex
                        ↪  *4+position] == counterpoint_2->getNotes()[
                        ↪  measureIndex*4+position]));
477                 } else {
478                     IntVar currentSpecies(*this, -1, 4);
479                     rel(*this, currentSpecies, IRT_EQ, counterpoint_2->
                        ↪  getSpeciesArray()[measureIndex*4+position]);
480                     rel(*this, (cp2IsSecond && currentSpecies == THIRD_SPECIES)
                        ↪  >> (upper_1->getNotes()[measureIndex*4+position] ==
                        ↪  counterpoint_2->getNotes()[measureIndex*4+position]));
481                     rel(*this, (cp2IsSecond && currentSpecies != THIRD_SPECIES)
                        ↪  >> (upper_1->getNotes()[measureIndex*4+position] ==
                        ↪  upper_1->getNotes()[measureIndex*4+position-1]));
482                 }
483             } else {
484                 rel(*this, cp2IsSecond >> (upper_1->getNotes()[measureIndex*4+
                    ↪  position] == counterpoint_2->getNotes()[measureIndex*4]));
485             }
486         }
487 
488         if(nVoices >= 4) {
489             // cp3: mirror lowest logic
490             if (counterpoint_3->getSpecies() == FIRST_SPECIES) {
491                 rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getFirstNotes()[measureIndex
                        ↪  ]));
492             } else if (counterpoint_3->getSpecies() == SECOND_SPECIES) {
493                 int noteIndex = (position < 2) ? 0 : 2;
494                 rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getNotes()[measureIndex*4+
                        ↪  noteIndex]));
495             } else if (counterpoint_3->getSpecies() == THIRD_SPECIES) {
496                 rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getNotes()[measureIndex*4+
                        ↪  position]));
497             } else if ((counterpoint_3->getSpecies() == FOURTH_SPECIES ||
                  ↪  counterpoint_3->getSpecies() == FIFTH_SPECIES) && measureIndex
                  ↪   == 0) {
498                 rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getNotes()[measureIndex*4+2])
                        ↪  );
499             } else if (counterpoint_3->getSpecies() == FOURTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
500                 if (position == 2) {
501                     rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex
                        ↪  *4+position] == counterpoint_3->getNotes()[
                        ↪  measureIndex*4+position]));
502                 } else {
503                     rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex
                        ↪  *4+position] == upper_1->getNotes()[measureIndex*4+
                        ↪  position-1]));
504                 }
505             } else if (counterpoint_3->getSpecies() == FIFTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
506                 if (position == 2) {
507                     rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex
                        ↪  *4+position] == counterpoint_3->getNotes()[
                        ↪  measureIndex*4+position]));
508                 } else {
509                     IntVar currentSpecies(*this, -1, 4);
```

```
510                     rel(*this, currentSpecies, IRT_EQ, counterpoint_3->
                          ↪  getSpeciesArray()[measureIndex*4+position]);
511                     rel(*this, (cp3IsSecond && currentSpecies == THIRD_SPECIES)
                          ↪  >> (upper_1->getNotes()[measureIndex*4+position] ==
                          ↪  counterpoint_3->getNotes()[measureIndex*4+position]));
512                     rel(*this, (cp3IsSecond && currentSpecies != THIRD_SPECIES)
                          ↪  >> (upper_1->getNotes()[measureIndex*4+position] ==
                          ↪  upper_1->getNotes()[measureIndex*4+position-1]));
513                 }
514             } else {
515                 rel(*this, cp3IsSecond >> (upper_1->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getNotes()[measureIndex*4]));
516             }
517         }
518
519         // For upper_2 (sorted position 2)
520         if(nVoices >= 3) {
521             BoolVar cfIsThird(*this, 0, 1);
522             BoolVar cp1IsThird(*this, 0, 1);
523             BoolVar cp2IsThird(*this, 0, 1);
524             BoolVar cp3IsThird(*this, 0, 1);
525             rel(*this, order[0], IRT_EQ, 2, Reify(cfIsThird));
526             rel(*this, order[1], IRT_EQ, 2, Reify(cp1IsThird));
527             rel(*this, order[2], IRT_EQ, 2, Reify(cp2IsThird));
528             if(nVoices >= 4) rel(*this, order[3], IRT_EQ, 2, Reify(cp3IsThird));
529
530             rel(*this, cfIsThird >> (upper_2->getNotes()[measureIndex*4+position]
                  ↪   == cantusFirmus->getNotes()[measureIndex]));
531
532             // cp1: mirror lowest logic
533             if (counterpoint_1->getSpecies() == FIRST_SPECIES) {
534                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getFirstNotes()[measureIndex
                      ↪  ]));
535             } else if (counterpoint_1->getSpecies() == SECOND_SPECIES) {
536                 int noteIndex = (position < 2) ? 0 : 2;
537                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                      ↪  noteIndex]));
538             } else if (counterpoint_1->getSpecies() == THIRD_SPECIES) {
539                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                      ↪  position]));
540             } else if ((counterpoint_1->getSpecies() == FOURTH_SPECIES ||
                  ↪  counterpoint_1->getSpecies() == FIFTH_SPECIES) && measureIndex
                  ↪   == 0) {
541                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+2])
                      ↪  );
542             } else if (counterpoint_1->getSpecies() == FOURTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
543                 if (position == 2) {
544                     rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+
                          ↪  position] == counterpoint_1->getNotes()[measureIndex
                          ↪  *4+position]));
545                 } else {
546                     rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+
                          ↪  position] == upper_2->getNotes()[measureIndex*4+
                          ↪  position-1]));
547                 }
548             } else if (counterpoint_1->getSpecies() == FIFTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
549                 if (position == 2) {
550                     rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+
                          ↪  position] == counterpoint_1->getNotes()[measureIndex
                          ↪  *4+position]));
551                 } else {
552                     IntVar currentSpecies(*this, -1, 4);
```

```
553                     rel(*this, currentSpecies, IRT_EQ, counterpoint_1->
                          ↪  getSpeciesArray()[measureIndex*4+position]);
554                     rel(*this, (cp1IsThird && currentSpecies == THIRD_SPECIES) >>
                          ↪   (upper_2->getNotes()[measureIndex*4+position] ==
                          ↪  counterpoint_1->getNotes()[measureIndex*4+position]));
555                     rel(*this, (cp1IsThird && currentSpecies != THIRD_SPECIES) >>
                          ↪   (upper_2->getNotes()[measureIndex*4+position] ==
                          ↪  upper_2->getNotes()[measureIndex*4+position-1]));
556                 }
557             } else {
558                 rel(*this, cp1IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4]));
559             }
560
561             // cp2: mirror lowest logic
562             if (counterpoint_2->getSpecies() == FIRST_SPECIES) {
563                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getFirstNotes()[measureIndex
                      ↪  ]));
564             } else if (counterpoint_2->getSpecies() == SECOND_SPECIES) {
565                 int noteIndex = (position < 2) ? 0 : 2;
566                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+
                      ↪  noteIndex]));
567             } else if (counterpoint_2->getSpecies() == THIRD_SPECIES) {
568                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+
                      ↪  position]));
569             } else if ((counterpoint_2->getSpecies() == FOURTH_SPECIES ||
                  ↪  counterpoint_2->getSpecies() == FIFTH_SPECIES) && measureIndex
                  ↪   == 0) {
570                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+2])
                      ↪  );
571             } else if (counterpoint_2->getSpecies() == FOURTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
572                 if (position == 2) {
573                     rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+
                          ↪  position] == counterpoint_2->getNotes()[measureIndex
                          ↪  *4+position]));
574                 } else {
575                     rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+
                          ↪  position] == upper_2->getNotes()[measureIndex*4+
                          ↪  position-1]));
576                 }
577             } else if (counterpoint_2->getSpecies() == FIFTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
578                 if (position == 2) {
579                     rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+
                          ↪  position] == counterpoint_2->getNotes()[measureIndex
                          ↪  *4+position]));
580                 } else {
581                     IntVar currentSpecies(*this, -1, 4);
582                     rel(*this, currentSpecies, IRT_EQ, counterpoint_2->
                          ↪  getSpeciesArray()[measureIndex*4+position]);
583                     rel(*this, (cp2IsThird && currentSpecies == THIRD_SPECIES) >>
                          ↪   (upper_2->getNotes()[measureIndex*4+position] ==
                          ↪  counterpoint_2->getNotes()[measureIndex*4+position]));
584                     rel(*this, (cp2IsThird && currentSpecies != THIRD_SPECIES) >>
                          ↪   (upper_2->getNotes()[measureIndex*4+position] ==
                          ↪  upper_2->getNotes()[measureIndex*4+position-1]));
585                 }
586             } else {
587                 rel(*this, cp2IsThird >> (upper_2->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4]));
588             }
589
590             if(nVoices >= 4) {
```

```
591                 // cp3 may also be third in 4 voices: mirror lowest logic
592                 if (counterpoint_3->getSpecies() == FIRST_SPECIES) {
593                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getFirstNotes()[
                        ↪  measureIndex]));
594                 } else if (counterpoint_3->getSpecies() == SECOND_SPECIES) {
595                     int noteIndex = (position < 2) ? 0 : 2;
596                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getNotes()[measureIndex
                        ↪  *4+noteIndex]));
597                 } else if (counterpoint_3->getSpecies() == THIRD_SPECIES) {
598                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getNotes()[measureIndex
                        ↪  *4+position]));
599                 } else if ((counterpoint_3->getSpecies() == FOURTH_SPECIES ||
                      ↪  counterpoint_3->getSpecies() == FIFTH_SPECIES) &&
                      ↪  measureIndex == 0) {
600                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getNotes()[measureIndex
                        ↪  *4+2]));
601                 } else if (counterpoint_3->getSpecies() == FOURTH_SPECIES &&
                      ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
602                     if (position == 2) {
603                         rel(*this, cp3IsThird >> (upper_2->getNotes()[
                            ↪  measureIndex*4+position] == counterpoint_3->
                            ↪  getNotes()[measureIndex*4+position]));
604                     } else {
605                         rel(*this, cp3IsThird >> (upper_2->getNotes()[
                            ↪  measureIndex*4+position] == upper_2->getNotes()[
                            ↪  measureIndex*4+position-1]));
606                     }
607                 } else if (counterpoint_3->getSpecies() == FIFTH_SPECIES &&
                      ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
608                     if (position == 2) {
609                         rel(*this, cp3IsThird >> (upper_2->getNotes()[
                            ↪  measureIndex*4+position] == counterpoint_3->
                            ↪  getNotes()[measureIndex*4+position]));
610                     } else {
611                         IntVar currentSpecies(*this, -1, 4);
612                         rel(*this, currentSpecies, IRT_EQ, counterpoint_3->
                            ↪  getSpeciesArray()[measureIndex*4+position]);
613                         rel(*this, (cp3IsThird && currentSpecies == THIRD_SPECIES
                            ↪  ) >> (upper_2->getNotes()[measureIndex*4+position]
                            ↪   == counterpoint_3->getNotes()[measureIndex*4+
                            ↪  position]));
614                         rel(*this, (cp3IsThird && currentSpecies != THIRD_SPECIES
                            ↪  ) >> (upper_2->getNotes()[measureIndex*4+position]
                            ↪   == upper_2->getNotes()[measureIndex*4+position
                            ↪  -1]));
615                     }
616                 } else {
617                     rel(*this, cp3IsThird >> (upper_2->getNotes()[measureIndex*4+
                        ↪  position] == counterpoint_3->getNotes()[measureIndex
                        ↪  *4]));
618                 }
619             }
620         }
621
622         // For upper_3 (sorted position 3)
623         if(nVoices >= 4) {
624             BoolVar cfIsFourth(*this, 0, 1);
625             BoolVar cp1IsFourth(*this, 0, 1);
626             BoolVar cp2IsFourth(*this, 0, 1);
627             BoolVar cp3IsFourth(*this, 0, 1);
628
629             rel(*this, order[0], IRT_EQ, 3, Reify(cfIsFourth));
630             rel(*this, order[1], IRT_EQ, 3, Reify(cp1IsFourth));
631             rel(*this, order[2], IRT_EQ, 3, Reify(cp2IsFourth));
```

```
632             rel(*this, order[3], IRT_EQ, 3, Reify(cp3IsFourth));
633
634             rel(*this, cfIsFourth >> (upper_3->getNotes()[measureIndex*4+position
                  ↪  ] == cantusFirmus->getNotes()[measureIndex]));
635
636             // cp1: mirror lowest logic
637             if (counterpoint_1->getSpecies() == FIRST_SPECIES) {
638                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getFirstNotes()[measureIndex
                      ↪  ]));
639             } else if (counterpoint_1->getSpecies() == SECOND_SPECIES) {
640                 int noteIndex = (position < 2) ? 0 : 2;
641                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                      ↪  noteIndex]));
642             } else if (counterpoint_1->getSpecies() == THIRD_SPECIES) {
643                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+
                      ↪  position]));
644             } else if ((counterpoint_1->getSpecies() == FOURTH_SPECIES ||
                  ↪  counterpoint_1->getSpecies() == FIFTH_SPECIES) && measureIndex
                  ↪   == 0) {
645                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4+2])
                      ↪  );
646             } else if (counterpoint_1->getSpecies() == FOURTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
647                 if (position == 2) {
648                     rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex
                          ↪  *4+position] == counterpoint_1->getNotes()[
                          ↪  measureIndex*4+position]));
649                 } else {
650                     rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex
                          ↪  *4+position] == upper_3->getNotes()[measureIndex*4+
                          ↪  position-1]));
651                 }
652             } else if (counterpoint_1->getSpecies() == FIFTH_SPECIES &&
                  ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
653                 if (position == 2) {
654                     rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex
                          ↪  *4+position] == counterpoint_1->getNotes()[
                          ↪  measureIndex*4+position]));
655                 } else {
656                     IntVar currentSpecies(*this, -1, 4);
657                     rel(*this, currentSpecies, IRT_EQ, counterpoint_1->
                          ↪  getSpeciesArray()[measureIndex*4+position]);
658                     rel(*this, (cp1IsFourth && currentSpecies == THIRD_SPECIES)
                          ↪  >> (upper_3->getNotes()[measureIndex*4+position] ==
                          ↪  counterpoint_1->getNotes()[measureIndex*4+position]));
659                     rel(*this, (cp1IsFourth && currentSpecies != THIRD_SPECIES)
                          ↪  >> (upper_3->getNotes()[measureIndex*4+position] ==
                          ↪  upper_3->getNotes()[measureIndex*4+position-1]));
660                 }
661             } else {
662                 rel(*this, cp1IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_1->getNotes()[measureIndex*4]));
663             }
664
665             // cp2: mirror lowest logic
666             if (counterpoint_2->getSpecies() == FIRST_SPECIES) {
667                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getFirstNotes()[measureIndex
                      ↪  ]));
668             } else if (counterpoint_2->getSpecies() == SECOND_SPECIES) {
669                 int noteIndex = (position < 2) ? 0 : 2;
670                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+
                      ↪  noteIndex]));
```

```
671             } else if (counterpoint_2->getSpecies() == THIRD_SPECIES) {
672                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+
                      ↪  position]));
673             } else if ((counterpoint_2->getSpecies() == FOURTH_SPECIES ||
                    ↪  counterpoint_2->getSpecies() == FIFTH_SPECIES) && measureIndex
                    ↪   == 0) {
674                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4+2])
                      ↪  );
675             } else if (counterpoint_2->getSpecies() == FOURTH_SPECIES &&
                    ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
676                 if (position == 2) {
677                     rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex
                          ↪  *4+position] == counterpoint_2->getNotes()[
                          ↪  measureIndex*4+position]));
678                 } else {
679                     rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex
                          ↪  *4+position] == upper_3->getNotes()[measureIndex*4+
                          ↪  position-1]));
680                 }
681             } else if (counterpoint_2->getSpecies() == FIFTH_SPECIES &&
                    ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
682                 if (position == 2) {
683                     rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex
                          ↪  *4+position] == counterpoint_2->getNotes()[
                          ↪  measureIndex*4+position]));
684                 } else {
685                     IntVar currentSpecies(*this, -1, 4);
686                     rel(*this, currentSpecies, IRT_EQ, counterpoint_2->
                          ↪  getSpeciesArray()[measureIndex*4+position]);
687                     rel(*this, (cp2IsFourth && currentSpecies == THIRD_SPECIES)
                          ↪  >> (upper_3->getNotes()[measureIndex*4+position] ==
                          ↪  counterpoint_2->getNotes()[measureIndex*4+position]));
688                     rel(*this, (cp2IsFourth && currentSpecies != THIRD_SPECIES)
                          ↪  >> (upper_3->getNotes()[measureIndex*4+position] ==
                          ↪  upper_3->getNotes()[measureIndex*4+position-1]));
689                 }
690             } else {
691                 rel(*this, cp2IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_2->getNotes()[measureIndex*4]));
692             }
693 
694             // cp3: mirror lowest logic
695             if (counterpoint_3->getSpecies() == FIRST_SPECIES) {
696                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getFirstNotes()[measureIndex
                      ↪  ]));
697             } else if (counterpoint_3->getSpecies() == SECOND_SPECIES) {
698                 int noteIndex = (position < 2) ? 0 : 2;
699                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getNotes()[measureIndex*4+
                      ↪  noteIndex]));
700             } else if (counterpoint_3->getSpecies() == THIRD_SPECIES) {
701                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getNotes()[measureIndex*4+
                      ↪  position]));
702             } else if ((counterpoint_3->getSpecies() == FOURTH_SPECIES ||
                    ↪  counterpoint_3->getSpecies() == FIFTH_SPECIES) && measureIndex
                    ↪   == 0) {
703                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+
                      ↪  position] == counterpoint_3->getNotes()[measureIndex*4+2])
                      ↪  );
704             } else if (counterpoint_3->getSpecies() == FOURTH_SPECIES &&
                    ↪  measureIndex != 0 && measureIndex != nMeasures-1) {
705                 if (position == 2) {
706                     rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex
                          ↪  *4+position] == counterpoint_3->getNotes()[
```

```
                        measureIndex*4+position]));
707                 } else {
708                     rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex
                        *4+position] == upper_3->getNotes()[measureIndex*4+
                        position-1]));
709                 }
710             } else if (counterpoint_3->getSpecies() == FIFTH_SPECIES &&
                    measureIndex != 0 && measureIndex != nMeasures-1) {
711                 if (position == 2) {
712                     rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex
                        *4+position] == counterpoint_3->getNotes()[
                        measureIndex*4+position]));
713                 } else {
714                     IntVar currentSpecies(*this, -1, 4);
715                     rel(*this, currentSpecies, IRT_EQ, counterpoint_3->
                        getSpeciesArray()[measureIndex*4+position]);
716                     rel(*this, (cp3IsFourth && currentSpecies == THIRD_SPECIES)
                        >> (upper_3->getNotes()[measureIndex*4+position] ==
                        counterpoint_3->getNotes()[measureIndex*4+position]));
717                     rel(*this, (cp3IsFourth && currentSpecies != THIRD_SPECIES)
                        >> (upper_3->getNotes()[measureIndex*4+position] ==
                        upper_3->getNotes()[measureIndex*4+position-1]));
718                 }
719             } else {
720                 rel(*this, cp3IsFourth >> (upper_3->getNotes()[measureIndex*4+
                        position] == counterpoint_3->getNotes()[measureIndex*4]));
721             }
722         }
723     }
724 }
725
726 void CounterpointProblem::setVoiceLowestFlags(int measureIndex, int nVoices, int
        size) {
727     // Set the isNotLowest flags for each voice
728     rel(*this, lowest->getFirstNotes()[measureIndex], IRT_NQ, cantusFirmus->
        getNotes()[measureIndex],
729         Reify(cantusFirmus->getIsNotLowest()[measureIndex]));
730
731     if(nVoices >= 2) {
732         // Set isNotLowest boolean for counterpoint_1
733         if(counterpoint_1->getSpecies() == FOURTH_SPECIES && measureIndex != size
                -1) {
734             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex
                    ]==1) &&
735                 (lowest->getFirstNotes()[measureIndex]==counterpoint_1->getNotes
                        ()[(measureIndex*4)+2])),
736                 IRT_NQ, 1, Reify(counterpoint_1->getIsNotLowest()[measureIndex]))
                    ;
737         } else if(counterpoint_1->getSpecies() == FIFTH_SPECIES && measureIndex
                == 0) {
738             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex
                    ]==1) &&
739                 (lowest->getFirstNotes()[measureIndex]==counterpoint_1->getNotes
                        ()[(measureIndex*4)+2])),
740                 IRT_NQ, 1, Reify(counterpoint_1->getIsNotLowest()[measureIndex]))
                    ;
741         } else {
742             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex
                    ]==1) &&
743                 (lowest->getFirstNotes()[measureIndex]==counterpoint_1->
                        getFirstNotes()[measureIndex])),
744                 IRT_NQ, 1, Reify(counterpoint_1->getIsNotLowest()[measureIndex]))
                    ;
745         }
746     }
747
748     if(nVoices == 3) {
```

```
749         rel(*this, expr(*this, counterpoint_1->getIsNotLowest()[measureIndex] !=
              ↪  cantusFirmus->getIsNotLowest()[measureIndex]),
750           IRT_EQ, counterpoint_2->getIsNotLowest()[measureIndex]);
751     }
752
753     if(nVoices == 4) {
754         // Set counterpoint_2 lowest boolean
755         if(counterpoint_2->getSpecies() == FOURTH_SPECIES && measureIndex != size
              ↪  -1) {
756             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex
                  ↪  ]==1) &&
757               (counterpoint_1->getIsNotLowest()[measureIndex]==1) &&
758               (lowest->getFirstNotes()[measureIndex]==counterpoint_2->getNotes
                    ↪  ()[(measureIndex*4)+2])),
759               IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[measureIndex])
                    ↪  ;
760         } else if(counterpoint_2->getSpecies() == FIFTH_SPECIES && measureIndex
              ↪  == 0) {
761             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex
                  ↪  ]==1) &&
762               (counterpoint_1->getIsNotLowest()[measureIndex]==1) &&
763               (lowest->getFirstNotes()[measureIndex]==counterpoint_2->getNotes
                    ↪  ()[(measureIndex*4)+2])),
764               IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[measureIndex])
                    ↪  ;
765         } else {
766             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex
                  ↪  ]==1) &&
767               (counterpoint_1->getIsNotLowest()[measureIndex]==1) &&
768               (lowest->getFirstNotes()[measureIndex]==counterpoint_2->
                    ↪  getFirstNotes()[measureIndex])),
769               IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[measureIndex])
                    ↪  ;
770         }
771
772         rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==1)
              ↪  &&
773             (counterpoint_1->getIsNotLowest()[measureIndex]==1) &&
774             (counterpoint_2->getIsNotLowest()[measureIndex]==1)),
775             IRT_NQ, counterpoint_3->getIsNotLowest()[measureIndex]);
776
777         // Set isHighest flags for 4-voice specific constraints
778         rel(*this, upper_3->getFirstNotes()[measureIndex], IRT_EQ, cantusFirmus->
              ↪  getNotes()[measureIndex],
779             Reify(cantusFirmus->getIsHighest()[measureIndex]));
780
781         // ... (similar logic for isHighest flags)
782     }
783 }
784
785 void CounterpointProblem::setMelodicIntervalConstraints(int measureIndex, int
      ↪  nVoices) {
786     // Set melodic interval constraints for the lowest stratum
787     vector<IntVarArray> corresponding_m_intervals;
788
789     corresponding_m_intervals.push_back(cantusFirmus->getMelodicIntervals());
790
791     for(int j = 0; j < nVoices-1; j++) {
792         Part* curr_cp;
793         if(j == 0) curr_cp = counterpoint_1;
794         else if(j == 1) curr_cp = counterpoint_2;
795         else curr_cp = counterpoint_3;
796
797         if(curr_cp->getSpecies() == FIRST_SPECIES) {
798             corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
                  ↪  getMelodicIntervals().slice(0, 4, curr_cp->getMelodicIntervals
                  ↪  ().size())));
799         } else if(curr_cp->getSpecies() == SECOND_SPECIES) {
```

```
800             corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
                  ↪ getMelodicIntervals().slice(2, 4, curr_cp->getMelodicIntervals
                  ↪ ().size())));
801         } else if(curr_cp->getSpecies() == THIRD_SPECIES) {
802             corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
                  ↪ getMelodicIntervals().slice(3, 4, curr_cp->getMelodicIntervals
                  ↪ ().size())));
803         } else if(curr_cp->getSpecies() == FOURTH_SPECIES) {
804             std::vector<int> selectedIndices;
805             for (int i = 4; i < curr_cp->getMelodicIntervals().size(); i += 4) {
806                 selectedIndices.push_back(i);
807             }
808             selectedIndices.push_back(curr_cp->getMelodicIntervals().size() - 2);
809 
810             IntVarArray selectedIntervals(*this, selectedIndices.size());
811             for (size_t i = 0; i < selectedIndices.size(); ++i) {
812                 selectedIntervals[i] = curr_cp->getMelodicIntervals()[
                      ↪ selectedIndices[i]];
813             }
814             corresponding_m_intervals.push_back(selectedIntervals);
815         } else if(curr_cp->getSpecies() == FIFTH_SPECIES) {
816             corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
                  ↪ getMelodicIntervals().slice(2, 4, curr_cp->getMelodicIntervals
                  ↪ ().size())));
817         }
818     }
819 
820     rel(*this, (cantusFirmus->getIsNotLowest()[measureIndex]==0) >>
821         (lowest->getMelodicIntervals()[measureIndex-1]==corresponding_m_intervals
              ↪ [0][measureIndex-1]));
822     rel(*this, (counterpoint_1->getIsNotLowest()[measureIndex]==0) >>
823         (lowest->getMelodicIntervals()[measureIndex-1]==corresponding_m_intervals
              ↪ [1][measureIndex-1]));
824     if(nVoices >= 3) {
825         rel(*this, (counterpoint_2->getIsNotLowest()[measureIndex]==0) >>
826             (lowest->getMelodicIntervals()[measureIndex-1]==
                  ↪ corresponding_m_intervals[2][measureIndex-1]));
827     }
828     if(nVoices >= 4) {
829         rel(*this, (counterpoint_3->getIsNotLowest()[measureIndex]==0) >>
830             (lowest->getMelodicIntervals()[measureIndex-1]==
                  ↪ corresponding_m_intervals[3][measureIndex-1]));
831     }
832 }
833 
834 IntVarArray CounterpointProblem::getSolutionArray(){
835     return solutionArray;
836 }
837 
838 /**
839  * Returns the size of the problem
840  * @return an integer representing the size of the vars array           RETURNS
      ↪ THE SIZE OF THE SOLUTION_ARRAY
841  */
842 int CounterpointProblem::getSize(){
843     string message = "getSize␣function␣called.␣size␣=␣" + std::to_string(
          ↪ solutionArray.size()) + "\n";
844     writeToLogFile(message.c_str());
845     return this->solutionArray.size();  /// have to use this-> ?
846 }
847 
848 /**
849  * Returns the values taken by the variables vars in a solution
850  * @todo Modify this to return the solution for your problem. This function uses
      ↪ @param size to generate an array of integers
851  * @return an array of integers representing the values of the variables in a
      ↪ solution
852  */
```

```
853 int* CounterpointProblem::return_solution(){
854     string message = "return_solution_method._Solution_:_[";
855     int* solution = new int[solutionArray.size()];
856     for(int i = 0; i < solutionArray.size(); i++){
857         solution[i] = solutionArray[i].val();
858         message += std::to_string(solution[i]) + "_";
859     }
860     message += "]\n";
861     writeToLogFile(message.c_str());
862     return solution;
863 }
864 
865 
866 int* CounterpointProblem::get_species_array_5sp(int ctp_index){
867 
868     FifthSpeciesCounterpoint* fifth_sp_ctp;
869     if (ctp_index == 0) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint*>(
        ↪  counterpoint_1);
870     else if (ctp_index == 1) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint
        ↪  *>(counterpoint_2);
871     else if (ctp_index == 2) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint
        ↪  *>(counterpoint_3);
872     else{
873         writeToLogFile("invalid_value_of_ctp_index_given_as_argument_to_
            ↪  get_species_array_5sp");
874         return nullptr;
875     }
876 
877     if (!fifth_sp_ctp){
878         cout << "type_cast_of_part_to_fifth_species_failed_" << endl;
879         writeToLogFile("type_cast_of_part_to_fifth_species_failed");
880         return nullptr;
881     }
882 
883     IntVarArray speciesArray = fifth_sp_ctp->getSpeciesArray();
884 
885     string message = "return_species_array_method._Species_array_:_[";
886     int* solution = new int[speciesArray.size()];
887     for(int i = 0; i < speciesArray.size(); i++){
888 
889         switch (speciesArray[i].val())
890         {
891         case FIRST_SPECIES:
892             solution[i] = 1;
893             break;
894 
895         case SECOND_SPECIES:
896             solution[i] = 2;
897             break;
898 
899         case THIRD_SPECIES:
900             solution[i] = 3;
901             break;
902 
903         case FOURTH_SPECIES:
904             solution[i] = 4;
905             break;
906 
907         case -1:
908             solution[i] = 0;
909             break;
910 
911         default:
912             cout << "invalid_value_in_species_array" << endl;
913             writeToLogFile("invalid_value_in_species_array");
914             return nullptr;
915         }
916         message += std::to_string(solution[i]) + "_";
```

```
917     }
918     message += "]\n";
919     writeToLogFile(message.c_str());
920     return solution;
921 }
922 
923 
924 int* CounterpointProblem::get_extended_cp_domain(int ctp_index){
925     vector<int> ext_cp_dom;
926     if(ctp_index == 0) ext_cp_dom = counterpoint_1->getExtendedDomain();
927     else if(ctp_index == 1) ext_cp_dom = counterpoint_2->getExtendedDomain();
928     else if(ctp_index == 2) ext_cp_dom = counterpoint_3->getExtendedDomain();
929     else{
930         writeToLogFile("invalid value of ctp_index given as argument to
            ↪ get_extended_cp_domain");
931         return nullptr;
932     }
933     writeToLogFile(int_vector_to_string(ext_cp_dom).c_str());
934 
935     int* ext_cp_dom_int_ptr = new int[ext_cp_dom.size()];
936     for(int i = 0; i < ext_cp_dom.size(); i++){
937         ext_cp_dom_int_ptr[i] = ext_cp_dom[i];
938         // message += std::to_string(solution[i]) + " ";
939     }
940 
941     return ext_cp_dom_int_ptr;
942 }
943 
944 int CounterpointProblem::get_ext_cp_domain_size(int ctp_index){
945     int cpDomSize;
946     if(ctp_index == 0) cpDomSize = counterpoint_1->getExtendedDomain().size();
947     else if(ctp_index == 1) cpDomSize = counterpoint_2->getExtendedDomain().size
        ↪ ();
948     else if(ctp_index == 2) cpDomSize = counterpoint_3->getExtendedDomain().size
        ↪ ();
949     else{
950         writeToLogFile("invalid value of ctp_index given as argument to
            ↪ get_extended_cp_domain");
951         return -1;
952     }
953 
954     string message = "getextcpdomainsize function called. size = " + std::
        ↪ to_string(cpDomSize) + "\n";
955     writeToLogFile(message.c_str());
956     return cpDomSize;
957 }
958 
959 Stratum* CounterpointProblem::getLowest(){
960     return lowest;
961 }
962 
963 Part* CounterpointProblem::getCantusFirmus(){
964     return cantusFirmus;
965 }
966 
967 Part* CounterpointProblem::getCounterpoint_1(){
968     return counterpoint_1;
969 }
970 
971 Part* CounterpointProblem::getCounterpoint_2(){
972     return counterpoint_2;
973 }
974 
975 Part* CounterpointProblem::getCounterpoint_3(){
976     return counterpoint_3;
977 }
978 
979 
```

```
/************************
 * Search engine methods *
 ************************/

//Gecode::Search::TimeStop global_timeout(50000);

/**
 * Creates a search engine for the given problem
 * Should only be used when using OM, otherwise you can create the solver etc in the main file
 * @todo Modify this function to add search options etc
 * @param pb an instance of the Problem class representing a given problem
 * @param type the type of search engine to create (see enumeration in headers/gecode_problem.hpp)
 * @return a search engine for the given problem
 */
Search::Base<CounterpointProblem>* make_solver(CounterpointProblem* pb, int type) {
    string message = "make_solver function called. type of solver :\n" + to_string(type) + "\n";
    writeToLogFile(message.c_str());

    Gecode::Search::Options opts;
    /**@todo add here any options you want*/
    //opts.stop = &global_timeout;
    opts.threads = 1;

    if (type == bab_solver)
        return new BAB<CounterpointProblem>(pb, opts);
    else // default case
        return new DFS<CounterpointProblem>(pb, opts);
}

/**
 * Returns the next solution space for the problem
 * Should only be used when using OM
 * @param solver a solver for the problem
 * @return an instance of the Problem class representing the next solution to the problem
 */
CounterpointProblem* get_next_solution_space(Search::Base<CounterpointProblem>* solver){
    string message = "get_next_solution_space function called.\n";
    // RESET TIMEOUT OBJECT HERE
    //global_timeout.reset();
    CounterpointProblem* sol_space = solver->next();
    if (sol_space == nullptr){
        message += "solution_space was null. \n";
        writeToLogFile(message.c_str());
        return NULL;
    }
    message += sol_space->to_string();
    writeToLogFile(message.c_str());
    return sol_space;
}

void CounterpointProblem::computeCombinedCosts(){
    int sz = 2;
    if(counterpoint_2!=nullptr){
        sz++;
    }
    if(counterpoint_3!=nullptr){
        sz++;
    }
    for (size_t i = 0; i < combinedCostNames.size(); i++) {
        string costName = combinedCostNames[i];
        IntVarArgs to_combined(sz);
        // get the cost of the cantus firmus
```

```
        vector<string> cf_vec = cantusFirmus->getToCombineCostNames();
        auto cf_it = std::find(cf_vec.begin(), cf_vec.end(), costName);
        if (cf_it != cf_vec.end()) {
            int index = std::distance(cf_vec.begin(), cf_it);
            to_combined[0] = cantusFirmus->getToCombineCosts()[index];
        } else {
            to_combined[0] = IntVar(*this, 0, 0);
        }
        // get the cost of the counterpoint 1
        vector<string> cp1_vec = counterpoint_1->getToCombineCostNames();
        auto cp1_it = std::find(cp1_vec.begin(), cp1_vec.end(), costName);
        if (cp1_it != cp1_vec.end()) {
            int index = std::distance(cp1_vec.begin(), cp1_it);
            to_combined[1] = counterpoint_1->getToCombineCosts()[index];
        } else {
            to_combined[1] = IntVar(*this, 0, 0);
        }
        // get the cost of the counterpoint 2
        if(counterpoint_2!=nullptr){
            vector<string> cp2_vec = counterpoint_2->getToCombineCostNames();
            auto cp2_it = std::find(cp2_vec.begin(), cp2_vec.end(), costName);
            if (cp2_it != cp2_vec.end()) {
                int index = std::distance(cp2_vec.begin(), cp2_it);
                to_combined[2] = counterpoint_2->getToCombineCosts()[index];
            } else {
                to_combined[2] = IntVar(*this, 0, 0);
            }
        }
        // get the cost of the counterpoint 3
        if(counterpoint_3!=nullptr){
            vector<string> cp3_vec = counterpoint_3->getToCombineCostNames();
            auto cp3_it = std::find(cp3_vec.begin(), cp3_vec.end(), costName);
            if (cp3_it != cp3_vec.end()) {
                int index = std::distance(cp3_vec.begin(), cp3_it);
                to_combined[3] = counterpoint_3->getToCombineCosts()[index];
            } else {
                to_combined[3] = IntVar(*this, 0, 0);
            }
        }
        // sum the costs
        rel(*this, combinedCosts[i], IRT_EQ, expr(*this, sum(to_combined)));
    }
}
```

FourVoiceCounterpoint.cpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../../headers/CounterpointProblems/FourVoiceCounterpoint.hpp"
#include "../../headers/CounterpointUtils.hpp"

FourVoiceCounterpoint::FourVoiceCounterpoint(vector<int> cf, vector<Species> sp,
  ↪  vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
        vector<int> s_costs, vector<int> imp, int bm):
    CounterpointProblem(cf, -1, m_costs, g_costs, s_costs, imp, FOUR_VOICES)
{
    species = sp;

    //initialize upper strata

    upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(),
      ↪  THREE_VOICES, FOUR_VOICES);
```

```
17      upper_2 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(),
          ↪ THREE_VOICES, FOUR_VOICES);
18      upper_3 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(),
          ↪ THREE_VOICES, FOUR_VOICES);
19
20      //create counterpoints
21
22      counterpoint_1 = create_counterpoint(*this, species[0], nMeasures, cf, (6 *
          ↪ v_type[0] - 12) + cf[0], (6 * v_type[0] + 12) + cf[0], lowest,
23          cantusFirmus, v_type[0], m_costs, g_costs, s_costs, bm, FOUR_VOICES);
24      counterpoint_2 = create_counterpoint(*this, species[1], nMeasures, cf, (6 *
          ↪ v_type[1] - 12) + cf[0], (6 * v_type[1] + 12) + cf[0], lowest,
25          cantusFirmus, v_type[1], m_costs, g_costs, s_costs, bm, FOUR_VOICES);
26      counterpoint_3 = create_counterpoint(*this, species[2], nMeasures, cf, (6 *
          ↪ v_type[2] - 12) + cf[0], (6 * v_type[2] + 12) + cf[0], lowest,
27          cantusFirmus, v_type[2], m_costs, g_costs, s_costs, bm, FOUR_VOICES);
28
29      //create strata
30
31      setStrata();
32
33      //creating variables
34
35      vector<Part*> parts = {cantusFirmus, counterpoint_1, counterpoint_2,
          ↪ counterpoint_3};
36      int scc_cz = 3*((cantusFirmus->getSize()/4)-1);
37      bool containsThirdSpecies = 0;
38
39      triadCostArray = IntVarArray(*this, counterpoint_1->getFirstHInterval().size
          ↪ (), IntSet({0, not_harmonic_triad_cost, double_fifths_cost,
          ↪ double_thirds_cost,
40          triad_with_octave_cost}));
41      successiveCostArray = IntVarArray(*this, scc_cz, IntSet({0, counterpoint_1->
          ↪ getSuccCost()}));
42
43      // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
44      if (activeConstraints[V4_G6]) {
45          for(int p = 1; p < parts.size(); p++){
46              G6_noChromaticMelodies(*this, parts[p], sp[p-1]);
47          }
48      }
49
50      // 1.H4 (G9) last chord must have the same fundamental as the cf (used
          ↪ throughout the composition)
51      if (activeConstraints[V4_1H4]) {
52          G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
53      }
54
55      //H8 : harmonic triads are preferred, adapted for 4 voices
56      if (activeConstraints[V4_1H8]) {
57          H8_4v_preferHarmonicTriad(*this, triadCostArray, upper_1, upper_2,
              ↪ upper_3);
58      }
59
60      //M4 variety cost (notes should be as diverse as possible)
61      if (activeConstraints[V4_1M4]) {
62          M2_1_varietyCost(*this, parts);
63      }
64
65      //P4 avoid successive perfect consonances
66      if (activeConstraints[V4_1P4]) {
67          P4_successiveCost(*this, parts, scc_cz, successiveCostArray, species);
68      }
69
70
71      //P6 : no move in same direction
72      if (activeConstraints[V4_1P6]) {
73          P6_4v_noMoveInSameDirection(*this, parts);
```

```
74          // if(counterpoint_1->getSpecies()!=FOURTH_SPECIES&&counterpoint_2->
              ↪ getSpecies()!=FOURTH_SPECIES&&counterpoint_3->getSpecies()!=
              ↪ FOURTH_SPECIES){
75          //     P6_4v_noMoveInSameDirection(*this, parts);
76          // }
77      }
78
79      //P7 : no suxxessive ascending sixths
80      if (activeConstraints[V4_1P7]) {
81          P7_noSuccessiveAscendingSixths(*this, parts);
82      }
83
84      //2.M2, have to write it here since it has a weird interaction with the third
          ↪  species
85      if (activeConstraints[V4_2M2]) {
86          M2_2_3v_melodicIntervalsNotExceedMinorSixth(*this, parts,
              ↪ containsThirdSpecies);
87      }
88
89      //two fifth species counterpoints should be as different as possible
90      if (activeConstraints[V4_5R9]) {
91          R9_5_twoFifthSpeciesDiversity_3v(*this, counterpoint_1, counterpoint_3);
92      }
93
94      //no minor second interval between upper
95      if (activeConstraints[V4_U2]) {
96          noMinorSecondBetweenUpper(*this, vector<Stratum*>{upper_1, upper_2,
              ↪ upper_3});
97      }
98
99      solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size()
          ↪  + counterpoint_2->getBranchingNotes().size() +
100         counterpoint_3->getBranchingNotes().size(), 0, 127);
101
102     unitedCosts = IntVarArray(*this, 14, 0, 10000000);
103     unitedCostNames = {};
104
105     uniteCounterpoints();
106     uniteCosts();
107
108     // compute combined costs
109     computeCombinedCosts();
110
111     orderCosts();
112
113     //Branching strategies
114
115     branch(*this, lowest->getNotes().slice(0, 4/notesPerMeasure.at(FIRST_SPECIES)
          ↪ , lowest->getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN
          ↪ ());
116
117     if(species[0]==FIFTH_SPECIES){
118         branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(),
              ↪ INT_VAL_RND(3U));
119     }
120     if(species[1]==FIFTH_SPECIES){
121         branch(*this, counterpoint_2->getSpeciesArray(), INT_VAR_DEGREE_MAX(),
              ↪ INT_VAL_RND(3U));
122     }
123     if(species[2]==FIFTH_SPECIES){
124         branch(*this, counterpoint_3->getSpeciesArray(), INT_VAR_DEGREE_MAX(),
              ↪ INT_VAL_RND(3U));
125     }
126
127     if(species[0]==FIFTH_SPECIES){
128         branch(*this, counterpoint_1->getCambiataCostArray(), INT_VAR_DEGREE_MAX
              ↪ (), INT_VAL_SPLIT_MIN());
129     }
```

```
130     if(species[1]==FIFTH_SPECIES){
131         branch(*this, counterpoint_2->getCambiataCostArray(), INT_VAR_DEGREE_MAX
              ↪ (), INT_VAL_SPLIT_MIN());
132     }
133     if(species[2]==FIFTH_SPECIES){
134         branch(*this, counterpoint_3->getCambiataCostArray(), INT_VAR_DEGREE_MAX
              ↪ (), INT_VAL_SPLIT_MIN());
135     }
136 
137     if(species[0]==FIFTH_SPECIES){
138         branch(*this, counterpoint_1->getSyncopeCostArray(), INT_VAR_DEGREE_MAX()
              ↪ , INT_VAL_SPLIT_MIN());
139     }
140     if(species[1]==FIFTH_SPECIES){
141         branch(*this, counterpoint_2->getSyncopeCostArray(), INT_VAR_DEGREE_MAX()
              ↪ , INT_VAL_SPLIT_MIN());
142     }
143     if(species[2]==FIFTH_SPECIES){
144         branch(*this, counterpoint_3->getSyncopeCostArray(), INT_VAR_DEGREE_MAX()
              ↪ , INT_VAL_SPLIT_MIN());
145     }
146 
147 
148     if(species[0]==FOURTH_SPECIES){
149         branch(*this, counterpoint_1->getSyncopeCostArray(),  INT_VAR_DEGREE_MAX
              ↪ (), INT_VAL_MIN());
150     }
151     if(species[1]==FOURTH_SPECIES){
152         branch(*this, counterpoint_2->getSyncopeCostArray(),  INT_VAR_DEGREE_MAX
              ↪ (), INT_VAL_MIN());
153     }
154     if(species[2]==FOURTH_SPECIES){
155         branch(*this, counterpoint_3->getSyncopeCostArray(),  INT_VAR_DEGREE_MAX
              ↪ (), INT_VAL_MIN());
156     }
157 
158     branch(*this, solutionArray, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
159     // cout << "HERE" << endl;
160 }
161 
162 // COPY CONSTRUCTOR
163 FourVoiceCounterpoint::FourVoiceCounterpoint(FourVoiceCounterpoint& s) :
      ↪ CounterpointProblem(s){
164     species = s.species;
165 }
166 
167 IntLexMinimizeSpace* FourVoiceCounterpoint::copy(){
168     return new FourVoiceCounterpoint(*this);
169 }
170 
171 string FourVoiceCounterpoint::to_string() const {
172     string text = CounterpointProblem::to_string();
173     text += "Counterpoint␣1␣:␣\n";
174     text += counterpoint_1->to_string();
175     text += "\n";
176     text += "Counterpoint␣2␣:␣\n";
177     text += counterpoint_2->to_string();
178     text += "\n";
179     text += "Counterpoint␣3␣:␣\n";
180     text += counterpoint_3->to_string();
181     text += "\n";
182     text += "␣Solution␣array␣:␣\n";
183     text += intVarArray_to_string(solutionArray);
184     text += "\n";
185     return text;
186 }
187 
188 void FourVoiceCounterpoint::uniteCounterpoints(){
```

```
189     int idx = 0;
190     for(int i = 0; i < counterpoint_1->getBranchingNotes().size(); i++){
191         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_1->getBranchingNotes
              ↪ ()[i]);
192         idx++;
193     }
194     for(int i = 0; i < counterpoint_2->getBranchingNotes().size(); i++){
195         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_2->getBranchingNotes
              ↪ ()[i]);
196         idx++;
197     }
198     for(int i = 0; i < counterpoint_3->getBranchingNotes().size(); i++){
199         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_3->getBranchingNotes
              ↪ ()[i]);
200         idx++;
201     }
202 }
203
204 void FourVoiceCounterpoint::uniteCosts(){
205     int cp1_idx = 0;
206     int cp2_idx = 0;
207     int cp3_idx = 0;
208     for(int i = 0; i < 14; i++){
209         string name = importanceNames[i];
210         if(name=="succ"){
211             unitedCostNames.push_back(name);
212             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(
                  ↪ successiveCostArray))));
213         } else if(name=="triad"){
214             unitedCostNames.push_back(name);
215             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(
                  ↪ triadCostArray))));
216         } else {
217
218             //decides if the cost is present for any counterpoint
219
220             bool cp1_contains = 0;
221             bool cp2_contains = 0;
222             bool cp3_contains = 0;
223             int sz = 0;
224             //determining in how many counterpoints a cost appears
225             for(int t = 0; t < counterpoint_1->getCostNames().size(); t++){
226                 if(name==counterpoint_1->getCostNames()[t]){
227                     cp1_contains=1;
228                     sz++;
229                 }
230             }
231             for(int t = 0; t < counterpoint_2->getCostNames().size(); t++){
232                 if(name==counterpoint_2->getCostNames()[t]){
233                     cp2_contains=1;
234                     sz++;
235                 }
236             }
237             for(int t = 0; t < counterpoint_3->getCostNames().size(); t++){
238                 if(name==counterpoint_3->getCostNames()[t]){
239                     cp3_contains=1;
240                     sz++;
241                 }
242             }
243             //if it is not present in any counterpoint -> leave it empty
244             if(!cp1_contains && !cp2_contains && !cp3_contains){
245                 unitedCostNames.push_back("NOT_ADDED");
246             } else { //else -> add the costs together for that entry
247                 unitedCostNames.push_back(name);
248                 //adds the cost to the IntVarArgs
249                 IntVarArgs x(sz);
250                 int idx=0;
251                 if(cp1_contains){
```

```
                    x[idx] = counterpoint_1->getCosts()[cp1_idx];
                    idx++;
                    cp1_idx++;
                }
                if(cp2_contains){
                    x[idx] = counterpoint_2->getCosts()[cp2_idx];
                    idx++;
                    cp2_idx++;
                }
                if(cp3_contains){
                    x[idx] = counterpoint_3->getCosts()[cp3_idx];
                    idx++;
                    cp3_idx++;
                }
                rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(x)));
            }
        }
    }
}
```

ThreeVoiceCounterpoint.cpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../../headers/CounterpointProblems/ThreeVoiceCounterpoint.hpp"
#include "../../headers/CounterpointUtils.hpp"

/**
 * Constructor of the class.
 * @param cf a vector<int> representing the cantus firmus.
 * @param sp the species of the counterpoint. it takes values from the enum "species" in headers/Utilities.hpp
 * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
 * @param lb the lowest note possible for the counterpoint in MIDI
 * @param ub the highest note possible for the counterpoint in MIDI
 */
ThreeVoiceCounterpoint::ThreeVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
    vector<int> s_costs, vector<int> imp, int bm) :
    CounterpointProblem(cf, -1, m_costs, g_costs, s_costs, imp, THREE_VOICES){
    species = sp;

    //initialize upper strata
    upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES);
    upper_2 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES);
    upper_3 = nullptr;

    //create counterpoints

    counterpoint_1 = create_counterpoint(*this, species[0], nMeasures, cf, (6 * v_type[0] - 12) + cf[0], (6 * v_type[0] + 12) + cf[0], lowest,
        cantusFirmus, v_type[0], m_costs, g_costs, s_costs, bm, THREE_VOICES);
    counterpoint_2 = create_counterpoint(*this, species[1], nMeasures, cf, (6 * v_type[1] - 12) + cf[0], (6 * v_type[1] + 12) + cf[0], lowest,
        cantusFirmus, v_type[1], m_costs, g_costs, s_costs, bm, THREE_VOICES);
    counterpoint_3 = nullptr;

    //create strata
    setStrata();

```

```
37      //creating variables
38
39      vector<Part*> parts = {cantusFirmus, counterpoint_1, counterpoint_2};
40      int scc_cz = ((cantusFirmus->getSize()/4)-1);
41      bool containsThirdSpecies = 0;
42
43      triadCostArray = IntVarArray(*this, counterpoint_1->getFirstHInterval().size
          ↪ (), IntSet({0, counterpoint_1->getTriadCost()}));
44      successiveCostArray = IntVarArray(*this, scc_cz, IntSet({0, counterpoint_1->
          ↪ getSuccCost()}));
45
46      // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
47      if (activeConstraints[V3_G6]) {
48          for(int p = 1; p < parts.size(); p++){
49              G6_noChromaticMelodies(*this, parts[p], sp[p-1]);
50          }
51      }
52
53      // 1.H4 (G9) last chord must have the same fundamental as the cf (used
          ↪ throughout the composition)
54      if (activeConstraints[V3_1H4]) {
55          G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
56      }
57
58      //H5 for three voices
59      if (activeConstraints[V3_1H5]) {
60          H5_1_differentNotes(*this, parts); //this function modified by Tom Lai
61      }
62
63      //H13
64      if (activeConstraints[V4_U2]) {
65          noMinorSecondBetweenUpper(*this, vector<Stratum*>{upper_1, upper_2});
66      }
67
68      //H8 : the triad should be used as much as possible
69      if (activeConstraints[V3_1H8]) {
70          H8_3v_preferHarmonicTriad(*this, counterpoint_1, triadCostArray, upper_1,
              ↪  upper_2);
71      }
72
73      //M4 variety cost (notes should be as diverse as possible)
74      if (activeConstraints[V3_1M4]) {
75          M2_1_varietyCost(*this, parts);
76      }
77
78      //P4 avoid successive perfect consonances
79      if (activeConstraints[V3_1P4]) {
80          P4_successiveCost(*this, parts, scc_cz, successiveCostArray, species);
81      }
82
83      //P6 : no move in same direction
84      if (activeConstraints[V3_1P6]) {
85          P6_3v_noMoveInSameDirection(*this, parts);
86          // if(counterpoint_1->getSpecies()!=FOURTH_SPECIES&&counterpoint_2->
              ↪ getSpecies()!=FOURTH_SPECIES){ //doesn't apply to 4th species
87              // P6_3v_noMoveInSameDirection(*this, parts);
88          // }
89      }
90
91      //P7 : no suxxessive ascending sixths
92      if (activeConstraints[V3_1P7]) {
93          P7_noSuccessiveAscendingSixths(*this, parts);
94      }
95
96      //2.M2, have to write it here since it has a weird interaction with the third
          ↪  species
97      if (activeConstraints[V3_2M2]) {
```

```
98          M2_2_3v_melodicIntervalsNotExceedMinorSixth(*this, parts,
              ↪  containsThirdSpecies);
99      }
100
101     // 1.H13 no minor second interval between upper
102     if (activeConstraints[V4_U2]) {
103         noMinorSecondBetweenUpper(*this, vector<Stratum*>{upper_1, upper_2});
104     }
105
106     //5.R9 two  fifth species counterpoints should be as different as possible
107     if (activeConstraints[V3_5R9]) {
108         R9_5_twoFifthSpeciesDiversity_3v(*this, counterpoint_1, counterpoint_2);
109     }
110
111     solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size()
          ↪   + counterpoint_2->getBranchingNotes().size(), 0, 127);
112
113     unitedCosts = IntVarArray(*this, 14, 0, 10000000);
114     unitedCostNames = {};
115
116     uniteCounterpoints();
117     uniteCosts();
118
119     // for(int i = 0; i < unitedCostNames.size(); i++){
120     //     cout << unitedCostNames[i] << endl;
121     // }
122
123     // compute combined costs
124     computeCombinedCosts();
125
126
127     orderCosts();
128
129     //Branching strategies
130
131     branch(*this, lowest->getNotes().slice(0, 4/notesPerMeasure.at(FIRST_SPECIES)
          ↪  , lowest->getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN
          ↪  ());
132
133     if(species[0]==FIFTH_SPECIES){
134         branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(),
              ↪  INT_VAL_RND(3U));
135     }
136     if(species[1]==FIFTH_SPECIES){
137         branch(*this, counterpoint_2->getSpeciesArray(), INT_VAR_DEGREE_MAX(),
              ↪  INT_VAL_RND(3U));
138     }
139
140     if(species[0]==FIFTH_SPECIES){
141         branch(*this, counterpoint_1->getCambiataCostArray(), INT_VAR_DEGREE_MAX
              ↪  (), INT_VAL_SPLIT_MIN());
142     }
143     if(species[1]==FIFTH_SPECIES){
144         branch(*this, counterpoint_2->getCambiataCostArray(), INT_VAR_DEGREE_MAX
              ↪  (), INT_VAL_SPLIT_MIN());
145     }
146
147     if(species[0]==FOURTH_SPECIES || species[0]==FIFTH_SPECIES){
148         branch(*this, counterpoint_1->getSyncopeCostArray(),  INT_VAR_DEGREE_MAX
              ↪  (), INT_VAL_MIN());
149     }
150     if(species[1]==FOURTH_SPECIES || species[1]==FIFTH_SPECIES){
151         branch(*this, counterpoint_2->getSyncopeCostArray(),  INT_VAR_DEGREE_MAX
              ↪  (), INT_VAL_MIN());
152     }
153
154     branch(*this, solutionArray, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
155
```

```
156     writeToLogFile(("solution␣array␣size␣:␣" + std::to_string(solutionArray.size
          ↪ ())).c_str());
157
158 }
159
160 // COPY CONSTRUCTOR
161 ThreeVoiceCounterpoint::ThreeVoiceCounterpoint(ThreeVoiceCounterpoint& s) :
      ↪ CounterpointProblem(s){
162     species = s.species;
163 }
164
165 IntLexMinimizeSpace* ThreeVoiceCounterpoint::copy(){
166     return new ThreeVoiceCounterpoint(*this);
167 }
168
169
170 string ThreeVoiceCounterpoint::to_string() const {
171     string text = CounterpointProblem::to_string();
172     text += "Counterpoint␣1␣:␣\n";
173     text += counterpoint_1->to_string();
174     text += "\n";
175     text += "Counterpoint␣2␣:␣\n";
176     text += counterpoint_2->to_string();
177     text += "\n";
178     text += "Successive␣cost␣array␣:␣\n";
179     text += intVarArray_to_string(successiveCostArray);
180     text += "\n";
181     text += "␣Solution␣array␣:␣\n";
182     text += intVarArray_to_string(solutionArray);
183     text += "\n";
184     return text;
185 }
186
187 void ThreeVoiceCounterpoint::uniteCounterpoints(){
188     //this function takes all the notes that are being branched on and adds them
          ↪ one after the other to the solution array
189     int idx = 0;
190     for(int i = 0; i < counterpoint_1->getBranchingNotes().size(); i++){
191         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_1->getBranchingNotes
              ↪ ()[i]);
192         idx++;
193     }
194     for(int i = 0; i < counterpoint_2->getBranchingNotes().size(); i++){
195         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_2->getBranchingNotes
              ↪ ()[i]);
196         idx++;
197     }
198 }
199
200 void ThreeVoiceCounterpoint::uniteCosts(){
201     //this function takes the costs that are present for each species and some 3v
          ↪  specific costs and adds them together
202     int cp1_idx = 0;
203     int cp2_idx = 0;
204     for(int i = 0; i < 14; i++){
205         //goes through every possible cost
206         string name = importanceNames[i];
207         //check seperately for the successive cost
208         if(name=="succ"){
209             unitedCostNames.push_back(name);
210             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(
                  ↪ successiveCostArray))));
211         } else if(name=="triad"){ //check seperately for the triad cost
212             unitedCostNames.push_back(name);
213             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(
                  ↪ triadCostArray))));
214         } else {
215
```

```
            //decides if the cost is present for any counterpoint

            bool cp1_contains = 0;
            bool cp2_contains = 0;
            int sz = 0;
            for(int t = 0; t < counterpoint_1->getCostNames().size(); t++){
                if(name==counterpoint_1->getCostNames()[t]){
                    cp1_contains=1;
                    sz++;
                }
            }
            for(int t = 0; t < counterpoint_2->getCostNames().size(); t++){
                if(name==counterpoint_2->getCostNames()[t]){
                    cp2_contains=1;
                    sz++;
                }
            }
            //if the cost is present in no counterpoint -> add a specific name to the array containing all present cost names
            if(!cp1_contains && !cp2_contains){
                unitedCostNames.push_back("NOT_ADDED");
            } else {
                //if it is in either counterpoint -> add the cost
                unitedCostNames.push_back(name);
                //adds the cost to the IntVarArgs
                IntVarArgs x(sz);
                int idx=0;
                if(cp1_contains){
                    x[idx] = counterpoint_1->getCosts()[cp1_idx];
                    idx++;
                    cp1_idx++;
                }
                if(cp2_contains){
                    x[idx] = counterpoint_2->getCosts()[cp2_idx];
                    idx++;
                    cp2_idx++;
                }
                //sum the cost together if it are present in both counterpoints
                rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(x)));
            }
        }
    }
}
```

TwoVoiceCounterpoint.cpp

```
    //
// Created by Damien Sprockeels on 13/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#include "../../headers/CounterpointProblems/TwoVoiceCounterpoint.hpp"
#include "../../headers/CounterpointUtils.hpp"

/**
 * Constructor of the class.
 * @param cf a vector<int> representing the cantus firmus.
 * @param sp the species of the counterpoint. it takes values from the enum "species" in headers/Utilities.hpp
 * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
 * @param lb the lowest note possible for the counterpoint in MIDI
 * @param ub the highest note possible for the counterpoint in MIDI
 */
```

```
17 TwoVoiceCounterpoint::TwoVoiceCounterpoint(vector<int> cf, Species sp, int v_type
     ↪ , vector<int> m_costs, vector<int> g_costs,
18     vector<int> s_costs, vector<int> imp, int bm) :
19     CounterpointProblem(cf, v_type, m_costs, g_costs, s_costs, imp, TWO_VOICES){
20     species = sp;
21     upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes());
22     upper_2 = nullptr;
23     upper_3 = nullptr;
24
25     counterpoint_1 = create_counterpoint(*this, species, nMeasures, cf, (6 *
         ↪ v_type - 12) + cf[0], (6 * v_type + 12) + cf[0], lowest, cantusFirmus,
26         v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
27     counterpoint_2 = nullptr;
28     counterpoint_3 = nullptr;
29
30     // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
31     if (activeConstraints[V2_G6]) {
32         G6_noChromaticMelodies(*this, counterpoint_1, species);
33     }
34
35     // 1.H4 (G9)
36     if (activeConstraints[V2_G9]) {
37         G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
38     }
39
40     /// H2 from Thibault: The first harmonic interval must be a perfect
         ↪ consonance
41     if (activeConstraints[V2_1H2]) {
42         //we check for fifth species since it always starts with a break, so it
             ↪ doesn't make sense to apply the constraint in this case
43         if(species!=FIFTH_SPECIES){
44             H2_1_startWithPerfectConsonance(*this, counterpoint_1);
45         }
46     }
47
48     /// H3 from Thibault: The last harmonic interval must be a perfect consonance
49     if (activeConstraints[V2_1H3]) {
50         H3_1_endWithPerfectConsonance(*this, counterpoint_1);
51     }
52
53     // H5 from Thibault : The cp and the cf cannot play the same note
54     if (activeConstraints[V2_1H5]) {
55         H5_1_cpAndCfDifferentNotes(*this, counterpoint_1, cantusFirmus);
56     }
57
58     //3.H4 : in the penultimate measure, if the cantusFirmus is in the upper part
         ↪ , then the h_interval of the first note should be a minor third
59     if (activeConstraints[SP3_3H4_2V]) {
60         // Create Boolean variables for each condition
61         BoolVar isThirdSpecies(*this, 0, 1);
62         BoolVar isNotLowestCantus(*this, 0, 1);
63
64         rel(*this, isThirdSpecies == (counterpoint_1->getSpecies() ==
             ↪ THIRD_SPECIES));
65         rel(*this, isNotLowestCantus == (cantusFirmus->getIsNotLowest()[
             ↪ cantusFirmus->getIsNotLowest().size()-2] == 1));
66
67         // Combine the conditions using Gecode's AND operator
68         BoolVar is3H4active(*this, 0, 1);
69         rel(*this, is3H4active == (isThirdSpecies && isNotLowestCantus));
70
71         // Implication constraint
72         rel(*this, is3H4active >>
73             (expr(*this, abs(cantusFirmus->getFirstHInterval()[cantusFirmus->
                 ↪ getFirstHInterval().size()-2])) == MINOR_THIRD));
74     }
75
76     setStrata();
```

```
77
78      unitedCosts = IntVarArray(*this, counterpoint_1->getCosts().size(), 0,
          ↪ 1000000);
79
80      for(int i = 0; i < unitedCosts.size(); i++){
81          rel(*this, unitedCosts[i], IRT_EQ, counterpoint_1->getCosts()[i]);
82      }
83
84      unitedCostNames = counterpoint_1->getCostNames();
85
86      // compute combined costs
87      computeCombinedCosts();
88
89      orderCosts();
90
91      solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size()
          ↪ , 0, 127);
92      rel(*this, solutionArray, IRT_EQ, counterpoint_1->getBranchingNotes());
93
94      branch(*this, lowest->getNotes().slice(0, 4/notesPerMeasure.at(FIRST_SPECIES)
          ↪ , lowest->getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN
          ↪ ());
95      if(species==FIFTH_SPECIES){
96          branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(),
              ↪ INT_VAL_RND(3U));
97      }
98      if(species==FIFTH_SPECIES){
99          branch(*this, counterpoint_1->getCambiataCostArray(),  INT_VAR_DEGREE_MAX
              ↪ (), INT_VAL_MIN());
100     }
101     if(species==FOURTH_SPECIES || species==FIFTH_SPECIES){
102         branch(*this, counterpoint_1->getSyncopeCostArray(),  INT_VAR_DEGREE_MAX
              ↪ (), INT_VAL_MIN());
103     }
104     branch(*this, solutionArray, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
105
106
107 }
108 // COPY CONSTRUCTOR
109 TwoVoiceCounterpoint::TwoVoiceCounterpoint(TwoVoiceCounterpoint& s) :
      ↪ CounterpointProblem(s){
110     species = s.species;
111 }
112
113 IntLexMinimizeSpace* TwoVoiceCounterpoint::copy(){
114     return new TwoVoiceCounterpoint(*this);
115 }
116
117
118 string TwoVoiceCounterpoint::to_string() const {
119     string text = "";
120     text += CounterpointProblem::to_string();
121     text += "Lowest␣:␣\n";
122     text += cantusFirmus->to_string();
123     text += "\n";
124     text += "Counterpoint␣1␣:␣\n";
125     text += counterpoint_1->to_string();
126     text += "\n";
127     text += "Solution␣Array␣:␣\n";
128     text += intVarArray_to_string(solutionArray);
129     text += "\n";
130     text += counterpoint_1->to_string();
131     return text;
132 }
```

A.5 Voice source files

Voice.cpp

```
1   //
2   // Created by Luc Cleenewerk and Diego de Patoul.
3   //
4
5   #include "../headers/Voice.hpp"
6
7   Voice::Voice(Home home, int nMes, int lb, int ub){
8       nMeasures = nMes;
9       size = nMes*4;
10
11      lowerBound = lb;
12      upperBound = ub;
13
14      notes = IntVarArray(home, size-3, lowerBound, upperBound);
15      //this is so that species other than first species are allowed to go beneath
          ↪ the lowest h_interval, since it is only calculated using the first
          ↪ note
16      //(so 2nd, 3rd or 4th note couldn't be smaller than 0)
17      h_intervals = IntVarArray(home, size-3, -PERFECT_OCTAVE, PERFECT_OCTAVE);
18      m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE,
          ↪ PERFECT_OCTAVE);
19
20  }
21
22
23  string Voice::to_string() const {
24      string text = "voice : ";
25      text += "VOICE TOSTRING NOT IMPLEMENTED YET";
26      text += "\n";
27      return text;
28  }
29
30
31  // clone constructor
32  Voice::Voice(Home home, Voice &s){
33      nMeasures = s.nMeasures;
34      size = s.size;
35      lowerBound = s.lowerBound;
36      upperBound = s.upperBound;
37      notes.update(home, s.notes);
38      h_intervals.update(home, s.h_intervals);
39      m_intervals_brut.update(home, s.m_intervals_brut);
40      motions.update(home, s.motions);
41  }
42
43  Voice* Voice::clone(Home home){
44      return new Voice(home, *this);
45  }
46
47  IntVarArgs Voice::getFirstNotes(){
48      return notes.slice(0, 4/notesPerMeasure.at(FIRST_SPECIES),notes.size());
49  }
50
51  IntVarArray Voice::getMelodicIntervals(){
52      return m_intervals_brut;
53  }
54
55  IntVarArray Voice::getHIntervals(){
56      return h_intervals;
57  }
58
59  IntVarArgs Voice::getSecondHInterval(){
60      return h_intervals.slice(2,4,h_intervals.size());
```

```
}

IntVarArray Voice::getMotions(){
    return motions;
}
```

Stratum.cpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../headers/Stratum.hpp"

Stratum::Stratum(Home home, int nMes, int lb, int ub) : Voice(home, nMes, lb, ub)
  ↪ {
    //notes = IntVarArray(home, nMes*4-3, 0, 127);
    //h_intervals = IntVarArray(home, notes.size(), -PERFECT_OCTAVE,
      ↪ PERFECT_OCTAVE);
    //m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE,
      ↪ PERFECT_OCTAVE);
}

//if the stratum is in the upper strata, this constructor will create the
  ↪ h_intervals to the lowest stratum
Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes) :
  ↪ Stratum(home, nMes, lb, ub){
    for(int i = 0; i < h_intervals.size(); i++){
        rel(home, (h_intervals[i])==((notes[i]-lowestNotes[i])%12));
    }
}

//this constructor applies 3rd voice specific constraints which apply to the
  ↪ upper strata
Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes,
  ↪ int nV) : Stratum(home, nMes, lb, ub, lowestNotes){

    //1.H3 (formerly G8) Last chord can only consist of notes of the harmonic
      ↪ triad
    if (activeConstraints[STRATUM_UPPER_1H3]) {
        dom(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IntSet(
          ↪ IntArgs(TRIAD)));
    }

    //REMOVED
    // //H10 No tenths in last chord
    // if (activeConstraints[STRATUM_UPPER_1H10]) {
    //     rel(home, ((notes[notes.size()-1]-lowestNotes[lowestNotes.size()-1])
      ↪ >12) >> (expr(home, abs(h_intervals[h_intervals.size()-1]))!=
      ↪ MINOR_THIRD &&
    //         expr(home, abs(h_intervals[h_intervals.size()-1]))!=MAJOR_THIRD));
    // }

    //H12 Last chord cannot include a minor third
    if (activeConstraints[STRATUM_UPPER_1H12]) {
        rel(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IRT_NQ, 3);
    }
}

Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes,
  ↪ int nV1, int nV2) : Stratum(home, nMes, lb, ub, lowestNotes){
    //1.H3 (formerly G8) Last chord can only consist of notes of the harmonic
      ↪ triad
    if (activeConstraints[STRATUM_1H3]) {
```

```
        dom(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IntSet(IntArgs(TRIAD)));
    }

    //H12 Last chord cannot include a minor third
    if (activeConstraints[STRATUM_1H12]) {
        rel(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IRT_NQ, 3);
    }
}


string Stratum::to_string() const {
    string text = "Notes : ";
    text += intVarArray_to_string(notes);
    text += "\n";
    text += "H_intervals : ";
    text += intVarArray_to_string(h_intervals);
    text += "\n";
    text += "M_intervals : ";
    text += intVarArray_to_string(m_intervals_brut);
    text += "\n";
    return text;
}


// clone constructor
Stratum::Stratum(Home home,  Stratum &s) : Voice(home, s){
    // update/clone here all variables that are not in voice
}

Stratum* Stratum::clone(Home home){
    return new Stratum(home, *this);
}

void Stratum::setNote(Home home, int index, IntVar note){
    rel(home, notes[index], IRT_EQ, note);
}
```

Part.cpp

```
    //
// Created by Damien Sprockeels on 11/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#include "../../headers/Parts/Part.hpp"

/// This class represents a part, so it creates all the variables associated to that part and posts the constraints that are species independent
Part::Part(Home home, int nMes, Species sp, vector<int> cf, int lb, int ub, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs,
    int nV, int bm) :
    Voice(home, nMes, lb, ub){
    nVoices         = nV;
    species = sp;
    borrowMode      = bm;
    voice_type      = v_type;
    //lowest          = low;
    isNotLowest        = BoolVarArray(home, nMeasures, 0, 1);
    isHighest          = BoolVarArray(home, nMeasures, 0, 1);

    borrowed_scale = get_all_notes_from_scale(cf[0]%12, BORROWED_SCALE);
    scale = get_all_notes_from_scale(cf[0]%12, MAJOR_SCALE);
    chromatic_scale = get_all_notes_from_scale(cf[0]%12, CHROMATIC_SCALE);
```

```
    cp_range = {};

    secondCost = m_costs[0];
    thirdCost = m_costs[1];
    fourthCost = m_costs[2];
    tritoneCost = m_costs[3];
    fifthCost = m_costs[4];
    sixthCost = m_costs[5];
    seventhCost = m_costs[6];
    octaveCost = m_costs[7];

    borrowCost = g_costs[0];
    h_fifthCost = g_costs[1];
    h_octaveCost = g_costs[2];
    succCost = g_costs[3];
    varietyCost = g_costs[4];
    triadCost = g_costs[5];
    directMoveCost = g_costs[6];
    penultCost = g_costs[7];

    penultSixthCost = s_costs[0];
    cambiataCost = s_costs[1];
    mSkipCost = s_costs[2];
    triad3rdCost = s_costs[3];
    m2ZeroCost = s_costs[4];
    syncopationCost = s_costs[5];
    prefSlider = s_costs[6];

    directCost = 0;
    obliqueCost = 1;
    contraryCost = 2;
}

string Part::to_string() const{
    string part = "Part characteristics :\n";
    part += "Part costs : \n";
    part += intVarArray_to_string(costs);
    part += "\n";
    return part;
}



Part::Part(Home home, Part& s) : Voice(home, s) {
    nVoices = s.nVoices;
    borrowMode = s.borrowMode;

    borrowed_scale = s.borrowed_scale;
    scale = s.scale;
    chromatic_scale = s.chromatic_scale;
    species = s.species;
    voice_type = s.voice_type;

    cp_range = s.cp_range;

    extended_domain = s.extended_domain;
    off_domain = s.off_domain;

    secondCost = s.secondCost;
    thirdCost = s.thirdCost;
    fourthCost = s.fourthCost;
    tritoneCost = s.tritoneCost;
    fifthCost = s.fifthCost;
    sixthCost = s.sixthCost;
    seventhCost = s.seventhCost;
    octaveCost = s.octaveCost;

```

```
91      borrowCost = s.borrowCost;
92      h_fifthCost = s.h_fifthCost;
93      h_octaveCost = s.h_octaveCost;
94      succCost = s.succCost;
95      varietyCost = s.varietyCost;
96      triadCost = s.triadCost;
97      directMoveCost = s.directMoveCost;
98      penultCost = s.penultCost;
99
100     penultSixthCost = s.penultSixthCost;
101     cambiataCost = s.cambiataCost;
102     mSkipCost = s.mSkipCost;
103     triad3rdCost = s.triad3rdCost;
104     m2ZeroCost = s.m2ZeroCost;
105     syncopationCost = s.syncopationCost;
106     prefSlider = s.prefSlider;
107
108     directCost = s.directCost;
109     obliqueCost = s.obliqueCost;
110     contraryCost = s.contraryCost;
111
112     cost_names = s.cost_names;
113
114     toCombineCosts = s.toCombineCosts;
115     toCombineCostNames = s.toCombineCostNames;
116
117     melodicDegreeCost.update(home, s.melodicDegreeCost);
118     fifthCostArray.update(home, s.fifthCostArray);
119     octaveCostArray.update(home, s.octaveCostArray);
120     is_off.update(home, s.is_off);
121     offCostArray.update(home, s.offCostArray);
122     costs.update(home, s.costs);
123     varietyCostArray.update(home, s.varietyCostArray);
124     directCostArray.update(home, s.directCostArray);
125     isConsonance.update(home, s.isConsonance);
126     isNotLowest.update(home, s.isNotLowest);
127     isHighest.update(home, s.isHighest);
128
129     firstSpeciesHarmonicIntervals.update(home, s.firstSpeciesHarmonicIntervals);
          ↪  // Interval for the first note of each measure
130     firstSpeciesNotesCp.update(home, s.firstSpeciesNotesCp);
131     firstSpeciesMelodicIntervals.update(home, s.firstSpeciesMelodicIntervals);
132     firstSpeciesMotions.update(home, s.firstSpeciesMotions);
133     firstSpeciesMotionCosts.update(home, s.firstSpeciesMotionCosts);
134
135     isDiminution.update(home, s.isDiminution);
136     penultCostArray.update(home, s.penultCostArray);
137     secondSpeciesMotions.update(home, s.secondSpeciesMotions);
138     secondSpeciesMelodicIntervals.update(home, s.secondSpeciesMelodicIntervals);
139     secondSpeciesRealMotions.update(home, s.secondSpeciesRealMotions);
140
141     is5QNArray.update(home, s.is5QNArray);
142     thirdSpeciesHarmonicIntervals.update(home, s.thirdSpeciesHarmonicIntervals);
          ↪  // Interval for the each measure
143     thirdSpeciesMelodicIntervals.update(home, s.thirdSpeciesMelodicIntervals);
144     cambiataCostArray.update(home, s.cambiataCostArray);
145
146     isNoSyncopeArray.update(home, s.isNoSyncopeArray);
147     snycopeCostArray.update(home, s.snycopeCostArray);
148
149     speciesArray.update(home, s.speciesArray);
150
151     toCombineCosts.update(home, s.toCombineCosts);
152 }
153
154 // Virtual clone function
155 Part* Part::clone(Home home) {
156     return new Part(home, *this);
```

```
157 }
158
159 IntVarArray Part::getBranchingNotes(){
160     return notes;
161 }
162
163 IntVarArray Part::getPartNotes(){
164     return notes;
165 }
166
167 int Part::getSuccCost(){
168     return succCost;
169 }
170
171 IntVarArray Part::getFirstHInterval(){
172     return h_intervals;
173 }
174
175 IntVarArray Part::getMotions(){
176     return motions;
177 }
178
179 IntVarArray Part::getFirstMInterval(){
180     return m_intervals_brut;
181 }
182
183 IntVarArray Part::getCosts(){
184     return costs;
185 }
186
187 int Part::getVarietyCost(){
188     return varietyCost;
189 }
190
191 IntVar Part::getVarietyArray(int idx){
192     return varietyCostArray[idx];
193 }
194
195 int Part::getHIntervalSize(){
196     return h_intervals.size();
197 }
198
199 int Part::getTriadCost(){
200     return triadCost;
201 }
202
203 int Part::getSecondCost(){
204     return secondCost;
205 }
206 int Part::getThirdCost(){
207     return thirdCost;
208 }
209 int Part::getFourthCost(){
210     return fourthCost;
211 }
212 int Part::getTritoneCost(){
213     return tritoneCost;
214 }
215 int Part::getFifthCost(){
216     return fifthCost;
217 }
218 int Part::getSixthCost(){
219     return sixthCost;
220 }
221 int Part::getSeventhCost(){
222     return seventhCost;
223 }
224 int Part::getOctaveCost(){
```

```
225     return octaveCost;
226 }
227
228 IntVarArray Part::getMelodicDegreeCost(){
229     return melodicDegreeCost;
230 }
231
232 vector<string> Part::getCostNames(){
233     return cost_names;
234 }
235
236 BoolVarArray Part::getConsonance(){
237     return isConsonance;
238 }
239
240 void Part::add_cost(Home home, int idx, IntVarArray to_be_added, IntVarArray
      ↪  costs){
241     int sz = to_be_added.size();
242     IntVarArgs args(sz);
243     for(int i = 0; i < sz; i++){
244         args[i] = to_be_added[i];
245     }
246     rel(home, costs[idx], IRT_EQ, expr(home, sum(args)));
247 }
248
249 void Part::add_toCombineCost(Home home, int idx, IntVarArray to_be_added,
      ↪  IntVarArray costs) {
250     int sz = to_be_added.size();
251     IntVarArgs args(sz);
252     for(int i = 0; i < sz; i++){
253         args[i] = to_be_added[i];
254     }
255     rel(home, costs[idx], IRT_EQ, expr(home, sum(args)));
256 }
257
258 BoolVarArray Part::getIsNotLowest(){
259     return isNotLowest;
260 }
261
262 IntVarArray Part::getFirstSpeciesHIntervals(){
263     return firstSpeciesHarmonicIntervals;
264 }
265
266 IntVarArray Part::getFirstSpeciesNotes(){
267     return firstSpeciesNotesCp;
268 }
269
270 IntVarArray Part::getFirstSpeciesMIntervals(){
271     return firstSpeciesMelodicIntervals;
272 }
273
274 IntVarArray Part::getFifthCostArray(){
275     return fifthCostArray;
276 }
277
278 IntVarArray Part::getOctaveCostArray(){
279     return octaveCostArray;
280 }
281
282 int Part::getHFifthCost(){
283     return h_fifthCost;
284 }
285
286 int Part::getHOctaveCost(){
287     return h_octaveCost;
288 }
289
290 IntVarArray Part::getFirstSpeciesMotions(){
```

```
291     return firstSpeciesMotions;
292 }
293
294 IntVarArray Part::getDirectCostArray(){
295     return directCostArray;
296 }
297
298 int Part::getDirectCost(){
299     return directCost;
300 }
301
302 BoolVarArray Part::getIsOffArray(){
303     return is_off;
304 }
305
306 vector<int> Part::getOffDomain(){
307     return off_domain;
308 }
309
310 vector<int> Part::getExtendedDomain(){
311     return extended_domain;
312 }
313
314 IntVarArray Part::getOffCostArray(){
315     return offCostArray;
316 }
317
318 int Part::getBorrowCost(){
319     return borrowCost;
320 }
321
322 BoolVarArray Part::getIsDiminution(){
323     return isDiminution;
324 }
325
326 int Part::getPenultCost(){
327     return penultCost;
328 }
329
330 IntVarArray Part::getPenultCostArray(){
331     return penultCostArray;
332 }
333
334 IntVarArray Part::getSecondSpeciesMotions(){
335     return secondSpeciesMotions;
336 }
337
338 IntVarArray Part::getSecondSpeciesMIntervals(){
339     return secondSpeciesMelodicIntervals;
340 }
341
342 IntVarArray Part::getSecondSpeciesRealMotions(){
343     return secondSpeciesRealMotions;
344 }
345
346 int Part::getDirectMoveCost(){
347     return directMoveCost;
348 }
349
350 BoolVarArray Part::getIs5QNArray(){
351     return is5QNArray;
352 }
353
354 IntVarArray Part::getThirdSpeciesHIntervals(){
355     return thirdSpeciesHarmonicIntervals;
356 }
357
358 IntVarArray Part::getThirdSpeciesMIntervals(){
```

```
    return thirdSpeciesMelodicIntervals;
}

int Part::getCambiataCost(){
    return cambiataCost;
}

IntVarArray Part::getCambiataCostArray(){
    return cambiataCostArray;
}

BoolVarArray Part::getNoSyncope(){
    return isNoSyncopeArray;
}

IntVarArray Part::getSyncopeCostArray(){
    return snycopeCostArray;
}

IntVarArray Part::getFourthSpeciesMIntervals(){
    return fourthSpeciesMelodicIntervals;
}

IntVarArray Part::getSpeciesArray(){
    return speciesArray;
}

BoolVarArray Part::getIsHighest(){
    return isHighest;
}

IntVarArray Part::getToCombineCosts(){
    return toCombineCosts;
}

vector<string> Part::getToCombineCostNames(){
    return toCombineCostNames;
}
```

CantusFirmus.cpp

```
    //
// Created by Damien Sprockeels on 12/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#include "../../headers/Parts/CantusFirmus.hpp"

CantusFirmus::CantusFirmus(Home home, int size, vector<int> cf, Stratum* low, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int nV) :
    Part(home, size, CANTUS_FIRMUS, cf, 0, 127, v_type, m_costs, g_costs, s_costs, nV, -1){
    cf_vector = cf;
    notes = IntVarArray(home, size, lowerBound, upperBound);
    /// Melodic intervals for the first species notes
    m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
    h_intervals = IntVarArray(home, size, -PERFECT_OCTAVE, PERFECT_OCTAVE);
    ///link melodic intervals
    for(int i = 0; i < m_intervals_brut.size(); i++)
        rel(home, m_intervals_brut[i], IRT_EQ, expr(home, notes[i+1] - notes[i]));
    for(int i = 0; i < size; i++){
```

```
19          rel(home, h_intervals[i], IRT_EQ, expr(home, (notes[i]-low->getFirstNotes
              ↪  ()[i])%12));
20      }
21
22      for(int i = 0; i < size; i++)
23          rel(home, notes[i], IRT_EQ, cf_vector[i]);
24
25      motions = IntVarArray(home, notes.size()-1, IntSet{-1, CONTRARY_MOTION,
            ↪  OBLIQUE_MOTION, PARALLEL_MOTION});
26      //create motions
27
28      for(int i = 0; i < motions.size(); i++){
29          //direct motions help creation
30
31          BoolVar both_up = expr(home, (m_intervals_brut[i]>0)&&(low->
              ↪  getMelodicIntervals()[i]>0)); //if both parts are going in the
              ↪  same direction
32          BoolVar both_stay = expr(home, (m_intervals_brut[i]==0)&&(low->
              ↪  getMelodicIntervals()[i]==0)); //if both parts are staying
33          BoolVar both_down = expr(home, (m_intervals_brut[i]<0)&&(low->
              ↪  getMelodicIntervals()[i]<0)); //if both parts are going down
34          //oblique motions help creation
35          BoolVar cf_stays_1 = expr(home, (m_intervals_brut[i]>0)&&(low->
              ↪  getMelodicIntervals()[i]==0)); //if the lowest part stays and one
              ↪  goes up
36          BoolVar cf_stays_2 = expr(home, (m_intervals_brut[i]<0)&&(low->
              ↪  getMelodicIntervals()[i]==0)); //if the lowest part stays and one
              ↪  goes down
37          BoolVar cp_stays_1 = expr(home, (m_intervals_brut[i]==0)&&(low->
              ↪  getMelodicIntervals()[i]>0)); //if the lowest part goes up and one
              ↪   stays
38          BoolVar cp_stays_2 = expr(home, (m_intervals_brut[i]==0)&&(low->
              ↪  getMelodicIntervals()[i]<0)); //if the lowest part goes down and
              ↪  one stays
39          //contrary motions help creation
40          BoolVar cpd_cfu = expr(home, (m_intervals_brut[i]<0)&&(low->
              ↪  getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
41          BoolVar cpu_cfd = expr(home, (m_intervals_brut[i]>0)&&(low->
              ↪  getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
42
43          //direct constraints
44          rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >>
              ↪   (motions[i]==PARALLEL_MOTION));
45          //oblique constraints
46          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪  isNotLowest[i]==1)) >> (motions[i]==OBLIQUE_MOTION));
47          //contrary constraints
48          rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (motions[i]==
              ↪  CONTRARY_MOTION));
49          //bass constraints
50          rel(home, (isNotLowest[i]==0) >> (motions[i]==-1));
51      }
52
53
54
55      // 1.H1 cf version (commented by Tom Lai)
56      if (activeConstraints[CF_1H1]) {
57          dom(home, h_intervals, IntSet({UNISSON, MINOR_THIRD, MAJOR_THIRD,
              ↪  PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
58              -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -
                  ↪  MAJOR_SIXTH, -PERFECT_OCTAVE}));
59
60          // disArray = IntVarArray(home, notes.size(), IntSet{0, 1});
61          // // Define the set of consonant intervals
62          // IntSet consonantIntervals({UNISSON, MINOR_THIRD, MAJOR_THIRD,
              ↪  PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
63          //     -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -
              ↪  MAJOR_SIXTH, -PERFECT_OCTAVE});
```

```
64
65          // // Loop through each harmonic interval
66          // for (size_t i = 0; i < h_intervals.size(); i++) {
67          //     // Create a Boolean variable to check if h_intervals[i] is in
                ↪ consonantIntervals
68          //     BoolVar isConsonant(home, 0, 1);
69          //     dom(home, h_intervals[i], consonantIntervals, isConsonant);
70          // // Loop through each harmonic interval
71          // for (size_t i = 0; i < h_intervals.size(); i++) {
72          //     // Create a Boolean variable to check if h_intervals[i] is in
                ↪ consonantIntervals
73          //     BoolVar isConsonant(home, 0, 1);
74          //     dom(home, h_intervals[i], consonantIntervals, isConsonant);
75
76          //     // If the interval is consonant, set disArray[i] to 0
77          //     rel(home, isConsonant >> (disArray[i] == 0));
78
79          //     // Otherwise, set disArray[i] to H1_1_cost
80          //     rel(home, !isConsonant >> (disArray[i] == 1));
81          // }
82      }
83
84      if(nV==TWO_VOICES){
85          /// 1.H2 from Thibault: The first harmonic interval must be a perfect
                ↪ consonance
86          if (activeConstraints[CF_1H2_2V]) {
87              dom(home, h_intervals[0], IntSet(IntArgs(PERFECT_CONSONANCES)));
88          }
89
90          /// 1.H3 from Thibault: The last harmonic interval must be a perfect
                ↪ consonance
91          if (activeConstraints[CF_1H3_2V]) {
92              dom(home, h_intervals[h_intervals.size()-1], IntSet(IntArgs(
                    ↪ PERFECT_CONSONANCES)));
93          }
94
95          //H7,H8 cf version
96          if (activeConstraints[CF_1H7_2V]) {
97              // rel(home, expr(home, notes[notes.size()-2] - low->getNotes()[low->
                    ↪ getNotes().size()-2]), IRT_EQ, MINOR_THIRD, Reify(isNotLowest[
                    ↪ isNotLowest.size()-2], RM_IMP));
98              LinIntExpr penultInterval = expr(home, notes[notes.size()-2] - low->
                    ↪ getNotes()[low->getNotes().size()-2]);
99              rel(home, (isNotLowest[isNotLowest.size()-2]==1) >> (penultInterval==
                    ↪ MINOR_THIRD || penultInterval==MAJOR_THIRD));
100         }
101
102
103         //P1 from Thibault : Perfect consonances cannot be reached by direct
                ↪ motion
104         if (activeConstraints[CF_1P1]) {
105             for(int j = 0; j < motions.size(); j++){
106                 rel(home, ((h_intervals[j+1]==UNISSON || h_intervals[j+1]==
                        ↪ PERFECT_FIFTH)&&isNotLowest[j+1]==1) >>
107                     (motions[j]!=PARALLEL_MOTION));
108             }
109         }
110
111     } else {
112         //H7,H8 cf version, 3v adapted
113         if (activeConstraints[CF_1H7_3V]) {
114             LinIntExpr penultInterval = (notes[notes.size()-2] - low->getNotes()[
                    ↪ low->getNotes().size()-2])%12;
115             rel(home, penultInterval==UNISSON || penultInterval==MINOR_THIRD ||
                    ↪ penultInterval==PERFECT_FIFTH || penultInterval==MAJOR_SIXTH);
116         }
117     }
118
```

```
    //No battuta
    if (activeConstraints[CF_1P3]) {
        P3_0_noBattuta(home, this);
    }

    // combined costs
    toCombineCosts = IntVarArray(home, 1, 0, 10000);
    // toCombineCostNames = {"1H1"};
    //need to set constraintCosts[0]
    // add_toCombineCost(home, 0, disArray, toCombineCosts);
}

string CantusFirmus::to_string() const {
    string text = "Cantus Firmus notes : ";
    text += intVarArray_to_string(notes);
    text += "\n";
    text += "H intervals brut : ";
    text += intVarArray_to_string(h_intervals);
    text += "\n";
    text += "Motions : ";
    text += intVarArray_to_string(motions);
    text += "\n";
    text += "is Lowest : ";
    text += boolVarArray_to_string(isNotLowest);
    text += "\n";
    text += "is Highest : ";
    text += boolVarArray_to_string(isHighest);
    text += "\n";
    return text;
}


// clone constructor
CantusFirmus::CantusFirmus(Home home, CantusFirmus &s) : Part(home, s){

}

CantusFirmus* CantusFirmus::clone(Home home){
    return new CantusFirmus(home, *this);
}

IntVarArray CantusFirmus::getFirstHInterval(){
    return h_intervals;
}

IntVarArray CantusFirmus::getMotions(){
    return motions;
}

IntVarArray CantusFirmus::getFirstMInterval(){
    return m_intervals_brut;
}

IntVarArgs CantusFirmus::getFirstNotes(){
    return notes.slice(0, 4/4, notes.size());
}

int CantusFirmus::getHIntervalSize(){
    return h_intervals.size();
}
```

FifthSpeciesCounterpoint.cpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
```

```
3  //
4
5  #include "../../headers/Parts/FifthSpeciesCounterpoint.hpp"
6
7  /**
8   * GENERAL CONSTRUCTOR
9   */
10 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<
     ↪ int> cf, int lb, int ub, Species mSpecies, Stratum* low, CantusFirmus* c,
11     int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs,
         ↪ int bm, int nV):
12     Part(home, nMes, mSpecies, cf, lb, ub, v_type, m_costs, g_costs, s_costs, nV,
         ↪  bm)
13 {
14
15     for(int i = lowerBound; i <= upperBound; i++){
16         cp_range.push_back(i);
17     }
18     // cout << lowerBound << endl;
19     // cout << upperBound << endl;
20     /*
21     if borrowMode is enabled, the extended domain is extended to make the
         ↪ inclusion of borrowed notes possible. We can see from Fux's examples
22     that he does like to borrow notes, so the borrow cost should just do the job
         ↪ and still allow borrowed notes, not outright forbid them
23     */
24     if(borrowMode==1){
25         extended_domain = vector_union(cp_range, vector_union(scale,
             ↪ borrowed_scale));
26     } else {
27         extended_domain = vector_intersection(cp_range, vector_union(scale,
             ↪ borrowed_scale));
28     }
29
30     off_domain = vector_difference(vector_intersection(cp_range, scale),
         ↪ lowerBound, upperBound);
31
32     solutionLength = notes.size();
33     speciesArray = IntVarArray(home, solutionLength, IntSet({-1, THIRD_SPECIES,
         ↪ FOURTH_SPECIES}));
34     isNthSpeciesArray = BoolVarArray(home, notes.size()*5, 0, 1);
35     isConstrainedArray = BoolVarArray(home, solutionLength, 0, 1);
36     isThirdSpeciesArray = BoolVarArray(home, notes.size(), 0, 1);
37     isFourthSpeciesArray = BoolVarArray(home, notes.size(), 0, 1);
38
39     /**
40      * CREATE THE SPECIES ARRAY
41      */
42     createSpeciesArrays(home);
43
44     fifthSpeciesNotesCp = IntVarArray(home, notes.size(), IntSet(IntArgs(
         ↪ vector_intersection(cp_range, extended_domain))));
45     if(borrowMode==1){
46         fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2] = IntVar(home, IntSet(
             ↪ IntArgs(vector_intersection(cp_range, chromatic_scale))));
47     }
48
49
50     fifthSpeciesHIntervals = IntVarArray(home, h_intervals.size(), -MAX_STEP,
         ↪ MAX_STEP);
51
52     firstSpeciesHarmonicIntervals = IntVarArray(home, fifthSpeciesHIntervals.
         ↪ slice(0, 4, fifthSpeciesHIntervals.size()));
53
54         //Harmonic intervals
55     for (int i = 0; i < h_intervals.size(); i++) {
56         BoolVar isFourthSpeciesNote(home, 0, 1);
```

```
57          rel(home, isFourthSpeciesArray[i], IRT_EQ, 1, Reify(isFourthSpeciesNote))
              ↪ ;
58          BoolVar isJoinedDegree(home, 0, 1);
59          if (i < h_intervals.size() - 2) {
60              rel(home, expr(home, abs(fifthSpeciesNotesCp[i+2] -
                  ↪ fifthSpeciesNotesCp[i])), IRT_LQ, 2, Reify(isJoinedDegree));
61          } else {
62              rel(home, isJoinedDegree == 0); // last two notes cannot be joined
63          }
64          rel(home, (isFourthSpeciesNote == 0) >> (h_intervals[i] == (
                ↪ fifthSpeciesNotesCp[i] - low->getNotes()[i]) % 12)); // thesis
65          rel(home, (isFourthSpeciesNote==1 && isJoinedDegree == 0) >> (h_intervals
                ↪ [i]==(notes[i]-low->getNotes()[i])%12));      // thesis
66          if (i < h_intervals.size() - 2) {
67              rel(home, (isFourthSpeciesNote==1 && isJoinedDegree == 1) >> (
                    ↪ h_intervals[i]==(notes[i+2]-low->getNotes()[i])%12));      //
                    ↪ arsis
68          }
69      }
70
71      // for(int i = 0; i < fifthSpeciesHIntervals.size(); i++){
72      //     rel(home, (fifthSpeciesHIntervals[i])==((fifthSpeciesNotesCp[i]-low->
            ↪ getNotes()[floor(i/4)*4])%12));
73      // }
74      // //rel(home, (fifthSpeciesHIntervals[0])==((fifthSpeciesNotesCp[2]-low->
            ↪ getNotes()[0])%12));
75
76      fifthSpeciesSuccMIntervals = IntVarArray(home, m_intervals_brut.size(), -
            ↪ PERFECT_OCTAVE, PERFECT_OCTAVE);
77
78      firstSpeciesMelodicIntervals = IntVarArray(home, nMeasures-1, -PERFECT_OCTAVE
            ↪ , PERFECT_OCTAVE);
79
80      for(int i = 0; i < fifthSpeciesSuccMIntervals.size(); i++){
81          rel(home, fifthSpeciesSuccMIntervals[i] == (fifthSpeciesNotesCp[i+1]-
                ↪ fifthSpeciesNotesCp[i]));
82      }
83
84      for(int i = 0; i < firstSpeciesMelodicIntervals.size(); i++){
85          rel(home, firstSpeciesMelodicIntervals[i] == (fifthSpeciesNotesCp[(i+1)
                ↪ *4]-fifthSpeciesNotesCp[i*4]));
86      }
87
88      fifthSpeciesMIntervals = IntVarArray(home, m_intervals_brut.size(), -16, 16);
89      for(int i = 0; i < fifthSpeciesMIntervals.size(); i+=4){
90          rel(home, fifthSpeciesMIntervals[i+2] == (fifthSpeciesNotesCp[i+4]-
                ↪ fifthSpeciesNotesCp[i+2]));
91          rel(home, fifthSpeciesMIntervals[i+3] == (fifthSpeciesNotesCp[i+4]-
                ↪ fifthSpeciesNotesCp[i+3]));
92      }
93
94      fifthSpeciesMTAIntervals = IntVarArray(home, nMeasures-1, -16, 16);
95      for(int i = 0; i < fifthSpeciesMTAIntervals.size(); i++){
96          rel(home, fifthSpeciesMTAIntervals[i] == (fifthSpeciesNotesCp[(i*4)+2]-
                ↪ fifthSpeciesNotesCp[(i*4)]));
97      }
98
99      m2Len = ((nMeasures-1)*4)-1;
100     fifthSpeciesM2Intervals = IntVarArray(home, m2Len, -16, 16);
101     for(int i = 0; i < m2Len; i++){
102         rel(home, fifthSpeciesM2Intervals[i] == (fifthSpeciesNotesCp[i+2]-
                ↪ fifthSpeciesNotesCp[i]));
103     }
104
105     fifthSpeciesMAllIntervals = IntVarArray(home, m_intervals_brut.size(), -
            ↪ PERFECT_OCTAVE, PERFECT_OCTAVE);
106     for(int i = 0; i < fifthSpeciesMAllIntervals.size(); i++){
```

```
107         rel(home, fifthSpeciesMAllIntervals[i] == (fifthSpeciesNotesCp[i+1]-
              ↪ fifthSpeciesNotesCp[i]));
108     }
109
110     fifthSpeciesMotions = IntVarArray(home, nMeasures-1, IntSet({-1,
          ↪ CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION}));
111     fifthSpeciesMotionsCosts = IntVarArray(home, nMeasures-1, IntSet({0,
          ↪ contraryCost, obliqueCost, directCost}));
112
113     //create motions
114     for(int i = 0; i < fifthSpeciesMotions.size(); i++){
115         //direct motions help creation
116
117         BoolVar both_up = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low->
              ↪ getMelodicIntervals()[i]>0)); //if both parts are going in the
              ↪ same direction
118         BoolVar both_stay = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(low
              ↪ ->getMelodicIntervals()[i]==0)); //if both parts are staying
119         BoolVar both_down = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low
              ↪ ->getMelodicIntervals()[i]<0)); //if both parts are going down
120         //oblique motions help creation
121         BoolVar cf_stays_1 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low
              ↪ ->getMelodicIntervals()[i]==0)); //if the lowest part stays and
              ↪ one goes up
122         BoolVar cf_stays_2 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low
              ↪ ->getMelodicIntervals()[i]==0)); //if the lowest part stays and
              ↪ one goes down
123         BoolVar cp_stays_1 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(
              ↪ low->getMelodicIntervals()[i]>0)); //if the lowest part goes up
              ↪ and one stays
124         BoolVar cp_stays_2 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(
              ↪ low->getMelodicIntervals()[i]<0)); //if the lowest part goes down
              ↪ and one stays
125         //contrary motions help creation
126         BoolVar cpd_cfu = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low->
              ↪ getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
127         BoolVar cpu_cfd = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low->
              ↪ getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
128
129         //direct constraints
130         rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i
              ↪ ]==1)) >> (fifthSpeciesMotions[i]==PARALLEL_MOTION));
131         rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i
              ↪ ]==1)) >> (fifthSpeciesMotionsCosts[i]==directCost));
132         //oblique constraints
133         rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪ this->isNotLowest[i]==1)) >> (fifthSpeciesMotions[i]==
              ↪ OBLIQUE_MOTION));
134         rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪ this->isNotLowest[i]==1)) >> (fifthSpeciesMotionsCosts[i]==
              ↪ obliqueCost));
135         //contrary constraints
136         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
              ↪ fifthSpeciesMotions[i]==CONTRARY_MOTION));
137         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
              ↪ fifthSpeciesMotionsCosts[i]==contraryCost));
138         //bass constraints
139         rel(home, (this->isNotLowest[i]==0) >> (fifthSpeciesMotions[i]==-1));
140         rel(home, (this->isNotLowest[i]==0) >> (fifthSpeciesMotionsCosts[i]==0));
141
142     }
143
144     is5QNArray = BoolVarArray(home, nMeasures-1, 0, 1);
145
146     for(int i = 0; i < is5QNArray.size()*4; i+=4){
147
148         BoolVar b1 = BoolVar(home, 0, 1);
149         BoolVar b2 = BoolVar(home, 0, 1);
```

```
150         BoolVar b3 = BoolVar(home, 0, 1);
151         BoolVar b4 = BoolVar(home, 0, 1);
152         BoolVar bb1 = BoolVar(home, 0, 1);
153         BoolVar bb2 = BoolVar(home, 0, 1);
154         BoolVar bb3 = BoolVar(home, 0, 1);
155         BoolVar bb4 = BoolVar(home, 0, 1);
156         BoolVar band1 = BoolVar(home, 0, 1);
157         BoolVar band2 = BoolVar(home, 0, 1);
158         BoolVar band3 = BoolVar(home, 0, 1);
159         BoolVar beq1 = BoolVar(home, 0, 1);
160         BoolVar beq2 = BoolVar(home, 0, 1);
161         BoolVar beq3 = BoolVar(home, 0, 1);
162
163         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i])), IRT_LQ, 2,
              ↪  Reify(b1));
164         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+1])), IRT_LQ, 2,
              ↪  Reify(b2));
165         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+2])), IRT_LQ, 2,
              ↪  Reify(b3));
166         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+3])), IRT_LQ, 2,
              ↪  Reify(b4));
167
168         rel(home, fifthSpeciesSuccMIntervals[i], IRT_GQ, 0, Reify(bb1));
169         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_GQ, 0, Reify(bb2));
170         rel(home, fifthSpeciesSuccMIntervals[i+2], IRT_GQ, 0, Reify(bb3));
171         rel(home, fifthSpeciesSuccMIntervals[i+3], IRT_GQ, 0, Reify(bb4));
172
173         rel(home, b1, BOT_AND, b2, band1);
174         rel(home, b3, BOT_AND, b4, band2);
175         rel(home, band1, BOT_AND, band2, band3);
176
177         rel(home, bb1, BOT_EQV, bb2, beq1);
178         rel(home, bb3, BOT_EQV, bb4, beq2);
179         rel(home, beq1, BOT_EQV, beq2, beq3);
180         rel(home, band3, BOT_AND, beq3, is5QNArray[i/4]);
181     }
182
183     isMostlyThirdArray = BoolVarArray(home, nMeasures-1, 0, 1);
184     for(int i = 0; i < isMostlyThirdArray.size(); i++){
185         BoolVar b23 = BoolVar(home, 0, 1);
186         /**
187          * EXPLANATION :
188          * -for each measure there are 20 entries in the isNthSpeciesArray (5
              ↪  possibilities for each note, and a measure has 4 notes)
189          * -therefor, we multiply i by 20 to always land at the start of a
              ↪  measure (0 is the first note of the first measure, 20 the first
              ↪  of the second
190          *  measure etc etc)
191          * -since each note has 5 entries to check for which type of species it
              ↪  is, to land on the first entry of the second note, we do 20+5
192          * -now to check if the second note is of the third species, we do 20+5+3
              ↪   (second measure, second note, check for third species)
193          */
194         rel(home, isNthSpeciesArray[(i*20)+8], BOT_AND, isNthSpeciesArray[(i*20)
              ↪  +13], b23);
195         rel(home, b23, BOT_AND, isNthSpeciesArray[(i*20)+18], isMostlyThirdArray[
              ↪  i]);
196
197     }
198
199     isConsonance = BoolVarArray(home, notes.size(), 0, 1);
200
201     //create isConsonance array
202     for(int i = 0; i < isConsonance.size(); i++){
203         rel(home, expr(home, (h_intervals[i]==UNISSON)||(h_intervals[i]==
              ↪  MINOR_THIRD)||(h_intervals[i]==MAJOR_THIRD)||(h_intervals[i]==
              ↪  PERFECT_FIFTH)||
```

```
204             (h_intervals[i]==MINOR_SIXTH)||(h_intervals[i]==MAJOR_SIXTH)||(
                  ↪ h_intervals[i]==PERFECT_OCTAVE)||(h_intervals[i]==-MINOR_THIRD
                  ↪ )||
205             (h_intervals[i]==-MAJOR_THIRD)||(h_intervals[i]==-PERFECT_FIFTH)||(
                  ↪ h_intervals[i]==-MINOR_SIXTH)||(h_intervals[i]==-MAJOR_SIXTH)
                  ↪ ||
206             (h_intervals[i]==-PERFECT_OCTAVE)), IRT_EQ, isConsonance[i]);
207     }
208
209     //create isThird and isFourth arrays
210     for(int i = 0; i < notes.size(); i++){
211         rel(home, isThirdSpeciesArray[i], IRT_EQ, isNthSpeciesArray[(i*5)+3]);
212         rel(home, isFourthSpeciesArray[i], IRT_EQ, isNthSpeciesArray[(i*5)+4]);
213     }
214
215     isDiminution = BoolVarArray(home, nMeasures-1, 0, 1);
216     for(int i = 0; i < isDiminution.size()*4; i+=4){
217
218         BoolVar btt3 = BoolVar(home, 0, 1);
219         BoolVar btt4 = BoolVar(home, 0, 1);
220         BoolVar bta2nd = BoolVar(home, 0, 1);
221         BoolVar btt3rd = BoolVar(home, 0, 1);
222         BoolVar bat2nd = BoolVar(home, 0, 1);
223         BoolVar band = BoolVar(home, 0, 1);
224         IntVar addition = expr(home, abs(fifthSpeciesM2Intervals[i+1]));
225         rel(home, addition, IRT_EQ, 3, Reify(btt3));
226         rel(home, addition, IRT_EQ, 4, Reify(btt4));
227         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LQ, 2, Reify(bta2nd));
228         rel(home, fifthSpeciesSuccMIntervals[i+2], IRT_LQ, 2, Reify(bat2nd));
229         rel(home, btt3, BOT_OR, btt4, btt3rd);
230         rel(home, bta2nd, BOT_AND, btt3rd, band);
231         rel(home, band, BOT_AND, bat2nd, isDiminution[i/4]);
232
233     }
234
235     //set cambiata array
236     isNotCambiata = BoolVarArray(home, nMeasures-1, 0, 1);
237     for(int i = 0; i < isNotCambiata.size(); i++){
238         rel(home, ((fifthSpeciesHIntervals[(i*4)+1]==UNISSON ||
                      ↪ fifthSpeciesHIntervals[(i*4)+1]==PERFECT_FIFTH)&&(
                      ↪ fifthSpeciesHIntervals[(i*4)+2]==UNISSON || fifthSpeciesHIntervals
                      ↪ [(i*4)+2]==PERFECT_FIFTH)
239               &&(abs(fifthSpeciesSuccMIntervals[(i*4)+1])<=2)) >> (isNotCambiata[i
                      ↪ ]==1));
240         rel(home, ((fifthSpeciesHIntervals[(i*4)+1]!=UNISSON &&
                      ↪ fifthSpeciesHIntervals[(i*4)+1]!=PERFECT_FIFTH)||(
                      ↪ fifthSpeciesHIntervals[(i*4)+2]!=UNISSON && fifthSpeciesHIntervals
                      ↪ [(i*4)+2]!=PERFECT_FIFTH)
241               ||(abs(fifthSpeciesSuccMIntervals[(i*4)+1])>2)) >> (isNotCambiata[i
                      ↪ ]==0));
242     }
243
244     is_off = BoolVarArray(home, notes.size(), 0, 1);
245     initializeIsOffArray(home, this);
246
247     isNoSyncopeArray = BoolVarArray(home, nMeasures-1, 0, 1);
248     for(int i = 0; i < isNoSyncopeArray.size(); i++){
249         rel(home, fifthSpeciesMIntervals[(i*2)+2], IRT_NQ, 0, Reify(
                      ↪ isNoSyncopeArray[i]));
250     }
251     rel(home, fifthSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(
                  ↪ FIFTH_SPECIES),(notes.size())));
252     rel(home, fifthSpeciesHIntervals, IRT_EQ, h_intervals.slice(0,4/
                  ↪ notesPerMeasure.at(FIFTH_SPECIES),h_intervals.size()));
253     rel(home, fifthSpeciesSuccMIntervals, IRT_EQ, m_intervals_brut.slice(0,4/
                  ↪ notesPerMeasure.at(FIFTH_SPECIES),m_intervals_brut.size()));
254
```

```
255     /**
          ↪  ================================================================================
          ↪
256      *                              HARMONIC CONSTRAINS
257      *
           ↪  ================================================================================
           ↪
258      */
259
260     //Only one possible value for non constrained variables
261     // for(int i = 0; i < fifthSpeciesNotesCp.size()-1; i++){
262     //     //rel(home, fifthSpeciesNotesCp[i], IRT_EQ, fifthSpeciesNotesCp[i+1],
          ↪  Reify(isNthSpeciesArray[(i*5)], RM_IMP));
263     // }
264
265     //DISABELED
266     // //is penult cons to cf
267     // if (activeConstraints[SP5_H3]) {
268     //     BoolVar isPenultConsToCf = BoolVar(home, 0, 1);
269     //     vector<int> consonances = {0,3,4,7,8,9,-3,-4,-7,-8,-9};
270     //     IntVarArray res = IntVarArray(home, consonances.size(), 0, 1);
271     //     IntVar sm = IntVar(home, 0, consonances.size());
272     //     for(int l = 0; l < consonances.size(); l++){
273     //         BoolVar b1 = BoolVar(home, 0, 1);
274     //         rel(home, h_intervals[h_intervals.size()-5], IRT_EQ, consonances[l
          ↪  ], Reify(b1));
275     //         ite(home, b1, IntVar(home, 1, 1), IntVar(home, 0, 0), res[l]);
276     //     }
277     //     IntVarArgs x(res.size());
278     //     for(int t = 0; t < consonances.size(); t++){
279     //         x[t] = res[t];
280     //     }
281     //     rel(home, sm, IRT_EQ, expr(home, sum(x)));
282     //     rel(home, sm, IRT_GR, 0, Reify(isPenultConsToCf));
283
284     //     rel(home, isFourthSpeciesArray[isFourthSpeciesArray.size()-5], BOT_AND
          ↪  , isPenultConsToCf, 0); // if the penultimate note is part of the
          ↪  fourth species (isFourthSpeciesArray[isFourthSpeciesArray.size()-5] is
          ↪   true), then it must not be consonant with the cantus firmus (
285
286     // }
287
288     // 1.H1 every thesis note should be consonant
289     if (activeConstraints[SP5_H4]) {
290         for(int i = 0; i < isConsonance.size(); i+=4){ //checks every thesis note
291             rel(home, isNthSpeciesArray[(i*5)+1], BOT_IMP, isConsonance[i], 1);
                  ↪  //first species check
292             rel(home, isNthSpeciesArray[(i*5)+2], BOT_IMP, isConsonance[i], 1);
                  ↪  //second species check
293             rel(home, isThirdSpeciesArray[i], BOT_IMP, isConsonance[i], 1);
                  ↪  //third species check
294             if(i!=isConsonance.size()-1){
295                 //rel(home, isFourthSpeciesArray[i+2], BOT_IMP, isConsonance[i
                      ↪  +2], 1);  //fourth species check
296             }
297             if(i!=0 && i!=isConsonance.size()-1){
298                 //rel(home, expr(home, isFourthSpeciesArray[i]==1 &&
                      ↪  isNoSyncopeArray[i]==0), BOT_IMP, isConsonance[i],1);
299             }
300         }
301
302         for(int i = 4; i < isConsonance.size()-1; i+=4){ //start at the first
              ↪  note of the second measure
303             rel(home, expr(home, isFourthSpeciesArray[i]==1 && isNoSyncopeArray[i
                  ↪  /4]==1), BOT_IMP, isConsonance[i], 1);
304         }
305     }
306
```

```
307     // 3.H1 five consecutive notes by joint degree (3rd species)
308     if (activeConstraints[SP5_H5]) {
309         for(int i = 0; i < is5QNArray.size(); i++){
310             BoolVar b = BoolVar(home, 0, 1);
311             rel(home, is5QNArray[i], BOT_AND, isMostlyThirdArray[i], b);
312             rel(home, b, BOT_IMP, isConsonance[i*4+2], 1);
313         }
314     }
315 
316     // 3.H2 any dissonant note implies it is surrounded by consonant notes (3rd
        ↪  species)
317     if (activeConstraints[SP5_H6]) {
318         for(int i = 0; i < isDiminution.size(); i++){
319             BoolVar band1 = BoolVar(home, 0, 1);
320             rel(home, expr(home, isConsonance[(i*4)+2] || isThirdSpeciesArray[(i
                ↪  *4)+2]), BOT_OR, isDiminution[i], 1);
321         }
322     }
323 
324     /**
        ↪  ================================================================================
        ↪
325      *                              MELODIC CONSTRAINS
326      *
         ↪  ================================================================================
         ↪
327      */
328 
329     //no melodic interval between MAJOR_SIXTH and MAJOR_SEVENTH
330     if (activeConstraints[SP5_M1]) {
331         for(int i = 0; i < fifthSpeciesMAllIntervals.size(); i++){
332             rel(home, expr(home, abs(fifthSpeciesMAllIntervals[i])), IRT_NQ,
                ↪  MAJOR_SIXTH, Reify(expr(home, isConstrainedArray[i]==1 &&
                ↪  isConstrainedArray[i+1]==1), RM_IMP));
333             rel(home, expr(home, abs(fifthSpeciesMAllIntervals[i])), IRT_NQ,
                ↪  MINOR_SEVENTH, Reify(expr(home, isConstrainedArray[i]==1 &&
                ↪  isConstrainedArray[i+1]==1), RM_IMP));
334             rel(home, expr(home, abs(fifthSpeciesMAllIntervals[i])), IRT_NQ,
                ↪  MAJOR_SEVENTH, Reify(expr(home, isConstrainedArray[i]==1 &&
                ↪  isConstrainedArray[i+1]==1), RM_IMP));
335         }
336     }
337 
338     //no unisson between two consecutive notes
339     if (activeConstraints[SP5_M2]) {
340         for(int i = 0; i < nMeasures-1; i++){
341             rel(home, fifthSpeciesNotesCp[(i*4)], IRT_NQ, fifthSpeciesNotesCp[(i
                ↪  *4)+1], Reify(expr(home, (isConstrainedArray[(i*4)]==1) && (
                ↪  isConstrainedArray[(i*4)+1]==1)), RM_IMP));
342             rel(home, fifthSpeciesNotesCp[(i*4+1)], IRT_NQ, fifthSpeciesNotesCp[(
                ↪  i*4)+2], Reify(expr(home, (isConstrainedArray[(i*4+1)]==1) &&
                ↪  (isConstrainedArray[(i*4)+2]==1)), RM_IMP));
343             rel(home, fifthSpeciesNotesCp[(i*4)+2], IRT_NQ, fifthSpeciesNotesCp[(
                ↪  i*4)+3], Reify(expr(home, (isConstrainedArray[(i*4)+2]==1) &&
                ↪  (isConstrainedArray[(i*4)+3]==1)), RM_IMP));
344             rel(home, fifthSpeciesNotesCp[(i*4)+3], IRT_NQ, fifthSpeciesNotesCp[(
                ↪  i*4)+4], Reify(expr(home, (isConstrainedArray[(i*4)+3]==1) &&
                ↪  (isConstrainedArray[(i*4)+4]==1)), RM_IMP));
345         }
346     }
347 
348     //REMOVED
349     // //no more than minor sixth interval between arsis and thesis notes
350     // if (activeConstraints[SP5_M3]) {
351     //     for(int i = 0; i < nMeasures-1; i++){
352     //         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ,
        ↪  MAJOR_SIXTH, Reify(expr(home, isConstrainedArray[i+1]==1), RM_IMP));
```

```
353     //         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ,
          ↪ MINOR_SEVENTH, Reify(expr(home, isConstrainedArray[i+1]==1), RM_IMP));
354     //         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ,
          ↪ MAJOR_SEVENTH, Reify(expr(home, isConstrainedArray[i+1]==1), RM_IMP));
355     //     }
356     // }
357
358
359     // 4.M2 ? no same syncopation
360     if (activeConstraints[SP5_M4]) {
361         for(int i = 1; i < nMeasures-1; i++){
362             rel(home, fifthSpeciesNotesCp[(i*4)], IRT_NQ, fifthSpeciesNotesCp[(i
                  ↪ *4)+2],
363                 Reify(expr(home, isFourthSpeciesArray[(i*4)]==1 &&
                      ↪ isFourthSpeciesArray[(i*4)+2]==1 && isConstrainedArray[(i
                      ↪ *4)+2]), RM_IMP));
364
365             rel(home, (isFourthSpeciesArray[(i*4)]==1 && isFourthSpeciesArray[(i
                  ↪ *4)+2]==1 && isConstrainedArray[(i*4)+2]==1) >> (
                  ↪ fifthSpeciesNotesCp[(i*4)]!=fifthSpeciesNotesCp[(i*4)+2]));
366         }
367     }
368
369     //Third note of penult measure must be below the fourth one
370     if (activeConstraints[SP5_3M3]) {
371         rel(home, fifthSpeciesSuccMIntervals[fifthSpeciesSuccMIntervals.size()
              ↪ -3], IRT_GR, MINOR_SECOND,
372         Reify(isThirdSpeciesArray[isThirdSpeciesArray.size()-2], RM_IMP));
373     }
374     //Second and third note distant by more than one semi tone from fourth note
375     if (activeConstraints[SP5_3M4]) {
376         rel(home, expr(home, abs(fifthSpeciesM2Intervals[fifthSpeciesM2Intervals.
              ↪ size()-2])), IRT_NQ, 1,
377             Reify(isThirdSpeciesArray[isThirdSpeciesArray.size()-2], RM_IMP));
378     }
379
380
381     /**
          ↪ ================================================================================
          ↪
382      *                              MOTION CONSTRAINS
383      *
           ↪ ================================================================================
           ↪
384      */
385
386     //no battuta kind of motion
387     if (activeConstraints[SP5_P1]) {
388         for(int i = 0; i < fifthSpeciesMotions.size(); i++){
389             BoolVar isCM = BoolVar(home, 0, 1);
390             BoolVar isOct = BoolVar(home, 0, 1);
391             BoolVar isCPDown = BoolVar(home, 0, 1);
392             BoolVar band1 = BoolVar(home, 0, 1);
393             BoolVar band2 = BoolVar(home, 0, 1);
394             BoolVar band3 = BoolVar(home, 0, 1);
395
396             rel(home, fifthSpeciesMotions[i], IRT_EQ, CONTRARY_MOTION, Reify(isCM
                  ↪ ));
397             rel(home, expr(home, (fifthSpeciesHIntervals[((i+1)*4)]==UNISSON ||
                  ↪ fifthSpeciesHIntervals[((i+1)*4)]==PERFECT_OCTAVE ||
398                   fifthSpeciesHIntervals[((i+1)*4)]==-PERFECT_OCTAVE)) >> isOct);
399             rel(home, fifthSpeciesMIntervals[(i*4)+3], IRT_LE, -4, Reify(isCPDown
                  ↪ ));
400
401             rel(home, isCM, BOT_AND, isOct, band1);
402             rel(home, band1, BOT_AND, isCPDown, band2);
403             rel(home, band2, BOT_AND, expr(home, c->getNotes()[i] <=
                  ↪ fifthSpeciesNotesCp[(i*4)+3]), band3);
```

```
404             rel(home, isThirdSpeciesArray[(i*4)+3], BOT_AND, band3, 0);
405         }
406     }
407 
408     //dissonant notes must be followed by the consonant note below
409     if (activeConstraints[SP5_4P1]) {
410         for(int i = 0; i < fifthSpeciesMTAIntervals.size(); i++){
411             //isFourthArray first note
412             //is Consonance array first note
413             //mTa intervals, general
414             BoolVar bNot = BoolVar(home, 0, 1);
415             BoolVar isCst = BoolVar(home, 0, 1);
416             BoolVar bAnd = BoolVar(home, 0, 1);
417 
418             rel(home, isConsonance[i*4], BOT_EQV, BoolVar(home, 0, 0), bNot);
419             rel(home, bNot, BOT_AND, isFourthSpeciesArray[i*4], bAnd);
420             rel(home, fifthSpeciesMTAIntervals[i], IRT_LE, 0, Reify(bAnd, RM_IMP)
                  ↪ );
421             rel(home, fifthSpeciesMTAIntervals[i], IRT_GQ, -2, Reify(bAnd, RM_IMP
                  ↪ ));
422         }
423     }
424 
425     //DISABELED
426     // //no second dissonant note if the cantusFirmus is at the bass
427     // if (activeConstraints[SP5_P3]) {
428     //     for(int i = 0; i < nMeasures-1; i++){
429     //         BoolVar bUni = BoolVar(home, 0, 1);
430     //         BoolVar bAnd = BoolVar(home, 0, 1);
431     //         BoolVar isCst = BoolVar(home, 0, 1);
432     //         BoolVar bAndCst = BoolVar(home, 0, 1);
433 
434     //         rel(home, isFourthSpeciesArray[((i+1)*4)], BOT_EQV, isCst, 1);
435     //         rel(home, fifthSpeciesHIntervals[(i*4)+2], IRT_EQ, 0, Reify(bUni))
          ↪ ;
436     //         rel(home, isNotLowest[i], BOT_AND, bUni, bAnd);
437     //         rel(home, bAnd, BOT_AND, isCst, bAndCst);
438     //         rel(home, fifthSpeciesHIntervals[((i+1)*4)], IRT_NQ, 1, Reify(
          ↪ bAndCst, RM_IMP));
439     //         rel(home, fifthSpeciesHIntervals[((i+1)*4)], IRT_NQ, 2, Reify(
          ↪ bAndCst, RM_IMP));
440     //     }
441     // }
442 
443     //DISABLED RULE
444     // //marcel's rule
445     // if (activeConstraints[SP5_P4]) {
446     //     for(int i = 0; i < fifthSpeciesSuccMIntervals.size()-1; i++){
447     //         BoolVar bSkip = BoolVar(home, 0, 1);
448     //         BoolVar bMbUp = BoolVar(home, 0, 1);
449     //         BoolVar bMbDown = BoolVar(home, 0, 1);
450     //         BoolVar bContrary = BoolVar(home, 0, 1);
451 
452     //         rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i])), IRT_GR,
          ↪ 2, Reify(bSkip));
453     //         rel(home, fifthSpeciesSuccMIntervals[i], IRT_GR, 0, Reify(bMbUp));
454     //         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LE, 1, Reify(
          ↪ bMbDown));
455     //         rel(home, bMbUp, BOT_EQV, bMbDown, bContrary);
456     //         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LQ, 2, Reify(bSkip,
          ↪  RM_IMP));
457     //         rel(home, bSkip, BOT_IMP, bContrary, 1);
458     //     }
459     // }
460 
461     /**
          ↪ ==========================================================================
          ↪
```

```
462         *                                COST CONSTRAINS
463         *
                ↪  ============================================================================
                ↪
464         */
465
466         //Imperfect consonances are preferred
467         fifthCostArray = IntVarArray(home, notes.size(), IntSet({0, fifthCost}));
468         octaveCostArray = IntVarArray(home, notes.size(), IntSet({0, octaveCost}));
469
470         //set fifth cost
471         for(int i = 0; i < notes.size(); i++){
472             BoolVar b = BoolVar(home, 0, 1);
473             BoolVar band = BoolVar(home, 0, 1);
474
475             rel(home, fifthSpeciesHIntervals[i], IRT_EQ, 7, Reify(b));
476             rel(home, b, BOT_AND, isConstrainedArray[i], band);
477             ite(home, band, IntVar(home, fifthCost, fifthCost), IntVar(home, 0, 0),
                    ↪  fifthCostArray[i]);
478         }
479
480         //set octave cost
481         for(int i = 0; i < notes.size(); i++){
482             BoolVar b = BoolVar(home, 0, 1);
483             BoolVar band = BoolVar(home, 0, 1);
484             BoolVar band2 = BoolVar(home, 0, 1);
485
486             rel(home, fifthSpeciesHIntervals[i], IRT_EQ, 7, Reify(b));
487             rel(home, isNotLowest[floor(i/4)], BOT_AND, isConstrainedArray[i], band2)
                    ↪  ;
488             rel(home, b, BOT_AND, band2, band);
489             ite(home, band, IntVar(home, octaveCost, octaveCost), IntVar(home, 0, 0),
                    ↪   octaveCostArray[i]);
490         }
491
492         //create off_cost array
493         offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
494         //set the cost for borrowing this note (G4 constraint, modified)
495         for(int i = 0; i < is_off.size(); i++){
496             rel(home, (is_off[i]==0) >> (offCostArray[i]==0));
497             rel(home, (is_off[i]==1 && isConstrainedArray[i]==1) >> (offCostArray[i
                    ↪  ]==borrowCost));
498             rel(home, (is_off[i]==1 && isConstrainedArray[i]==0) >> (offCostArray[i
                    ↪  ]==0));
499         }
500
501         melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({0,
                ↪  secondCost, thirdCost, fourthCost, tritoneCost, fifthCost,
502             sixthCost, seventhCost, octaveCost}));
503
504         //G7 (modified)
505         for(int i = 0; i < m_intervals_brut.size(); i+=4/notesPerMeasure.at(
                ↪  FIFTH_SPECIES)){
506             rel(home, (isConstrainedArray[i]==0 || isConstrainedArray[i+1]==0) >> (
                    ↪  melodicDegreeCost[i]==0));
507             rel(home, (abs(m_intervals_brut[i])<MINOR_THIRD && isConstrainedArray[i
                    ↪  ]==1 && isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==
                    ↪  secondCost));
508             rel(home, ((abs(m_intervals_brut[i])==MINOR_THIRD || abs(m_intervals_brut
                    ↪  [i])==MAJOR_THIRD)&& isConstrainedArray[i]==1&& isConstrainedArray
                    ↪  [i+1]==1) >> (melodicDegreeCost[i]==thirdCost));
509             rel(home, (abs(m_intervals_brut[i])==PERFECT_FOURTH && isConstrainedArray
                    ↪  [i]==1&& isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==
                    ↪  fourthCost));
510             rel(home, (abs(m_intervals_brut[i])==TRITONE && isConstrainedArray[i
                    ↪  ]==1&& isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==
                    ↪  tritoneCost));
```

```
511         rel(home, (abs(m_intervals_brut[i])==PERFECT_FIFTH && isConstrainedArray[
              ↪ i]==1&& isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==
              ↪ fifthCost));
512         rel(home, ((abs(m_intervals_brut[i])==MINOR_SIXTH || abs(m_intervals_brut
              ↪ [i])==MAJOR_SIXTH)&& isConstrainedArray[i]==1&& isConstrainedArray
              ↪ [i+1]==1) >> (melodicDegreeCost[i]==sixthCost));
513         rel(home, ((abs(m_intervals_brut[i])==MINOR_SEVENTH || abs(
              ↪ m_intervals_brut[i])==MAJOR_SEVENTH)&& isConstrainedArray[i]==1&&
              ↪ isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==seventhCost)
              ↪ );
514         rel(home, (abs(m_intervals_brut[i])==PERFECT_OCTAVE && isConstrainedArray
              ↪ [i]==1&& isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==
              ↪ octaveCost));
515     }
516
517     cambiataCostArray = IntVarArray(home, nMeasures-1, IntSet({0, cambiataCost}))
          ↪ ;
518
519     for(int i = 0; i < cambiataCostArray.size(); i++){
520         BoolVar band = BoolVar(home, 0, 1);
521         rel(home, isNotCambiata[i], BOT_AND, isMostlyThirdArray[i], band);
522         ite(home, band, IntVar(home, cambiataCost, cambiataCost), IntVar(home, 0,
              ↪  0), cambiataCostArray[i]);
523     }
524
525     m2ZeroCostArray = IntVarArray(home, m2Len, IntSet({0, m2ZeroCost}));
526
527     for(int i = 0; i < fifthSpeciesM2Intervals.size(); i++){
528         BoolVar b = BoolVar(home, 0, 1);
529         BoolVar band = BoolVar(home, 0, 1);
530         BoolVar band2 = BoolVar(home, 0, 1);
531
532         rel(home, fifthSpeciesM2Intervals[i], IRT_EQ, 0, Reify(b));
533         rel(home, isConstrainedArray[i], BOT_AND, isConstrainedArray[i+2], band2)
              ↪ ;
534         rel(home, b, BOT_AND, band2, band);
535         ite(home, band, IntVar(home, m2ZeroCost, m2ZeroCost), IntVar(home, 0, 0),
              ↪  m2ZeroCostArray[i]);
536     }
537
538     snycopeCostArray = IntVarArray(home, (fifthSpeciesMIntervals.size())/4,
          ↪ IntSet({0, syncopationCost}));
539     for(int i = 0; i < snycopeCostArray.size(); i++){
540         BoolVar b = BoolVar(home, 0, 1);
541         BoolVar band = BoolVar(home, 0, 1);
542
543         rel(home, fifthSpeciesMIntervals[(i*4)+2], IRT_NQ, 0, Reify(b));
544         rel(home, b, BOT_AND, isFourthSpeciesArray[(i*4)+2], band);
545         ite(home, band, IntVar(home, syncopationCost, syncopationCost), IntVar(
              ↪ home, 0, 0), snycopeCostArray[i]);
546     }
547 }
548
549 /**
550  * 2 VOICES CONSTRUCTOR
551  */
552 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<
      ↪ int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type,
553     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV
          ↪ ):
554 FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type,
      ↪ m_costs, g_costs, s_costs, bm, nV)
555 {
556     //following two rel are H2_1 but modified, so we check in TwoVoiceProblem
          ↪ that that constrained isn't activated if fifth species
557     if (activeConstraints[SP5_2V_1]) {
558         //if the first note is constrained, then it must be a perfect consonance
              ↪ (works)
```

```
559         rel(home, (isConstrainedArray[0]==1) >> (fifthSpeciesHIntervals[0]==
              ↪ UNISSON || fifthSpeciesHIntervals[0]==PERFECT_FIFTH));
560         //else if the first note is not constrained, then the third note must be
              ↪ a perfect consonance (works)
561         rel(home, (isConstrainedArray[0]==0) >> (fifthSpeciesHIntervals[2]==
              ↪ UNISSON || fifthSpeciesHIntervals[2]==PERFECT_FIFTH));
562     }
563
564
565     //If the cantusFirmus is in the upper part, then no hamonic seventh
566     if (activeConstraints[SP5_2V_2]) {
567         for(int j = 1; j < c->getIsNotLowest().size(); j++){
568             rel(home, (getIsNotLowest()[j]==0 && isFourthSpeciesArray[j*4]) >> (
                  ↪ firstSpeciesHarmonicIntervals[j]!=MINOR_SEVENTH &&
                  ↪ firstSpeciesHarmonicIntervals[j]!=-MINOR_SEVENTH));
569             rel(home, (getIsNotLowest()[j]==0 && isFourthSpeciesArray[j*4]) >> (
                  ↪ firstSpeciesHarmonicIntervals[j]!=MAJOR_SEVENTH &&
                  ↪ firstSpeciesHarmonicIntervals[j]!=-MAJOR_SEVENTH));
570         }
571     }
572
573
574     //H3_1 can be activated as-is
575     if (activeConstraints[SP5_2V_3]) {
576         //penultimate note (only applies if it is 3rd species)
577         BoolVar bandThen_1 = BoolVar(home, 0, 1);
578         BoolVar testNot_1 = BoolVar(home, 0, 1);
579         BoolVar bandElse_1 = BoolVar(home, 0, 1);
580         BoolVar test_1 = expr(home, c->getNotes()[c->getNotes().size()-2] <=
              ↪ fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2]);
581
582         rel(home, test_1, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()
              ↪ -2], bandThen_1);
583         rel(home, test_1, BOT_EQV, BoolVar(home, 0, 0), testNot_1);
584         rel(home, testNot_1, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.
              ↪ size()-2], bandElse_1);
585
586         rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2]-c
              ↪ ->getNotes()[c->getNotes().size()-2]), IRT_EQ, MAJOR_SIXTH, Reify(
              ↪ bandThen_1, RM_IMP));
587         rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2]-c
              ↪ ->getNotes()[c->getNotes().size()-2]), IRT_EQ, MINOR_THIRD, Reify(
              ↪ bandElse_1, RM_IMP));
588
589         //penultimate note (only applies if it is 4th species)
590         BoolVar bandThen = BoolVar(home, 0, 1);
591         BoolVar testNot = BoolVar(home, 0, 1);
592         BoolVar bandElse = BoolVar(home, 0, 1);
593         BoolVar test = expr(home, c->getNotes()[c->getNotes().size()-2] <=
              ↪ fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-3]);
594
595         rel(home, test, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()
              ↪ -2], bandThen);
596         rel(home, test, BOT_EQV, BoolVar(home, 0, 0), testNot);
597         rel(home, testNot, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size
              ↪ ()-2], bandElse);
598         rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-3]-c
              ↪ ->getNotes()[c->getNotes().size()-2]), IRT_EQ, MAJOR_SIXTH, Reify(
              ↪ bandThen, RM_IMP));
599         rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-3]-c
              ↪ ->getNotes()[c->getNotes().size()-2]), IRT_EQ, MINOR_THIRD, Reify(
              ↪ bandElse, RM_IMP));
600     }
601
602
603     //no direct motion to reach a perfect consonance
604     if (activeConstraints[SP5_2V_4]) {
605         for(int i = 0; i < fifthSpeciesMotions.size(); i++){
```

```
606             BoolVar isThirdSpeciesAndPCons = BoolVar(home, 0, 1);
607             BoolVar isPConsAndNotLowest = BoolVar(home, 0, 1);
608 
609             rel(home, isThirdSpeciesArray[i*4], BOT_AND, expr(home,
                  ↪ fifthSpeciesHIntervals[(i*4)+3]==UNISSON ||
                  ↪ fifthSpeciesHIntervals[(i*4)+3]==PERFECT_FIFTH ||
610               fifthSpeciesHIntervals[(i*4)+3]==-PERFECT_FIFTH),
                    ↪ isThirdSpeciesAndPCons);
611             rel(home, isNotLowest[i], BOT_AND, isThirdSpeciesAndPCons,
                  ↪ isPConsAndNotLowest);
612 
613             rel(home, fifthSpeciesMotions[i], IRT_NQ, PARALLEL_MOTION, Reify(
                  ↪ isPConsAndNotLowest, RM_IMP));
614 
615         }
616     }
617 
618 
619     costs = IntVarArray(home, 8, 0, 1000000);
620     cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "
          ↪ m2", "syncopation"};
621 
622     //set cost[0] to be fifth cost
623     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
          ↪ .at(FIFTH_SPECIES), fifthCostArray.size())), costs);
624     //set cost[1] to be octave cost
625     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/
          ↪ notesPerMeasure.at(FIFTH_SPECIES), octaveCostArray.size())), costs);
626     //set cost[2] to be off cost
627     add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
          ↪ at(FIFTH_SPECIES), offCostArray.size())), costs);
628     //set cost[3] to be melodic cost
629     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪ notesPerMeasure.at(FIFTH_SPECIES), melodicDegreeCost.size())), costs);
630     //set cost[4] to be motion cost
631     add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
632     //need to set cost[5] to be cambiata cost
633     add_cost(home, 5, cambiataCostArray, costs);
634     //need to set cost[6] to be m2Zero cost
635     add_cost(home, 6, m2ZeroCostArray, costs);
636     //need to set cost[7] to be syncopation cost
637     add_cost(home, 7, snycopeCostArray, costs);
638 }
639 
640 /**
641  * 3 VOICES CONSTRUCTOR
642  */
643 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<
      ↪ int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type,
644     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
          ↪ nV1, int nV2):
645 FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type,
      ↪ m_costs, g_costs, s_costs, bm, nV2)
646 {
647     directCostArray = IntVarArray(home, fifthSpeciesMotions.size()-1,IntSet({0,
          ↪ directMoveCost}));
648     varietyCostArray = IntVarArray(home, 3*(fifthSpeciesHIntervals.size()-2),
          ↪ IntSet({0, varietyCost}));
649 
650     //3.H6 : harmonic triad should be used on the second or third beat
651     if (activeConstraints[SP5_3V_1]) {
652         thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost
              ↪ }));
653         for(int i = 0; i < thirdHTriadArray.size(); i++){
654             rel(home, ((fifthSpeciesHIntervals[(i*4)+1]!=UNISSON&&
                  ↪ fifthSpeciesHIntervals[(i*4)+1]!=MINOR_THIRD&&
                  ↪ fifthSpeciesHIntervals[(i*4)+1]!=MAJOR_THIRD&&
                  ↪ fifthSpeciesHIntervals[(i*4)+1]!=PERFECT_FIFTH)&&
```

```
655                 (fifthSpeciesHIntervals[(i*4)+2]!=UNISSON&&fifthSpeciesHIntervals
                      ↪  [(i*4)+2]!=MINOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=
                      ↪  MAJOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=
                      ↪  PERFECT_FIFTH)) >>
656                 (thirdHTriadArray[i]==triad3rdCost));
657             rel(home, ((fifthSpeciesHIntervals[(i*4)+1]==UNISSON||
                  ↪  fifthSpeciesHIntervals[(i*4)+1]==MINOR_THIRD||
                  ↪  fifthSpeciesHIntervals[(i*4)+1]==MAJOR_THIRD||
                  ↪  fifthSpeciesHIntervals[(i*4)+1]==PERFECT_FIFTH)||
658                 (fifthSpeciesHIntervals[(i*4)+2]==UNISSON||fifthSpeciesHIntervals
                      ↪  [(i*4)+2]==MINOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==
                      ↪  MAJOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==
                      ↪  PERFECT_FIFTH)) >>
659                 (thirdHTriadArray[i]==0));
660         }
661     }
662
663     //1.P1 3 voices version
664     if (activeConstraints[SP5_3V_2]) {
665         for(int j = 0; j < fifthSpeciesMotions.size()-1; j++){
666             //set a cost when it is reached through direct motion, it is 0 when
                  ↪  not
667             rel(home, (fifthSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[
                  ↪  j+1]==0||firstSpeciesHarmonicIntervals[j+1]==7))>>
668                 (directCostArray[j]==directMoveCost));
669             rel(home, (fifthSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[
                  ↪  j+1]!=0&&firstSpeciesHarmonicIntervals[j+1]!=7))>>
670                 (directCostArray[j]==0));
671         }
672     }
673
674
675     costs = IntVarArray(home, 11, 0, 1000000);
676     cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "
          ↪  m2", "syncopation", "direct", "variety", "triad3"};
677
678     //set cost[0] to be fifth cost
679     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(1, 4/notesPerMeasure
          ↪  .at(FIFTH_SPECIES), fifthCostArray.size())), costs);
680     //set cost[1] to be octave cost
681     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(1, 4/
          ↪  notesPerMeasure.at(FIFTH_SPECIES), octaveCostArray.size())), costs);
682     //set cost[2] to be off cost
683     add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
          ↪  at(FIFTH_SPECIES), offCostArray.size())), costs);
684     //set cost[3] to be melodic cost
685     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪  notesPerMeasure.at(FIFTH_SPECIES), melodicDegreeCost.size())), costs);
686     //set cost[4] to be motion cost
687     add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
688     //need to set cost[5] to be cambiata cost
689     add_cost(home, 5, cambiataCostArray, costs);
690     //need to set cost[6] to be m2Zero cost
691     add_cost(home, 6, m2ZeroCostArray, costs);
692     //need to set cost[7] to be syncopation cost
693     add_cost(home, 7, snycopeCostArray, costs);
694     //need to set cost[8] to be syncopation cost
695     add_cost(home, 8, directCostArray, costs);
696     //need to set cost[8] to be syncopation cost
697     add_cost(home, 9, varietyCostArray, costs);
698     //set cost[10] to be triad h third cost
699     add_cost(home, 10, thirdHTriadArray, costs);
700 }
701
702 /**
703  * 4 VOICES CONSTRUCTOR
704  */
```

```
FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type,
    vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3):
FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type, m_costs, g_costs, s_costs, bm, nV3)
{
    directCostArray = IntVarArray(home, fifthSpeciesMotions.size()-1,IntSet({0, directMoveCost}));
    varietyCostArray = IntVarArray(home, 3*(fifthSpeciesHIntervals.size()-2), IntSet({0, varietyCost}));

    //3.H6 : harmonic triad should be used on the second or third beat
    if (activeConstraints[SP5_4V_1]) {
        thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
        for(int i = 0; i < thirdHTriadArray.size(); i++){
            rel(home, ((fifthSpeciesHIntervals[(i*4)+1]!=UNISSON&&
              fifthSpeciesHIntervals[(i*4)+1]!=MINOR_THIRD&&
              fifthSpeciesHIntervals[(i*4)+1]!=MAJOR_THIRD&&
              fifthSpeciesHIntervals[(i*4)+1]!=PERFECT_FIFTH)&&
                (fifthSpeciesHIntervals[(i*4)+2]!=UNISSON&&fifthSpeciesHIntervals[(i*4)+2]!=MINOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=MAJOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=PERFECT_FIFTH)) >>
                (thirdHTriadArray[i]==triad3rdCost));
            rel(home, ((fifthSpeciesHIntervals[(i*4)+1]==UNISSON||
              fifthSpeciesHIntervals[(i*4)+1]==MINOR_THIRD||
              fifthSpeciesHIntervals[(i*4)+1]==MAJOR_THIRD||
              fifthSpeciesHIntervals[(i*4)+1]==PERFECT_FIFTH)||
                (fifthSpeciesHIntervals[(i*4)+2]==UNISSON||fifthSpeciesHIntervals[(i*4)+2]==MINOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==MAJOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==PERFECT_FIFTH)) >>
                (thirdHTriadArray[i]==0));
        }
    }


    //1.P1 4 voices version
    if (activeConstraints[SP5_4V_2]) {
        for(int j = 0; j < fifthSpeciesMotions.size()-1; j++){
            //set a cost when it is reached through direct motion, it is 0 when not
            rel(home, (fifthSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||firstSpeciesHarmonicIntervals[j+1]==7))>>
                (directCostArray[j]==directMoveCost));
            rel(home, (fifthSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&firstSpeciesHarmonicIntervals[j+1]!=7))>>
                (directCostArray[j]==0));
        }
    }

    costs = IntVarArray(home, 11, 0, 1000000);
    cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "m2", "syncopation", "direct", "variety", "triad3"};

    //set cost[0] to be fifth cost
    add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(1, 4/notesPerMeasure.at(FIFTH_SPECIES), fifthCostArray.size())), costs);
    //set cost[1] to be octave cost
    add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(1, 4/notesPerMeasure.at(FIFTH_SPECIES), octaveCostArray.size())), costs);
    //set cost[2] to be off cost
    add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(FIFTH_SPECIES), offCostArray.size())), costs);
    //set cost[3] to be melodic cost
```

```
747     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪  notesPerMeasure.at(FIFTH_SPECIES), melodicDegreeCost.size())), costs);
748     //set cost[4] to be motion cost
749     add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
750     //need to set cost[5] to be cambiata cost
751     add_cost(home, 5, cambiataCostArray, costs);
752     //need to set cost[6] to be m2Zero cost
753     add_cost(home, 6, m2ZeroCostArray, costs);
754     //need to set cost[7] to be syncopation cost
755     add_cost(home, 7, snycopeCostArray, costs);
756     //need to set cost[8] to be syncopation cost
757     add_cost(home, 8, directCostArray, costs);
758     //need to set cost[8] to be syncopation cost
759     add_cost(home, 9, varietyCostArray, costs);
760     //set cost[10] to be triad h third cost
761     add_cost(home, 10, thirdHTriadArray, costs);
762 }
763
764 /**
765  * This function returns a string with the characteristics of the counterpoint.
       ↪  It calls the to_string() method from
766  * the Part class and adds 1st species specific characteristics.
767  * @return a string representation of the current instance of the
       ↪  FirstSpeciesCounterpoint class.
768  */
769 string FifthSpeciesCounterpoint::to_string() const {
770     string text = Part::to_string() + "\n␣Fifth␣species␣:\n";
771     text += "Fifth␣species␣notes␣:␣" + intVarArray_to_string(fifthSpeciesNotesCp)
          ↪   + "\n";
772     text += "M␣Intervals␣array␣:␣" + intVarArray_to_string(
          ↪  fifthSpeciesSuccMIntervals) + "\n";
773     text += "isMostlyThird␣:␣" + boolVarArray_to_string(isMostlyThirdArray) + "\n
          ↪  ";
774     text += "not␣Lowest␣array␣:␣" + boolVarArray_to_string(isNotLowest) + "\n";
775     text += "Species␣array␣:␣" + intVarArray_to_string(speciesArray) + "\n";
776     return text;
777 }
778
779 // clone constructor
780 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home,
       ↪  FifthSpeciesCounterpoint &s) : Part(home, s){
781     solutionLength = s.solutionLength;
782     m2Len = s.m2Len;
783     fifthSpeciesNotesCp.update(home, s.fifthSpeciesNotesCp);
784     fifthSpeciesHIntervals.update(home, s.fifthSpeciesHIntervals);
785     firstHInterval.update(home, s.firstHInterval);
786     isNthSpeciesArray.update(home, s.isNthSpeciesArray);
787     isConstrainedArray.update(home, s.isConstrainedArray);
788     fifthSpeciesSuccMIntervals.update(home, s.fifthSpeciesSuccMIntervals);
789     fifthSpeciesMIntervals.update(home, s.fifthSpeciesMIntervals);
790     fifthSpeciesMTAIntervals.update(home, s.fifthSpeciesMTAIntervals);
791     fifthSpeciesM2Intervals.update(home, s.fifthSpeciesM2Intervals);
792     fifthSpeciesMAllIntervals.update(home, s.fifthSpeciesMAllIntervals);
793     fifthSpeciesMotions.update(home, s.fifthSpeciesMotions);
794     fifthSpeciesMotionsCosts.update(home, s.fifthSpeciesMotionsCosts);
795     isMostlyThirdArray.update(home, s.isMostlyThirdArray);
796     isFourthSpeciesArray.update(home, s.isFourthSpeciesArray);
797     isThirdSpeciesArray.update(home, s.isThirdSpeciesArray);
798     isNotCambiata.update(home, s.isNotCambiata);
799     m2ZeroCostArray.update(home, s.m2ZeroCostArray);
800     thirdHTriadArray.update(home, s.thirdHTriadArray);
801 }
802
803 FifthSpeciesCounterpoint* FifthSpeciesCounterpoint::clone(Home home){
804     return new FifthSpeciesCounterpoint(home, *this);
805 }
806
807 IntVarArray FifthSpeciesCounterpoint::getBranchingNotes(){
```

```
808     return fifthSpeciesNotesCp;
809 }
810 
811 IntVarArray FifthSpeciesCounterpoint::getFirstHInterval(){
812     return firstSpeciesHarmonicIntervals;
813 }
814 
815 IntVarArray FifthSpeciesCounterpoint::getMotions(){
816     return fifthSpeciesMotions;
817 }
818 
819 IntVarArray FifthSpeciesCounterpoint::getFirstMInterval(){
820     return firstSpeciesMelodicIntervals;
821 }
822 
823 int FifthSpeciesCounterpoint::getHIntervalSize(){
824     return fifthSpeciesHIntervals.size();
825 }
826 
827 void FifthSpeciesCounterpoint::createSpeciesArrays(Home home){
828     IntVarArray countThird = IntVarArray(home, solutionLength, 0, 1);
829     IntVarArray countFourth = IntVarArray(home, solutionLength, 0, 1);
830     int nThirdInt = floor((prefSlider-1)*0.66);
831     int nFourthInt = floor(prefSlider*0.5);
832     IntVar sumThird = IntVar(home, nThirdInt, solutionLength);
833     IntVar sumFourth = IntVar(home, nFourthInt, solutionLength);
834 
835     //Count 3rd and 4th species
836     for(int i = 0; i < solutionLength; i++){
837         BoolVar b = BoolVar(home, 0, 1);
838         rel(home, speciesArray[i], IRT_EQ, THIRD_SPECIES, Reify(b));
839         ite(home, b, IntVar(home, 1, 1), IntVar(home, 0, 0), countThird[i]);
840     }
841 
842     for(int i = 0; i < solutionLength; i++){
843         BoolVar b = BoolVar(home, 0, 1);
844         rel(home, speciesArray[i], IRT_EQ, FOURTH_SPECIES, Reify(b));
845         ite(home, b, IntVar(home, 1, 1), IntVar(home, 0, 0), countFourth[i]);
846     }
847 
848     //sum the counts
849     //rel(home, sumThird, IRT_EQ, expr(home, sum(countThird)));
850     //rel(home, sumFourth, IRT_EQ, expr(home, sum(countFourth)));
851 
852     //no 4th species in second and fourth positions in the species array
853     for(int i = 1; i < solutionLength; i+=2){
854         rel(home, speciesArray[i], IRT_NQ, FOURTH_SPECIES);
855     }
856 
857     //4th species in the third position is followed by no species (no note /
        ↪  constraint) and then a 4th species
858     for(int i = 0; i < solutionLength-1; i+=4){
859         BoolVar b34 = BoolVar(home, 0, 1);
860         BoolVar b14 = BoolVar(home, 0, 1);
861 
862         rel(home, speciesArray[i+2], IRT_EQ, FOURTH_SPECIES, Reify(b34));
863         //the following line works to allow quarter notes after fourth species,
            ↪  but behaves weird
864         //rel(home, expr(home, speciesArray[i+4]==FOURTH_SPECIES), BOT_OR, expr(
            ↪  home, speciesArray[i+4]==THIRD_SPECIES), b14);
865         rel(home, speciesArray[i+4], IRT_EQ, FOURTH_SPECIES, Reify(b14));
866         rel(home, speciesArray[i+3], IRT_EQ, -1, Reify(b34, RM_IMP));
867         rel(home, b34, BOT_EQV, b14, 1);
868     }
869 
870     //only 3rd and 4th species
871     for(int i = 0; i < speciesArray.size(); i++){
872         rel(home, speciesArray[i], IRT_NQ, FIRST_SPECIES);
```

```
873             rel(home, speciesArray[i], IRT_NQ, SECOND_SPECIES);
874         }
875 
876         //first and penultimate measures are 4th species
877         rel(home, speciesArray[0], IRT_EQ, -1);
878         rel(home, speciesArray[1], IRT_EQ, -1);
879         rel(home, speciesArray[2], IRT_EQ, FOURTH_SPECIES);
880 
881         rel(home, speciesArray[solutionLength-5], IRT_EQ, FOURTH_SPECIES);
882         rel(home, speciesArray[solutionLength-4], IRT_EQ, -1);
883         rel(home, speciesArray[solutionLength-3], IRT_EQ, FOURTH_SPECIES);
884 
885         //no silence after third species notes
886         for(int i = 0; i < solutionLength-1; i++){
887             BoolVar b = BoolVar(home, 0, 1);
888             rel(home, speciesArray[i], IRT_EQ, THIRD_SPECIES, Reify(b));
889             rel(home, speciesArray[i+1], IRT_NQ, -1, Reify(b, RM_IMP));
890         }
891 
892         //no silence after fourth species notes
893         for(int i = 0; i < solutionLength-5; i++){
894             BoolVar b = BoolVar(home, 0, 1);
895             rel(home, speciesArray[i], IRT_EQ, FOURTH_SPECIES, Reify(b));
896             rel(home, speciesArray[i+4], IRT_NQ, -1, Reify(b, RM_IMP));
897         }
898 
899         //maximum two consecutive measures without a fourth species
900         for(int i = 2; i < solutionLength-13; i+=4){
901             BoolVar b1n4 = BoolVar(home, 0, 1);
902             BoolVar b2n4 = BoolVar(home, 0, 1);
903             BoolVar band = BoolVar(home, 0, 1);
904 
905             rel(home, speciesArray[i+4], IRT_NQ, FOURTH_SPECIES, Reify(b1n4));
906             rel(home, speciesArray[i+8], IRT_NQ, FOURTH_SPECIES, Reify(b2n4));
907             rel(home, b1n4, BOT_AND, b2n4, band);
908             rel(home, speciesArray[i+12], IRT_EQ, FOURTH_SPECIES, Reify(band, RM_IMP)
                  ↪ );
909         }
910 
911         /**
912          * CREATE NTH SPECIES ARRAY
913          */
914         // cout << speciesArray.size() << endl;
915         // cout << isNthSpeciesArray.size() << endl;
916         for(int i = 0; i < isNthSpeciesArray.size(); i+=5){
917             rel(home, speciesArray[floor(i/5)], IRT_EQ, -1, Reify(isNthSpeciesArray[i
                  ↪ ]));
918             rel(home, speciesArray[floor(i/5)], IRT_EQ, FIRST_SPECIES, Reify(
                  ↪ isNthSpeciesArray[i+1]));
919             rel(home, speciesArray[floor(i/5)], IRT_EQ, SECOND_SPECIES, Reify(
                  ↪ isNthSpeciesArray[i+2]));
920             rel(home, speciesArray[floor(i/5)], IRT_EQ, THIRD_SPECIES, Reify(
                  ↪ isNthSpeciesArray[i+3]));
921             rel(home, speciesArray[floor(i/5)], IRT_EQ, FOURTH_SPECIES, Reify(
                  ↪ isNthSpeciesArray[i+4]));
922         }
923 
924         /**
925          * CREATE IS CONSTRAINED ARRAY
926          */
927         for(int i = 0; i < isConstrainedArray.size(); i++){
928             rel(home, isNthSpeciesArray[i*5], IRT_EQ, 0, Reify(isConstrainedArray[i])
                  ↪ );
929         }
930 }
```

FourthSpeciesCounterpoint.cpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../../headers/Parts/FourthSpeciesCounterpoint.hpp"

/**
 * GENERAL CONSTRUCTOR
 */
FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species mSpecies, Stratum* low, CantusFirmus* c,
    int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
    Part(home, nMes, mSpecies, cf, lb, ub, v_type, m_costs, g_costs, s_costs, nV, bm)
{

    for(int i = lowerBound; i <= upperBound; i++){
        cp_range.push_back(i);
    }

    /*
    if borrowMode is enabled, the extended domain is extended to make the inclusion of borrowed notes possible. We can see from Fux's examples
    that he does like to borrow notes, so the borrow cost should just do the job and still allow borrowed notes, not outright forbid them
    */
    if(borrowMode==1){
        extended_domain = vector_union(cp_range, vector_union(scale, borrowed_scale));
    } else {
        extended_domain = vector_intersection(cp_range, vector_union(scale, borrowed_scale));
    }

    off_domain = vector_difference(vector_intersection(cp_range, scale), lowerBound, upperBound);

    fourthSpeciesNotesCp = IntVarArray(home, ((nMeasures*notesPerMeasure.at(FOURTH_SPECIES))-1)-1, IntSet(IntArgs(vector_intersection(cp_range, extended_domain))));
    if(borrowMode==1){
        fourthSpeciesNotesCp[fourthSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(vector_intersection(cp_range, chromatic_scale))));
    }

    sol = IntVarArray(home, fourthSpeciesNotesCp.slice(0,1,fourthSpeciesNotesCp.size()));

    //Harmonic intervals
    for (int i = 0; i < h_intervals.size(); i++) {
        rel(home, (h_intervals[i])==(notes[i]-low->getNotes()[i])%12);
    }

    fourthSpeciesHIntervals = IntVarArray(home, ((nMeasures*notesPerMeasure.at(FOURTH_SPECIES))-1)-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);

    for(int i = 0; i < fourthSpeciesHIntervals.size(); i++){
        rel(home, (fourthSpeciesHIntervals[i])==((fourthSpeciesNotesCp[i]-low->getNotes()[floor((i+1)/2)*4])%12));
    }


    firstHInterval = IntVarArray(home, fourthSpeciesHIntervals.slice(1, 2, fourthSpeciesHIntervals.size()));
```

```
51
52      //m-intervals : arsis and next thesis
53      //m-succ-intervals : thesis and arsis in same measure
54      //m2-intervals : between n and n+2 for whole counterpoint
55      //m-all-intervals : between all notes
56
57      fourthSpeciesMelodicIntervals = IntVarArray(home, fourthSpeciesNotesCp.size()
          ↪  -1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
58      for(int i = 0; i < fourthSpeciesMelodicIntervals.size(); i++){
59          rel(home, fourthSpeciesMelodicIntervals[i] == expr(home,
              ↪  fourthSpeciesNotesCp[i+1] - fourthSpeciesNotesCp[i]));
60      }
61
62      m2IntervalsArray = IntVarArray(home, fourthSpeciesNotesCp.size()-2, -
          ↪  PERFECT_OCTAVE, PERFECT_OCTAVE);
63      for(int i = 0; i < fourthSpeciesNotesCp.size()-2; i++){
64          rel(home, m2IntervalsArray[i], IRT_EQ, expr(home, fourthSpeciesNotesCp[i
              ↪  +2]-fourthSpeciesNotesCp[i]));
65      }
66
67      //create motions array
68      // Arsis motions for the fourth species correspond to the first species
          ↪  motions
69      firstSpeciesMotions = IntVarArray(home, nMeasures-1, IntSet{-1,
          ↪  CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION});
70      firstSpeciesMotionCosts = IntVarArray(home, firstSpeciesMotions.size(),
          ↪  IntSet{0, directCost, obliqueCost, contraryCost});
71
72      //create motions
73      for(int i = 0; i < firstSpeciesMotions.size(); i++){
74          // cout << "i: " << i << endl;
75          int j = 1;
76          if (i == firstSpeciesMotions.size()-1) {
77              j = 0; // last motion is between the arsis of penultimate measure and
                  ↪   the thesis of the last measure
78          }
79
80          //direct motions help creation
81          BoolVar both_up = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]>0)&&(
              ↪  low->getMelodicIntervals()[i]>0)); //if both parts are going in
              ↪  the same direction
82          BoolVar both_stay = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]==0)
              ↪  &&(low->getMelodicIntervals()[i]==0)); //if both parts are staying
83          BoolVar both_down = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]<0)
              ↪  &&(low->getMelodicIntervals()[i]<0)); //if both parts are going
84          //oblique motions help creation
85          BoolVar cf_stays_1 = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]>0)
              ↪  &&(low->getMelodicIntervals()[i]==0)); //if the lowest part stays
              ↪  and one goes up
86          BoolVar cf_stays_2 = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]<0)
              ↪  &&(low->getMelodicIntervals()[i]==0)); //if the lowest part stays
              ↪  and one goes down
87          BoolVar cp_stays_1 = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]==0)
              ↪  &&(low->getMelodicIntervals()[i]>0)); //if the lowest part goes up
              ↪   and one stays
88          BoolVar cp_stays_2 = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]==0)
              ↪  &&(low->getMelodicIntervals()[i]<0)); //if the lowest part goes
              ↪  down and one stays
89          //contrary motions help creation
90          BoolVar cpd_cfu = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]<0)&&(
              ↪  low->getMelodicIntervals()[i]>0)); //if the cf goes up and the cp
              ↪  down
91          BoolVar cpu_cfd = expr(home, (fourthSpeciesMelodicIntervals[i*2+j]>0)&&(
              ↪  low->getMelodicIntervals()[i]<0)); //if the cf goes down and the
              ↪  cp up
92
93          //direct constraints
```

```
94          rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i
              ↪ ]==1)) >> (firstSpeciesMotions[i]==PARALLEL_MOTION));
95          rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i
              ↪ ]==1)) >> (firstSpeciesMotionCosts[i]==directCost));
96          //oblique constraints
97          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪ this->isNotLowest[i]==1)) >> (firstSpeciesMotions[i]==
              ↪ OBLIQUE_MOTION));
98          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪ this->isNotLowest[i]==1)) >> (firstSpeciesMotionCosts[i]==
              ↪ obliqueCost));
99          //contrary constraints
100         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
              ↪ firstSpeciesMotions[i]==CONTRARY_MOTION));
101         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
              ↪ firstSpeciesMotionCosts[i]==contraryCost));
102         //bass constraints
103         rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotions[i]==-1));
104         rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotionCosts[i]==0));
105     }
106
107     is_off = BoolVarArray(home, notes.size(), 0, 1);
108     for(int i = 0; i < is_off.size(); i++){
109         IntVarArray res = IntVarArray(home, off_domain.size(), 0, 1);
110         IntVar sm = IntVar(home, 0, off_domain.size());
111         for(int l = 0; l < off_domain.size(); l++){
112             BoolVar b1 = BoolVar(home, 0, 1);
113             rel(home, notes[i], IRT_EQ, off_domain[l], Reify(b1));
114             ite(home, b1, IntVar(home, 1, 1), IntVar(home, 0, 0), res[l]);
115         }
116         IntVarArgs x(res.size());
117         for(int t = 0; t < off_domain.size(); t++){
118             x[t] = res[t];
119         }
120         rel(home, sm, IRT_EQ, expr(home, sum(x)));
121         rel(home, sm, IRT_GR, 0, Reify(is_off[i]));
122     }
123
124     //create off_cost array
125     offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
126     //set the cost for borrowing this note
127     for(int i = 0; i < is_off.size(); i++){
128         rel(home, (is_off[i]==0) >> (offCostArray[i]==0));
129         rel(home, (is_off[i]==1) >> (offCostArray[i]==borrowCost));
130     }
131
132     melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({
              ↪ secondCost, thirdCost, fourthCost, tritoneCost, fifthCost,
133         sixthCost, seventhCost, octaveCost}));
134
135     // create pefectConsArray
136     fifthCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_fifthCost
              ↪ }));
137     octaveCostArray = IntVarArray(home, h_intervals.size(), IntSet({0,
              ↪ h_octaveCost}));
138
139     isConsonance = BoolVarArray(home, h_intervals.size(), 0, 1);
140     for(int i = 0; i < isConsonance.size(); i++){
141         rel(home, expr(home, (abs(h_intervals[i])==UNISSON)||(abs(h_intervals[i])
              ↪ ==MINOR_THIRD)||(abs(h_intervals[i])==MAJOR_THIRD)||(abs(
              ↪ h_intervals[i])==PERFECT_FIFTH)||
142           (abs(h_intervals[i])==MINOR_SIXTH)||(abs(h_intervals[i])==MAJOR_SIXTH
                ↪ )||(abs(h_intervals[i])==PERFECT_OCTAVE)), IRT_EQ,
                ↪ isConsonance[i]);
143     }
144
145     isNoSyncopeArray = BoolVarArray(home, nMeasures-1, 0, 1);
146     for(int i = 0; i < isNoSyncopeArray.size(); i++){
```

```
147         rel(home, fourthSpeciesMelodicIntervals[(i*2)], IRT_NQ, 0, Reify(
              ↪ isNoSyncopeArray[i]));
148     }
149
150     // G6 : no chromatic melodies
151     if (activeConstraints[SP4_G6]) {
152         for(int i = 0; i < m_intervals_brut.size()-1; i+=4/notesPerMeasure.at(
              ↪ FOURTH_SPECIES)){
153             rel(home, expr(home, m_intervals_brut[i]==1), BOT_AND, expr(home,
                  ↪ m_intervals_brut[i+1]==1), 0);
154             rel(home, expr(home, m_intervals_brut[i]==-1), BOT_AND, expr(home,
                  ↪ m_intervals_brut[i+1]==-1), 0);
155         }
156     }
157
158     // G7 : melodic intervals should be small (works for 1st, 2nd and 3rd species
          ↪ )
159     if (activeConstraints[SP4_G7]) {
160         int idx = 0;
161         for(int i = 0; i < m_intervals_brut.size(); i+=4/notesPerMeasure.at(
              ↪ FOURTH_SPECIES)){
162             rel(home, (abs(m_intervals_brut[i])<MINOR_THIRD) >> (
                  ↪ melodicDegreeCost[i]==secondCost));
163             rel(home, (abs(m_intervals_brut[i])==MINOR_THIRD || abs(
                  ↪ m_intervals_brut[i])==MAJOR_THIRD) >> (melodicDegreeCost[i]==
                  ↪ thirdCost));
164             rel(home, (abs(m_intervals_brut[i])==PERFECT_FOURTH) >> (
                  ↪ melodicDegreeCost[i]==fourthCost));
165             rel(home, (abs(m_intervals_brut[i])==TRITONE) >> (melodicDegreeCost[i
                  ↪ ]==tritoneCost));
166             rel(home, (abs(m_intervals_brut[i])==PERFECT_FIFTH) >> (
                  ↪ melodicDegreeCost[i]==fifthCost));
167             rel(home, (abs(m_intervals_brut[i])==MINOR_SIXTH || abs(
                  ↪ m_intervals_brut[i])==MAJOR_SIXTH) >> (melodicDegreeCost[i]==
                  ↪ sixthCost));
168             rel(home, (abs(m_intervals_brut[i])==MINOR_SEVENTH || abs(
                  ↪ m_intervals_brut[i])==MAJOR_SEVENTH) >> (melodicDegreeCost[i
                  ↪ ]==seventhCost));
169             rel(home, (abs(m_intervals_brut[i])==PERFECT_OCTAVE) >> (
                  ↪ melodicDegreeCost[i]==octaveCost));
170             idx++;
171         }
172     }
173
174     m2ZeroArray = IntVarArray(home, ((fourthSpeciesNotesCp.size()-1)/2)-2, IntSet
          ↪ ({0, m2ZeroCost}));
175     snycopeCostArray = IntVarArray(home, (fourthSpeciesMelodicIntervals.size())
          ↪ /2, IntSet({0, syncopationCost}));
176
177     //link them
178     rel(home, fourthSpeciesNotesCp, IRT_EQ, notes.slice(2, 4/notesPerMeasure.at(
          ↪ FOURTH_SPECIES), notes.size()));
179     rel(home, fourthSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(2,4/
          ↪ notesPerMeasure.at(FOURTH_SPECIES),m_intervals_brut.size()));
180     // rel(home, fourthSpeciesHIntervals, IRT_EQ, h_intervals.slice(2, 4/
          ↪ notesPerMeasure.at(FOURTH_SPECIES), h_intervals.size()));
181
182     // H6 from Thibault : Imperfect consonances are preferred
183     if (activeConstraints[SP4_1H6]) {
184         for(int i = 0; i < h_intervals.size(); i++){
185
186             rel(home, octaveCostArray[i], IRT_EQ, h_octaveCost, Reify(expr(home,
                  ↪ h_intervals[i]==UNISSON), RM_PMI));
187             rel(home, octaveCostArray[i], IRT_EQ, 0, Reify(expr(home, h_intervals
                  ↪ [i]!=UNISSON), RM_PMI));
188
189             rel(home, fifthCostArray[i], IRT_EQ, h_fifthCost, Reify(expr(home,
                  ↪ h_intervals[i]==PERFECT_FIFTH), RM_PMI));
```

```
            rel(home, fifthCostArray[i], IRT_EQ, 0, Reify(expr(home, h_intervals[
              ↪ i]!=PERFECT_FIFTH), RM_PMI));

        }
    }


    //4.H1 : arsis harmonies must be consonant (could be the reason for 3 / 4
      ↪ voice bugs)
    if (activeConstraints[SP4_4H1]) {
        for(int i = 2; i < h_intervals.size(); i+=4){
            dom(home, expr(home, abs(h_intervals[i])), IntSet(CONSONANCES));
        }
    }

    //4.M1 Arsis half notes should be the same as their next halves in thesis
    if (activeConstraints[SP4_4M1]) {
        for(int i = 0; i < snycopeCostArray.size(); i++){
            rel(home, (isNoSyncopeArray[i]==1) >> (snycopeCostArray[i]==
              ↪ syncopationCost));
            rel(home, (isNoSyncopeArray[i]==0) >> (snycopeCostArray[i]==0));
        }
    }

    //4.M2 notes and two beats further are preferred to be different
    if (activeConstraints[SP4_4M2]) {
        for(int i = 0; i < m2ZeroArray.size(); i++){
            rel(home, (fourthSpeciesNotesCp[(i*2)]==fourthSpeciesNotesCp[(i*2)
              ↪ +4]) >> (m2ZeroArray[i]==m2ZeroCost));
            rel(home, (fourthSpeciesNotesCp[(i*2)]!=fourthSpeciesNotesCp[(i*2)
              ↪ +4]) >> (m2ZeroArray[i]==0));
        }
    }

    //4.P1
    if (activeConstraints[SP4_4P1]) {
        for(int i = 1; i < nMeasures-2; i++){
            rel(home, (isConsonance[(i*4)]==0) >> ((fourthSpeciesMelodicIntervals
              ↪ [(i*2)-1]==-MINOR_SECOND) ||
               (fourthSpeciesMelodicIntervals[(i*2)-1]==-MAJOR_SECOND)));
        }
    }

    //4.P2
    if (activeConstraints[SP4_4P2]) {
        for(int i = 0; i < nMeasures-1; i++){
            BoolVar buni = BoolVar(home, 0, 1);
            BoolVar band = BoolVar(home, 0, 1);
            rel(home, fourthSpeciesHIntervals[(i*2)], IRT_EQ, UNISSON, buni);
            rel(home, expr(home, !c->getIsNotLowest()[i]), BOT_AND, buni, band);
            rel(home, fourthSpeciesHIntervals[i*2], IRT_NQ, 1, Reify(band, RM_IMP
              ↪ ));
            rel(home, fourthSpeciesHIntervals[i*2], IRT_NQ, 2, Reify(band, RM_IMP
              ↪ ));
        }
    }

    //1.M2 Melodic intervals cannot be greater than Minor Sixth (except octave)
    if (activeConstraints[SP4_1M2]) {
        dom(home, fourthSpeciesMelodicIntervals, IntSet({UNISSON, -PERFECT_OCTAVE
          ↪ , -MINOR_SIXTH, -PERFECT_FIFTH, -AUGMENTED_FOURTH, -PERFECT_FOURTH
          ↪ ,
            -MAJOR_THIRD, -MINOR_THIRD, -MAJOR_SECOND, -MINOR_SECOND,
              ↪ PERFECT_OCTAVE, MINOR_SIXTH, PERFECT_FIFTH, AUGMENTED_FOURTH,
              ↪ PERFECT_FOURTH,
            MAJOR_THIRD, MINOR_THIRD, MAJOR_SECOND, MINOR_SECOND}));
    }

```

```
246     //DISABELED
247     // //no chromatic motion between 3 consecutive notes
248     // if (activeConstraints[SP4_U2]) {
249     //     for(int i = 1; i < fourthSpeciesMelodicIntervals.size()-1; i++){
250     //         BoolVar b1 = BoolVar(home, 0, 1);
251     //         BoolVar b2 = BoolVar(home, 0, 1);
252     //         BoolVar b3 = BoolVar(home, 0, 1);
253     //         BoolVar b4 = BoolVar(home, 0, 1);
254 
255     //         rel(home, fourthSpeciesMelodicIntervals[i+1], IRT_EQ, 1, Reify(b1)
          ↪ );
256     //         rel(home, m2IntervalsArray[i], IRT_EQ, 2, Reify(b2));
257     //         rel(home, b1, BOT_AND, b2, 0);
258 
259     //         rel(home, fourthSpeciesMelodicIntervals[i+1], IRT_EQ, -1, Reify(b3
          ↪ ));
260     //         rel(home, m2IntervalsArray[i], IRT_EQ, -2, Reify(b4));
261     //         rel(home, b3, BOT_AND, b4, 0);
262     //     }
263     // }
264 }
265 
266 /**
267  * 2 VOICES CONSTRUCTOR
268  */
269 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<
      ↪ int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type,
        vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV
          ↪ ):
270     FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c,
          ↪ v_type, m_costs, g_costs, s_costs, bm, nV)
271 {
272 
273     //4.H2 : If the 4th species is the lowest stratum, then no hamonic seventh
274     if (activeConstraints[SP4_4H2_2V]) {
275         for(int j = 1; j < getIsNotLowest().size(); j++){
276             rel(home, (getIsNotLowest()[j]==0) >> (fourthSpeciesHIntervals[(j*2)
                  ↪ -1]!=MINOR_SEVENTH && fourthSpeciesHIntervals[(j*2)-1]!=-
                  ↪ MINOR_SEVENTH));
277             rel(home, (getIsNotLowest()[j]==0) >> (fourthSpeciesHIntervals[(j*2)
                  ↪ -1]!=MAJOR_SEVENTH && fourthSpeciesHIntervals[(j*2)-1]!=-
                  ↪ MAJOR_SEVENTH));
278         }
279     }
280 
281     //4.H3 Penult note condition
282     if (activeConstraints[SP4_4H3_2V]) {
283         rel(home,(isNotLowest[isNotLowest.size()-2]==1) >> (expr(home, abs(
              ↪ h_intervals[h_intervals.size()-3]))==MAJOR_SIXTH));
284     }
285 
286     //Must start with a perfect consonance (since 1.H2 applies to the first note,
          ↪  which is most likely a rest in 4th species)
287     if (activeConstraints[SP4_1H2_2V]) {
288         dom(home, fourthSpeciesHIntervals[0], IntSet({UNISSON, PERFECT_FIFTH, -
              ↪ PERFECT_FIFTH}));
289     }
290 
291     costs = IntVarArray(home, 6, 0, 1000000);
292     cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation"};
293 
294     //set cost[0] to be fifth cost
295     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure
          ↪ .at(FOURTH_SPECIES), fifthCostArray.size())), costs);
296     //set cost[1] to be octave cost
297     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/
          ↪ notesPerMeasure.at(FOURTH_SPECIES), octaveCostArray.size())), costs);
298     //set cost[2] to be melodic cost
299 
```

```
    add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/notesPerMeasure.at(FOURTH_SPECIES), melodicDegreeCost.size())), costs);
    //set cost[3] to be off cost
    add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.at(FOURTH_SPECIES), offCostArray.size())), costs);
    //need to set cost[4] to be m2Zero cost
    add_cost(home, 4, m2ZeroArray, costs);
    //need to set cost[5] to be syncopation cost
    add_cost(home, 5, snycopeCostArray, costs);
}

/**
 * 3 VOICES CONSTRUCTOR
 */
FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type,
    vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2):
    FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c, v_type, m_costs, g_costs, s_costs, bm, nV2)
{
    varietyCostArray = IntVarArray(home, 3*(getHIntervalSize()-2), IntSet({0, varietyCost}));

    //4.P5 -- after careful testing, Fux does not seem to follow this rule in many of his examples. Suspended for now, but implementation left in case the decision is taken to reactivate it.
    if (activeConstraints[SP4_4P5_3V]) {
        for(int j = 1; j < nMeasures-1; j++){
            rel(home, (low->getMelodicIntervals()[j]==0)>>(expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SECOND || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SECOND
              || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==PERFECT_FOURTH || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==AUGMENTED_FOURTH ||
              expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SEVENTH || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SEVENTH));
        }
    }


    costs = IntVarArray(home, 7, 0, 1000000);
    cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation", "variety"};

    //set cost[0] to be fifth cost
    add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure.at(FOURTH_SPECIES), fifthCostArray.size())), costs);
    //set cost[1] to be octave cost
    add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/notesPerMeasure.at(FOURTH_SPECIES), octaveCostArray.size())), costs);
    //set cost[2] to be melodic cost
    add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/notesPerMeasure.at(FOURTH_SPECIES), melodicDegreeCost.size())), costs);
    //set cost[3] to be off cost
    add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.at(FOURTH_SPECIES), offCostArray.size())), costs);
    //need to set cost[4] to be m2Zero cost
    add_cost(home, 4, m2ZeroArray, costs);
    //need to set cost[5] to be syncopation cost
    add_cost(home, 5, snycopeCostArray, costs);
    //need to set cost[6] to be variety cost
    add_cost(home, 6, varietyCostArray, costs);
}
```

```
347 /**
348  * 4 VOICES CONSTRUCTOR
349  */
350 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<
      ↪ int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type,
351     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
        ↪ nV1, int nV2, int nV3):
352     FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c,
        ↪ v_type, m_costs, g_costs, s_costs, bm, nV2)
353 {
354     varietyCostArray = IntVarArray(home, 3*(getHIntervalSize()-2), IntSet({0,
        ↪ varietyCost}));
355
356     //4.P5 -- after careful testing, Fux does not seem to follow this rule in
        ↪ many of his examples. Suspended for now, but implementation left in
        ↪ case the decision is taken to reactivate it.
357     if (activeConstraints[SP4_4P5_4V]) {
358         for(int j = 1; j < nMeasures-1; j++){
359             rel(home, (low->getMelodicIntervals()[j]==0)>>(expr(home, abs(
                  ↪ fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SECOND || expr(home,
                  ↪  abs(fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SECOND
360                 || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==
                      ↪ PERFECT_FOURTH || expr(home, abs(fourthSpeciesHIntervals[(
                      ↪ j*2)+1]))==AUGMENTED_FOURTH ||
361                 expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SEVENTH
                      ↪ || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==
                      ↪ MAJOR_SEVENTH));
362         }
363     }
364
365
366     costs = IntVarArray(home, 7, 0, 1000000);
367     cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation", "
        ↪ variety"};
368
369     //set cost[0] to be fifth cost
370     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure
        ↪ .at(FOURTH_SPECIES), fifthCostArray.size())), costs);
371     //set cost[1] to be octave cost
372     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/
        ↪ notesPerMeasure.at(FOURTH_SPECIES), octaveCostArray.size())), costs);
373     //set cost[2] to be melodic cost
374     add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/
        ↪ notesPerMeasure.at(FOURTH_SPECIES), melodicDegreeCost.size())), costs)
        ↪ ;
375     //set cost[3] to be off cost
376     add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.
        ↪ at(FOURTH_SPECIES), offCostArray.size())), costs);
377     //need to set cost[4] to be m2Zero cost
378     add_cost(home, 4, m2ZeroArray, costs);
379     //need to set cost[5] to be syncopation cost
380     add_cost(home, 5, snycopeCostArray, costs);
381     //need to set cost[6] to be variety cost
382     add_cost(home, 6, varietyCostArray, costs);
383 }
384
385 /** j < nMeasures-1; j++){
386     //     rel(home, (low->getMelodicIntervals()[j]==0)>>(expr(home, abs(
        ↪ fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SECOND || expr(home, abs(
        ↪ fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SECOND
387     //         || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==
        ↪ PERFECT_FOURTH || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==
        ↪ AUGMENTED_FOURTH ||
388     //         expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SEVENTH ||
        ↪  expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SEVENTH));
389     // }
390  * This function returns a string with the characteristics of the counterpoint.
      ↪ It calls the to_string() method from
```

```
 * the Part class and adds 1st species specific characteristics.
 * @return a string representation of the current instance of the
    ↪ FirstSpeciesCounterpoint class.
 */
string FourthSpeciesCounterpoint::to_string() const {
    string text = Part::to_string() + "\nFourth species :\n";
    text += "No Syncope array : " + boolVarArray_to_string(isNoSyncopeArray) + "\
      ↪ n";
    text += "Fourth species notes : " + intVarArray_to_string(
      ↪ fourthSpeciesNotesCp) + "\n";
    text += "Fourth species h intervals : " + intVarArray_to_string(
      ↪ fourthSpeciesHIntervals) + "\n";
    text += "Fourth species m intervals : " + intVarArray_to_string(
      ↪ fourthSpeciesMelodicIntervals) + "\n";
    text += "is Consonance array : " + boolVarArray_to_string(isConsonance) + "\n
      ↪ ";
    return text;
}

// clone constructor
FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home,
  ↪ FourthSpeciesCounterpoint &s) : Part(home, s){
    fourthSpeciesNotesCp.update(home, s.fourthSpeciesNotesCp);
    fourthSpeciesHIntervals.update(home, s.fourthSpeciesHIntervals);
    fourthSpeciesMelodicIntervals.update(home, s.fourthSpeciesMelodicIntervals);
    m2IntervalsArray.update(home, s.m2IntervalsArray);
    firstHInterval.update(home, s.firstHInterval);
    m2ZeroArray.update(home, s.m2ZeroArray);
    sol.update(home, s.sol);
}

FourthSpeciesCounterpoint* FourthSpeciesCounterpoint::clone(Home home){
    return new FourthSpeciesCounterpoint(home, *this);
}

IntVarArray FourthSpeciesCounterpoint::getBranchingNotes(){
    return sol;
}

IntVarArray FourthSpeciesCounterpoint::getFirstHInterval(){
    return firstHInterval;
}

IntVarArray FourthSpeciesCounterpoint::getMotions(){
    return motions;
}

IntVarArray FourthSpeciesCounterpoint::getFirstMInterval(){
    return m2IntervalsArray;
}

int FourthSpeciesCounterpoint::getHIntervalSize(){
    return fourthSpeciesHIntervals.size();
}
```

ThirdSpeciesCounterpoint.cpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//

#include "../../headers/Parts/ThirdSpeciesCounterpoint.hpp"

/**
 * Third species general constructor
```

```
 */
ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec, Stratum* low, CantusFirmus* c,
    int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
    FirstSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs, g_costs, s_costs, bm, nV)
{
    thirdSpeciesNotesCp = IntVarArray(home, notes.size(), IntSet(IntArgs(vector_intersection(cp_range, extended_domain))));
    if(borrowMode==1){
        thirdSpeciesNotesCp[thirdSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(vector_intersection(cp_range, chromatic_scale))));
    }
    rel(home, thirdSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(THIRD_SPECIES),(notes.size())));

    thirdSpeciesHarmonicIntervals = IntVarArray(home, h_intervals.size(), -PERFECT_OCTAVE, PERFECT_OCTAVE);
    // rel(home, thirdSpeciesHarmonicIntervals, IRT_EQ, h_intervals);
    // for(int i = 0; i < thirdSpeciesHarmonicIntervals.size(); i++){
    //     rel(home, (thirdSpeciesHarmonicIntervals[i])==((thirdSpeciesNotesCp[i]-low->getNotes()[floor(i/4)*4])%12));
    // }


    thirdSpeciesMelodicIntervals = IntVarArray(home, m_intervals_brut.size(), -MAX_STEP, MAX_STEP);

    for(int i = 0; i < thirdSpeciesMelodicIntervals.size(); i++)
        rel(home, thirdSpeciesMelodicIntervals[i], IRT_EQ, expr(home, thirdSpeciesNotesCp[i+1] - thirdSpeciesNotesCp[i]));

    rel(home, thirdSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure.at(THIRD_SPECIES),m_intervals_brut.size()));

    for(int i = 0; i < thirdSpeciesMelodicIntervals.size(); i++){
        rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i])), IRT_GR, 0);
    }

    thirdSpeciesMotions = IntVarArray(home, notes.size()/4, IntSet({-1, CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION}));
    thirdSpeciesMotionCosts = IntVarArray(home, notes.size()/4, IntSet{0, directCost, obliqueCost, contraryCost});

    for(int i = 0; i < thirdSpeciesMotions.size(); i++){

        //direct motions help creation
        BoolVar both_up = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(low->getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
        BoolVar both_stay = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->getMelodicIntervals()[i]==0)); //if both parts are staying
        BoolVar both_down = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(low->getMelodicIntervals()[i]<0)); //if both parts are going down
        //oblique motions help creation
        BoolVar cf_stays_1 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(low->getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
        BoolVar cf_stays_2 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(low->getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
        BoolVar cp_stays_1 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
        BoolVar cp_stays_2 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->getMelodicIntervals()[i]<0)); //if the lowest part
```

```
            ↪ goes down and one stays
        //contrary motions help creation
        BoolVar cpd_cfu = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(
            ↪ low->getMelodicIntervals()[i]>0)); //if the cf goes up and the cp
            ↪ down
        BoolVar cpu_cfd = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(
            ↪ low->getMelodicIntervals()[i]<0)); //if the cf goes down and the
            ↪ cp up

        //direct constraints
        rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i])) >> (
            ↪ thirdSpeciesMotions[i]==PARALLEL_MOTION));
        rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i])) >> (
            ↪ thirdSpeciesMotionCosts[i]==directCost));
        //oblique constraints
        rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
            ↪ isNotLowest[i])) >> (thirdSpeciesMotions[i]==OBLIQUE_MOTION));
        rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
            ↪ isNotLowest[i])) >> (thirdSpeciesMotionCosts[i]==obliqueCost));
        //contrary constraints
        rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i])) >> (
            ↪ thirdSpeciesMotions[i]==CONTRARY_MOTION));
        rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i])) >> (
            ↪ thirdSpeciesMotionCosts[i]==contraryCost));
        //bass constraints
        rel(home, (!isNotLowest[i]) >> (thirdSpeciesMotions[i]==-1));
        rel(home, (!isNotLowest[i]) >> (thirdSpeciesMotionCosts[i]==0));
    }

    thirdSpeciesAbsMelodic = IntVarArray(home, thirdSpeciesMelodicIntervals.size
        ↪ (), UNISSON, PERFECT_OCTAVE);
    for(int i = 0; i < thirdSpeciesAbsMelodic.size(); i++){
        rel(home, thirdSpeciesAbsMelodic[i] == abs(thirdSpeciesMelodicIntervals[i
            ↪ ]));
    }

    m2IntervalsArray = IntVarArray(home, thirdSpeciesNotesCp.size()-2, -MAX_STEP,
        ↪  MAX_STEP);
    for(int i = 0; i < thirdSpeciesNotesCp.size()-2; i++){
        //size-2 of notes (butlast butlast) ; from 2 onwards in notes (rest rest)
        rel(home, m2IntervalsArray[i], IRT_EQ, expr(home, thirdSpeciesNotesCp[i
            ↪ +2]-thirdSpeciesNotesCp[i]));
    }

    is5QNArray = BoolVarArray(home, nMeasures-1, 0, 1);
    for(int i = 0; i < is5QNArray.size()*4; i+=4){

        BoolVar b1 = BoolVar(home, 0, 1);
        BoolVar b2 = BoolVar(home, 0, 1);
        BoolVar b3 = BoolVar(home, 0, 1);
        BoolVar b4 = BoolVar(home, 0, 1);
        BoolVar bb1 = BoolVar(home, 0, 1);
        BoolVar bb2 = BoolVar(home, 0, 1);
        BoolVar bb3 = BoolVar(home, 0, 1);
        BoolVar bb4 = BoolVar(home, 0, 1);
        BoolVar band1 = BoolVar(home, 0, 1);
        BoolVar band2 = BoolVar(home, 0, 1);
        BoolVar band3 = BoolVar(home, 0, 1);
        BoolVar beq1 = BoolVar(home, 0, 1);
        BoolVar beq2 = BoolVar(home, 0, 1);
        BoolVar beq3 = BoolVar(home, 0, 1);

        rel(home, thirdSpeciesAbsMelodic[i], IRT_LQ, 2, Reify(b1));
        rel(home, thirdSpeciesAbsMelodic[i+1], IRT_LQ, 2, Reify(b2));
        rel(home, thirdSpeciesAbsMelodic[i+2], IRT_LQ, 2, Reify(b3));
        rel(home, thirdSpeciesAbsMelodic[i+3], IRT_LQ, 2, Reify(b4));

        rel(home, thirdSpeciesMelodicIntervals[i], IRT_GQ, 0, Reify(bb1));
```

```
105         rel(home, thirdSpeciesMelodicIntervals[i+1], IRT_GQ, 0, Reify(bb2));
106         rel(home, thirdSpeciesMelodicIntervals[i+2], IRT_GQ, 0, Reify(bb3));
107         rel(home, thirdSpeciesMelodicIntervals[i+3], IRT_GQ, 0, Reify(bb4));
108 
109         rel(home, b1, BOT_AND, b2, band1);
110         rel(home, b3, BOT_AND, b4, band2);
111         rel(home, band1, BOT_AND, band2, band3);
112 
113         rel(home, bb1, BOT_EQV, bb2, beq1);
114         rel(home, bb3, BOT_EQV, bb4, beq2);
115         rel(home, beq1, BOT_EQV, beq2, beq3);
116         rel(home, band3, BOT_AND, beq3, is5QNArray[i/4]);
117     }
118 
119     isDiminution = BoolVarArray(home, (nMeasures* notesPerMeasure.at(
        ↪ FIRST_SPECIES) -1), 0, 1);
120     for(int i = 0; i < isDiminution.size()*4; i+=4){
121 
122         BoolVar btt3 = BoolVar(home, 0, 1);
123         BoolVar btt4 = BoolVar(home, 0, 1);
124         BoolVar bta2nd = BoolVar(home, 0, 1);
125         BoolVar btt3rd = BoolVar(home, 0, 1);
126         BoolVar bat2nd = BoolVar(home, 0, 1);
127         BoolVar band = BoolVar(home, 0, 1);
128         IntVar addition = expr(home, abs(m2IntervalsArray[i+1]));
129         rel(home, addition, IRT_EQ, 3, Reify(btt3));
130         rel(home, addition, IRT_EQ, 4, Reify(btt4));
131         rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i+1])), IRT_LQ, 2,
            ↪ Reify(bta2nd));
132         rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i+2])), IRT_LQ, 2,
            ↪ Reify(bat2nd));
133         rel(home, btt3, BOT_OR, btt4, btt3rd);
134         rel(home, bta2nd, BOT_AND, btt3rd, band);
135         rel(home, band, BOT_AND, bat2nd, isDiminution[i/4]);
136 
137     }
138 
139     cambiataCostArray = IntVarArray(home, nMeasures-1, IntSet({0, cambiataCost}))
        ↪ ;
140 
141     m2ZeroArray = IntVarArray(home, thirdSpeciesMelodicIntervals.size()-2, IntSet
        ↪ ({0, m2ZeroCost}));
142 
143     //DISABELED
144     // //3.H1 : five consecutive notes by joint degree implies that the first and
        ↪  the third note are consonants
145     // if (activeConstraints[SP3_3H1]) {
146     //     H1_3_fiveConsecutiveNotesByJointDegree(home, this);
147     // }
148 
149     //3.H2 : any dissonant note implies that it is surrounded by consonant notes
150     if (activeConstraints[SP3_3H2]) {
151         H2_3_disonanceImpliesDiminution(home, this);
152     }
153     //3.H3 : cambiata cost
154     if (activeConstraints[SP3_3H3]) {
155         H3_3_cambiataCost(home, this);
156     }
157     //3.M1 : each note and its two beats further peer are preferred to be
        ↪ different
158     //i + i+1 + i+2
159     if (activeConstraints[SP3_3M1]) {
160         for(int i = 0; i < m2ZeroArray.size(); i++){
161             rel(home, ((thirdSpeciesMelodicIntervals[i]+
                ↪ thirdSpeciesMelodicIntervals[i+1]+thirdSpeciesMelodicIntervals
                ↪ [i+2])==0) >> (m2ZeroArray[i]==m2ZeroCost));
162             rel(home, ((thirdSpeciesMelodicIntervals[i]+
                ↪ thirdSpeciesMelodicIntervals[i+1]+thirdSpeciesMelodicIntervals
```

```
                ↪  [i+2])!=0) >> (m2ZeroArray[i]==0));
163         }
164     }
165
166     //marcel's rule
167     /*for(int i = 0; i < thirdSpeciesMelodicIntervals.size()-1; i++){
168         BoolVar bSkip = BoolVar(home, 0, 1);
169         BoolVar bMbUp = BoolVar(home, 0, 1);
170         BoolVar bMbDown = BoolVar(home, 0, 1);
171         BoolVar bContrary = BoolVar(home, 0, 1);
172
173         rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i])), IRT_GR, 2,
              ↪  Reify(bSkip));
174         rel(home, thirdSpeciesMelodicIntervals[i], IRT_GR, 0, Reify(bMbUp));
175         rel(home, thirdSpeciesMelodicIntervals[i+1], IRT_LE, 1, Reify(bMbDown));
176         rel(home, bMbUp, BOT_EQV, bMbDown, bContrary);
177         rel(home, thirdSpeciesMelodicIntervals[i+1], IRT_LQ, 2, Reify(bSkip,
              ↪  RM_IMP));
178         rel(home, bSkip, BOT_IMP, bContrary, 1);
179     }*/
180
181     //no battuta adapted for third species (1.P3 ? Tom Lai)
182     if (activeConstraints[SP3_1P3]) {
183         for(int j = 0; j < thirdSpeciesMotions.size(); j++){
184             rel(home, expr(home, thirdSpeciesMotions[j]==CONTRARY_MOTION &&
                    ↪  firstSpeciesHarmonicIntervals[j+1]==0 &&
                    ↪  thirdSpeciesMelodicIntervals[(j*4)+3]<-4),
185                 BOT_AND, isNotLowest[j], 0);
186         }
187     }
188
189     // 3.M2
190     //no melodic interval between 9 and 11
191     if (activeConstraints[SP3_U1]) {
192         for(int i = 0; i < nMeasures-1; i++){
193             for (size_t j = 0; j < 4; j++)
194             {
195                 rel(home, abs(thirdSpeciesMelodicIntervals[(i*4)+j])!=MAJOR_SIXTH
                        ↪   && abs(thirdSpeciesMelodicIntervals[(i*4)+j])!=
                        ↪  MINOR_SEVENTH && abs(thirdSpeciesMelodicIntervals[(i*4)+j
                        ↪  ])!=MAJOR_SEVENTH);
196             }
197
198
199         }
200     }
201
202     // REMOVED UNDOCUMENTED RULE
203     // //third note of the penultimate measure must be below the fourth one
204     // if (activeConstraints[SP3_U2]) {
205     //     BoolVar isLowest(home, 0, 1);
206     //     rel(home, isNotLowest[isNotLowest.size()-2], IRT_EQ, 0, Reify(isLowest
          ↪  ));
207     //     //third note of the penultimate measure must be more distant than a
          ↪  semi tone from the last note of the penultimate measure
208     //     rel(home, (!isLowest) >> (thirdSpeciesMelodicIntervals[
          ↪  thirdSpeciesMelodicIntervals.size()-2] > 0));
209     // }
210
211     // 3.M4
212     //second one must also be more distant than a semi tone from the last note of
          ↪   the penultimate measure
213     if (activeConstraints[SP3_U3]) {
214         rel(home, expr(home, thirdSpeciesMelodicIntervals[
                ↪  thirdSpeciesMelodicIntervals.size()-3]+
                ↪  thirdSpeciesMelodicIntervals[thirdSpeciesMelodicIntervals.size()
                ↪  -2])
215         , IRT_NQ, 1);
```

```
216     }
217
218     // combined costs
219     toCombineCosts = IntVarArray(home, 1, 0, 10000);
220     toCombineCostNames = {"1H1"};
221     //need to set constraintCosts[0]
222     add_toCombineCost(home, 0, disArray, toCombineCosts);
223 }
224
225 /**
226  * 2 VOICES CONSTRUCTOR
227  */
228 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<
      ↪ int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
229     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV
        ↪ ):
230     ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c,
        ↪ v_type, m_costs,g_costs, s_costs, bm, nV)
231 {
232     //3.H4 : in the penultimate measure, if the cantusFirmus is in the upper part
        ↪ , then the h_interval of the first note should be a minor third
233     // REPLACED in twoVoiceCounterpoint.cpp
234     // if (activeConstraints[SP3_3H4_2V]) {
235     //     rel(home, (getIsNotLowest()[getIsNotLowest().size()-2]==0) >>
236     //     (expr(home, abs(h_intervals[h_intervals.size()-5]))==MINOR_THIRD));
237     // }
238     costs = IntVarArray(home, 7, 0, 10000);
239     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "cambiata", "
        ↪ m2"};
240
241     //set cost[0] to be fifth cost
242     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
        ↪ .at(THIRD_SPECIES), fifthCostArray.size())), costs);
243     //set cost[1] to be octave cost
244     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/
        ↪ notesPerMeasure.at(THIRD_SPECIES), octaveCostArray.size())), costs);
245     //set cost[2] to be motion cost
246     add_cost(home, 2, thirdSpeciesMotionCosts, costs);
247     //set cost[3] to be melodic cost
248     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/
        ↪ notesPerMeasure.at(THIRD_SPECIES), melodicDegreeCost.size())), costs);
249     //need to set cost[4] to be off cost
250     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
        ↪ at(THIRD_SPECIES), offCostArray.size())), costs);
251     //need to set cost[5] to be cambiata cost
252     add_cost(home, 5, cambiataCostArray, costs);
253     //need to set cost[5] to be cambiata cost
254     add_cost(home, 6, m2ZeroArray, costs);
255 }
256
257 /**
258  * 3 VOICES CONSTRUCTOR
259  */
260 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<
      ↪ int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
261     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
        ↪ nV1, int nV2):
262     ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c,
        ↪ v_type, m_costs,g_costs, s_costs, bm, nV2)
263 {
264     varietyCostArray = IntVarArray(home, 3*(thirdSpeciesHarmonicIntervals.size()
        ↪ -2), IntSet({0, varietyCost}));
265     directCostArray = IntVarArray(home, thirdSpeciesMotions.size()-1,IntSet({0,
        ↪ directMoveCost}));
266
267     //1.H7,H8 adapted
268     if (activeConstraints[SP3_1H7_3V]) {
```

```
269         rel(home, thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.
              ↪ size()-1]==UNISSON||thirdSpeciesHarmonicIntervals[
              ↪ thirdSpeciesHarmonicIntervals.size()-1]==MINOR_THIRD||
              ↪ thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()
              ↪ -1]==PERFECT_FIFTH||
270         thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==
              ↪ MAJOR_SIXTH);
271
272     }
273
274     //3.H6 : harmonic triad should be used on the second or third beat
275     if (activeConstraints[SP3_3H6_3V]) {
276         thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost
              ↪ }));
277         for(int i = 0; i < thirdHTriadArray.size(); i++){
278             rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]!=UNISSON&&
                  ↪ thirdSpeciesHarmonicIntervals[(i*4)+1]!=MINOR_THIRD&&
                  ↪ thirdSpeciesHarmonicIntervals[(i*4)+1]!=MAJOR_THIRD&&
                  ↪ thirdSpeciesHarmonicIntervals[(i*4)+1]!=PERFECT_FIFTH)&&
279                 (thirdSpeciesHarmonicIntervals[(i*4)+2]!=UNISSON&&
                      ↪ thirdSpeciesHarmonicIntervals[(i*4)+2]!=MINOR_THIRD&&
                      ↪ thirdSpeciesHarmonicIntervals[(i*4)+2]!=MAJOR_THIRD&&
                      ↪ thirdSpeciesHarmonicIntervals[(i*4)+2]!=PERFECT_FIFTH)) >>
280                 (thirdHTriadArray[i]==triad3rdCost));
281             rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]==UNISSON||
                  ↪ thirdSpeciesHarmonicIntervals[(i*4)+1]==MINOR_THIRD||
                  ↪ thirdSpeciesHarmonicIntervals[(i*4)+1]==MAJOR_THIRD||
                  ↪ thirdSpeciesHarmonicIntervals[(i*4)+1]==PERFECT_FIFTH)||
282                 (thirdSpeciesHarmonicIntervals[(i*4)+2]==UNISSON||
                      ↪ thirdSpeciesHarmonicIntervals[(i*4)+2]==MINOR_THIRD||
                      ↪ thirdSpeciesHarmonicIntervals[(i*4)+2]==MAJOR_THIRD||
                      ↪ thirdSpeciesHarmonicIntervals[(i*4)+2]==PERFECT_FIFTH)) >>
283                 (thirdHTriadArray[i]==0));
284         }
285     }
286
287     //1.P1 3 voices version
288     if (activeConstraints[SP3_1P1_3V]) {
289         for(int j = 0; j < firstSpeciesMotions.size()-1; j++){
290             //set a cost when it is reached through direct motion, it is 0 when
                  ↪ not
291             rel(home, (thirdSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[
                  ↪ j+1]==0||firstSpeciesHarmonicIntervals[j+1]==7))>>
292                 (directCostArray[j]==directMoveCost));
293             rel(home, (thirdSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[
                  ↪ j+1]!=0&&firstSpeciesHarmonicIntervals[j+1]!=7))>>
294                 (directCostArray[j]==0));
295         }
296     }
297
298
299     costs = IntVarArray(home, 10, 0, 10000);
300     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "
          ↪ direct", "cambiata", "m2", "triad3"};
301     //need to set cost[0] to be off cost
302     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
          ↪ at(THIRD_SPECIES), offCostArray.size())), costs);
303     //set cost[1] to be fifth cost
304     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
          ↪ .at(THIRD_SPECIES), fifthCostArray.size())), costs);
305     //set cost[2] to be octave cost
306     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/
          ↪ notesPerMeasure.at(THIRD_SPECIES), octaveCostArray.size())), costs);
307     //need to set cost[3] to be variety cost
308     add_cost(home, 3, varietyCostArray, costs);
309     //set cost[4] to be motion cost
310     add_cost(home, 4, thirdSpeciesMotionCosts, costs);
311     //set cost[5] to be melodic cost
```

```
312     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪  notesPerMeasure.at(THIRD_SPECIES), melodicDegreeCost.size())), costs);
313     //need to set cost[6] to be direct cost
314     add_cost(home, 6, directCostArray, costs);
315     //set cost[7] to be cambiata sixth cost
316     add_cost(home, 7, cambiataCostArray, costs);
317     //set cost[8] to be m2 zero cost
318     add_cost(home, 8, m2ZeroArray, costs);
319     //set cost[9] to be triad h third cost
320     add_cost(home, 9, thirdHTriadArray, costs);
321 }
322
323 /**
324  * 4 VOICES CONSTRUCTOR
325  */
326 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<
      ↪  int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
327     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
          ↪  nV1, int nV2, int nV3):
328     ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c,
          ↪  v_type, m_costs,g_costs, s_costs, bm, nV3)
329 {
330     varietyCostArray = IntVarArray(home, 3*(thirdSpeciesHarmonicIntervals.size()
          ↪  -2), IntSet({0, varietyCost}));
331     directCostArray = IntVarArray(home, thirdSpeciesMotions.size()-1,IntSet({0,
          ↪  2, directMoveCost}));
332
333     //1.H7,H8 adapted
334     if (activeConstraints[SP3_1H7_4V]) {
335         rel(home, thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.
              ↪  size()-1]==UNISSON||thirdSpeciesHarmonicIntervals[
              ↪  thirdSpeciesHarmonicIntervals.size()-1]==MINOR_THIRD||
              ↪  thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()
              ↪  -1]==PERFECT_FIFTH||
336         thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==
              ↪  MAJOR_SIXTH);
337     }
338
339     //3.H6 : harmonic triad should be used on the second or third beat
340     if (activeConstraints[SP3_3H6_4V]) {
341         thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost
              ↪  }));
342         for(int i = 0; i < thirdHTriadArray.size(); i++){
343             rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]!=UNISSON&&
                  ↪  thirdSpeciesHarmonicIntervals[(i*4)+1]!=MINOR_THIRD&&
                  ↪  thirdSpeciesHarmonicIntervals[(i*4)+1]!=MAJOR_THIRD&&
                  ↪  thirdSpeciesHarmonicIntervals[(i*4)+1]!=PERFECT_FIFTH)&&
344                 (thirdSpeciesHarmonicIntervals[(i*4)+2]!=UNISSON&&
                      ↪  thirdSpeciesHarmonicIntervals[(i*4)+2]!=MINOR_THIRD&&
                      ↪  thirdSpeciesHarmonicIntervals[(i*4)+2]!=MAJOR_THIRD&&
                      ↪  thirdSpeciesHarmonicIntervals[(i*4)+2]!=PERFECT_FIFTH)) >>
345                 (thirdHTriadArray[i]==triad3rdCost));
346             rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]==UNISSON||
                  ↪  thirdSpeciesHarmonicIntervals[(i*4)+1]==MINOR_THIRD||
                  ↪  thirdSpeciesHarmonicIntervals[(i*4)+1]==MAJOR_THIRD||
                  ↪  thirdSpeciesHarmonicIntervals[(i*4)+1]==PERFECT_FIFTH)||
347                 (thirdSpeciesHarmonicIntervals[(i*4)+2]==UNISSON||
                      ↪  thirdSpeciesHarmonicIntervals[(i*4)+2]==MINOR_THIRD||
                      ↪  thirdSpeciesHarmonicIntervals[(i*4)+2]==MAJOR_THIRD||
                      ↪  thirdSpeciesHarmonicIntervals[(i*4)+2]==PERFECT_FIFTH)) >>
348                 (thirdHTriadArray[i]==0));
349         }
350     }
351
352     //1.P1 4 voices version
353     if (activeConstraints[SP3_1P1_4V]) {
354         for(int j = 0; j < firstSpeciesMotions.size()-1; j++){
```

```
355             //set a cost when it is reached through direct motion, it is 0 when not
356             rel(home, (thirdSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||firstSpeciesHarmonicIntervals[j+1]==7))>>
357               (directCostArray[j]==directMoveCost));
358             rel(home, (thirdSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&firstSpeciesHarmonicIntervals[j+1]!=7))>>
359               (directCostArray[j]==0));
360         }
361     }
362
363
364     costs = IntVarArray(home, 10, 0, 10000);
365     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct", "cambiata", "m2", "triad3"};
366     //need to set cost[0] to be off cost
367     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(THIRD_SPECIES), offCostArray.size())), costs);
368     //set cost[1] to be fifth cost
369     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(THIRD_SPECIES), fifthCostArray.size())), costs);
370     //set cost[2] to be octave cost
371     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(THIRD_SPECIES), octaveCostArray.size())), costs);
372     //need to set cost[3] to be variety cost
373     add_cost(home, 3, varietyCostArray, costs);
374     //set cost[4] to be motion cost
375     add_cost(home, 4, thirdSpeciesMotionCosts, costs);
376     //set cost[5] to be melodic cost
377     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(THIRD_SPECIES), melodicDegreeCost.size())), costs);
378     //need to set cost[6] to be direct cost
379     add_cost(home, 6, directCostArray, costs);
380     //set cost[7] to be cambiata sixth cost
381     add_cost(home, 7, cambiataCostArray, costs);
382     //set cost[8] to be m2 zero cost
383     add_cost(home, 8, m2ZeroArray, costs);
384     //set cost[9] to be triad h third cost
385     add_cost(home, 9, thirdHTriadArray, costs);
386 }
387
388 string ThirdSpeciesCounterpoint::to_string() const {
389     string text = Part::to_string() + "\nThird species isConsonance intervals : \n";
390     text += boolVarArray_to_string(isConsonance) += "\n";
391     text += "Third species is Lowest : \n";
392     text += boolVarArray_to_string(isNotLowest) += "\n";
393     text += "Third species motions : \n";
394     text += intVarArray_to_string(thirdSpeciesMotions) += "\n";
395     text += "Third species m intervals : \n";
396     text += intVarArray_to_string(thirdSpeciesMelodicIntervals) += "\n";
397     text += "m intervals : \n";
398     text += intVarArray_to_string(m_intervals_brut) += "\n";
399     text += "Third species is5QN array : \n";
400     text += boolVarArray_to_string(is5QNArray) += "\n";
401     return text;
402 }
403
404
405 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, ThirdSpeciesCounterpoint &s): FirstSpeciesCounterpoint(home, s){
406     thirdSpeciesNotesCp.update(home, s.thirdSpeciesNotesCp);
407     thirdSpeciesMotions.update(home, s.thirdSpeciesMotions);
408     thirdSpeciesMotionCosts.update(home, s.thirdSpeciesMotionCosts);
409     m2IntervalsArray.update(home, s.m2IntervalsArray);
410     m2ZeroArray.update(home, s.m2ZeroArray);
411     thirdHTriadArray.update(home, s.thirdHTriadArray);
412     thirdSpeciesAbsMelodic.update(home, s.thirdSpeciesAbsMelodic);
```

```
}

ThirdSpeciesCounterpoint* ThirdSpeciesCounterpoint::clone(Home home){
    return new ThirdSpeciesCounterpoint(home, *this);
}

IntVarArray ThirdSpeciesCounterpoint::getFirstHInterval(){
    return firstSpeciesHarmonicIntervals;
}

IntVarArray ThirdSpeciesCounterpoint::getMotions(){
    return thirdSpeciesMotions;
}

IntVarArray ThirdSpeciesCounterpoint::getFirstMInterval(){
    return firstSpeciesMelodicIntervals;
}

IntVarArray ThirdSpeciesCounterpoint::getBranchingNotes(){
    return thirdSpeciesNotesCp;
}

int ThirdSpeciesCounterpoint::getHIntervalSize(){
    return thirdSpeciesHarmonicIntervals.size();
}
```

SecondSpeciesCounterpoint.cpp

```
    //
// Created by Damien Sprockeels on 12/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
//

#include "../../headers/Parts/SecondSpeciesCounterpoint.hpp"

/**
 * Second species constructor. Does general constructing. Then calls the general first species constructor
 */
SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec, Stratum* low, CantusFirmus* c, int v_type
    , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
    FirstSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs, g_costs, s_costs, bm, nV) /// super constructor. Applies all rules for the first species to the 1st note of each measure
{
    /// Second species notes in the counterpoint
    secondSpeciesNotesCp = IntVarArray(home, (nMeasures*notesPerMeasure.at(SECOND_SPECIES))-1, IntSet(IntArgs(vector_intersection(cp_range, extended_domain))));
    if(borrowMode==1){
        secondSpeciesNotesCp[secondSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(vector_intersection(cp_range, chromatic_scale))));
    }
    rel(home, secondSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(SECOND_SPECIES),(notes.size())));
    /// Harmonic intervals for the second species notes
    secondSpeciesHarmonicIntervals = IntVarArray(home, (nMeasures*notesPerMeasure.at(SECOND_SPECIES))-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
    // rel(home, secondSpeciesHarmonicIntervals, IRT_EQ, h_intervals.slice(0,4/notesPerMeasure.at(SECOND_SPECIES),(h_intervals.size())));
```

```
24      for(int i = 0; i < secondSpeciesHarmonicIntervals.size(); i++){
25          rel(home, (secondSpeciesHarmonicIntervals[i])==((secondSpeciesNotesCp[i]-
              ↪  low->getNotes()[floor(i/2)*4])%12));
26      }
27
28      /// Melodic intervals for the second species notes
29      secondSpeciesMelodicIntervals = IntVarArray(home, secondSpeciesNotesCp.size()
          ↪  -1, -MAX_STEP, MAX_STEP);
30
31      /// link melodic intervals
32      for(int i = 0; i < secondSpeciesMelodicIntervals.size(); i++)
33          rel(home, secondSpeciesMelodicIntervals[i], IRT_EQ, expr(home,
              ↪  secondSpeciesNotesCp[i+1] - secondSpeciesNotesCp[i]));
34      rel(home, secondSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/
          ↪  notesPerMeasure.at(SECOND_SPECIES),m_intervals_brut.size()));
35
36      secondSpeciesArsisArray = IntVarArray(home, firstSpeciesMelodicIntervals.size
          ↪  (), -MAX_STEP, MAX_STEP);
37      for(int i = 0; i < secondSpeciesArsisArray.size()-1; i++){
38          rel(home, secondSpeciesArsisArray[i], IRT_EQ, expr(home, (
              ↪  secondSpeciesNotesCp[(i*2)+2]-secondSpeciesNotesCp[(i*2)+1]))); //
              ↪   modified by Tom Lai
39      }
40      rel(home, secondSpeciesArsisArray[secondSpeciesArsisArray.size()-1], IRT_EQ,
          ↪  expr(home, secondSpeciesNotesCp[secondSpeciesNotesCp.size()-2]-
41          secondSpeciesNotesCp[secondSpeciesNotesCp.size()-1]));
42
43      secondSpeciesMotions = IntVarArray(home, (nMeasures* notesPerMeasure.at(
          ↪  FIRST_SPECIES) -1), IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION,
          ↪  PARALLEL_MOTION});
44      secondSpeciesMotionCosts = IntVarArray(home, (nMeasures* notesPerMeasure.at(
          ↪  FIRST_SPECIES) -1), IntSet{0, directCost, obliqueCost, contraryCost});
45
46      //the second motions are between the two arsis notes as are the first motions
          ↪   between thesis notes. this way of doing it takes the correct
47      //m intervals between two arsis notes and then calculates the motion with
          ↪  respect to the cantusFirmus
48      for(int i = 0; i < secondSpeciesArsisArray.size(); i++){
49
50          //direct motions help creation
51          BoolVar both_up = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->
              ↪  getMelodicIntervals()[i]>0)); //if both parts are going in the
              ↪  same direction
52          BoolVar both_stay = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->
              ↪  getMelodicIntervals()[i]==0)); //if both parts are staying
53          BoolVar both_down = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->
              ↪  getMelodicIntervals()[i]<0)); //if both parts are going down
54          //oblique motions help creation
55          BoolVar cf_stays_1 = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->
              ↪  getMelodicIntervals()[i]==0)); //if the lowest part stays and one
              ↪  goes up
56          BoolVar cf_stays_2 = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->
              ↪  getMelodicIntervals()[i]==0)); //if the lowest part stays and one
              ↪  goes down
57          BoolVar cp_stays_1 = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->
              ↪  getMelodicIntervals()[i]>0)); //if the lowest part goes up and one
              ↪   stays
58          BoolVar cp_stays_2 = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->
              ↪  getMelodicIntervals()[i]<0)); //if the lowest part goes down and
              ↪  one stays
59          //contrary motions help creation
60          BoolVar cpd_cfu = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->
              ↪  getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
61          BoolVar cpu_cfd = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->
              ↪  getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
62
63          //direct constraints
```

```
64          rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >>
              ↪   (secondSpeciesMotions[i]==PARALLEL_MOTION));
65          rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >>
              ↪   (secondSpeciesMotionCosts[i]==directCost));
66          //oblique constraints
67          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪  isNotLowest[i]==1)) >> (secondSpeciesMotions[i]==OBLIQUE_MOTION));
68          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪  isNotLowest[i]==1)) >> (secondSpeciesMotionCosts[i]==obliqueCost))
              ↪  ;
69          //contrary constraints
70          rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (
              ↪  secondSpeciesMotions[i]==CONTRARY_MOTION));
71          rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (
              ↪  secondSpeciesMotionCosts[i]==contraryCost));
72          //bass constraints
73          rel(home, (isNotLowest[i]==0) >> (secondSpeciesMotions[i]==-1));
74          rel(home, (isNotLowest[i]==0) >> (secondSpeciesMotionCosts[i]==0));
75      }
76
77      //create real motions. we have the thesis-thesis and arsis-arsis motions. now
          ↪   we need the successive m intervals in meas, meaning thesis-arsis
          ↪  inside
78      //a measure. to get this, we iterate in increments of 2 starting from i=0
79      secondSpeciesRealMotions = IntVarArray(home, secondSpeciesMotions.size(),
          ↪  IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION});
80      secondSpeciesRealMotionCosts = IntVarArray(home, secondSpeciesRealMotions.
          ↪  size(), IntSet{0, directCost, obliqueCost, contraryCost});
81      for(int i = 0; i < secondSpeciesRealMotions.size(); i++){
82              rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==1)
                  ↪  >> (secondSpeciesRealMotions[i]==secondSpeciesMotions[i]));
83              rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==0)
                  ↪  >> (secondSpeciesRealMotions[i]==firstSpeciesMotions[i]));
84
85              rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==1)
                  ↪  >> (secondSpeciesRealMotionCosts[i]==secondSpeciesMotionCosts[
                  ↪  i]));
86              rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==0)
                  ↪  >> (secondSpeciesRealMotionCosts[i]==firstSpeciesMotionCosts[i
                  ↪  ]));
87      }
88
89      //create isDiminution Array
90      isDiminution = BoolVarArray(home, (nMeasures* notesPerMeasure.at(
          ↪  FIRST_SPECIES) -1), 0, 1);
91
92      for(int i = 0; i < isDiminution.size(); i++){
93
94          BoolVar btt3 = BoolVar(home, 0, 1);
95          BoolVar btt4 = BoolVar(home, 0, 1);
96          BoolVar bta2nd = BoolVar(home, 0, 1);
97          BoolVar btt3rd = BoolVar(home, 0, 1);
98          BoolVar bat2nd = BoolVar(home, 0, 1);
99          BoolVar band = BoolVar(home, 0, 1);
100
101         rel(home, expr(home, abs(firstSpeciesMelodicIntervals[i])), IRT_EQ, 3,
              ↪  Reify(btt3));
102         rel(home, expr(home, abs(firstSpeciesMelodicIntervals[i])), IRT_EQ, 4,
              ↪  Reify(btt4));
103         rel(home, expr(home, abs(secondSpeciesMelodicIntervals[i*2])), IRT_LQ, 2,
              ↪   Reify(bta2nd));
104         rel(home, expr(home, abs(secondSpeciesMelodicIntervals[i*2+1])), IRT_LQ,
              ↪  2, Reify(bat2nd));
105         rel(home, btt3, BOT_OR, btt4, btt3rd);
106         rel(home, bta2nd, BOT_AND, btt3rd, band);
107         rel(home, band, BOT_AND, bat2nd, isDiminution[i]);
108
109     }
```

```
110
111     penultCostArray = IntVarArray(home, 1, IntSet({0, penultCost}));
112     /// Constraints
113
114     // 2.H2 : Arsis harmonies cannot be dissonant except if there is a diminution
          ↪ .
115     if (activeConstraints[SP2_2H2]) {
116         H2_2_arsisHarmoniesCannotBeDisonnant(home, this);
117     }
118
119     //2.M1
120     if (activeConstraints[SP2_2M1]) {
121         M1_2_octaveLeap(home, this, low);
122     }
123
124     //2.P2 : battuta adapted (2.P3 ? Tom Lai)
125     if (activeConstraints[SP2_2P3]) {
126         P3_2_noBattuta(home, this);
127     }
128
129     //1.M2
130     if (activeConstraints[SP1_1M2_2V]) {
131         for (size_t i = 0; i < secondSpeciesMelodicIntervals.size(); i++)
132         {
133             rel(home, (secondSpeciesMelodicIntervals[i] <= 8) || (
                  ↪  secondSpeciesMelodicIntervals[i] == 12));
134         }
135
136     }
137
138     // combined costs
139     toCombineCosts = IntVarArray(home, 1, 0, 10000);
140     toCombineCostNames = {"1H1"};
141     //need to set constraintCosts[0]
142     add_toCombineCost(home, 0, disArray, toCombineCosts);
143 }
144
145 /**
146  * 2 VOICES CONSTRUCTOR
147  */
148 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<
      ↪  int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type
149     , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
          ↪  nV) :
150     SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c,
          ↪  v_type, m_costs, g_costs, s_costs, bm, nV)
151 {
152     costs = IntVarArray(home, 6, 0, 1000000);
153     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "penult"};
154
155     // 2.H3 : penult cost
156     if (activeConstraints[SP2_2H3_2V]) {
157         H3_2_penultimateNoteDomain(home, this);
158     }
159
160     //can do better than this?
161     //2.M2
162     if (activeConstraints[SP2_2M2_2V]) {
163         M2_2_2v_twoConsecutiveNotesAreNotTheSame(home, this);
164     }
165
166     //2.P1 : adapted no direct motion rule from first species to real motions
167     if (activeConstraints[SP2_2P1_2V]) {
168         P1_2_2v_noDirectMotionFromPerfectConsonance(home, this);
169     }
170
171     //set cost[0] to be fifth cost
```

```
172     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
          ↪  .at(SECOND_SPECIES)), fifthCostArray.size())), costs);
173     //set cost[1] to be octave cost
174     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/
          ↪  notesPerMeasure.at(SECOND_SPECIES)), octaveCostArray.size())), costs);
175     //set cost[2] to be motion cost
176     add_cost(home, 2, secondSpeciesRealMotionCosts, costs);
177     //set cost[3] to be melodic cost
178     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪  notesPerMeasure.at(SECOND_SPECIES)), melodicDegreeCost.size())), costs)
          ↪  ;
179     //need to set cost[4] to be off cost
180     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
          ↪  at(SECOND_SPECIES)), offCostArray.size())), costs);
181     //set cost[5] to be penult sixth cost
182     add_cost(home, 5, penultCostArray, costs);
183
184 }
185
186 /**
187  * 3 VOICES CONSTRUCTOR
188  */
189 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<
      ↪  int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
190     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
          ↪  nV1, int nV2) :
191     SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c,
          ↪  v_type, m_costs, g_costs, s_costs, bm, nV2)
192 {
193     costs = IntVarArray(home, 8, 0, 1000000);
194
195     varietyCostArray = IntVarArray(home, 3*(secondSpeciesHarmonicIntervals.size()
          ↪  -2), IntSet({0, varietyCost}));
196     directCostArray = IntVarArray(home, secondSpeciesRealMotions.size()-1,IntSet
          ↪  ({0, directMoveCost}));
197
198     // DISABLED
199     // // 2.H3 : penult cost
200     // if (activeConstraints[SP2_2H3_3V]) {
201     //     H3_2_penultimateNoteDomain(home, this);
202     // }
203
204     //P1 3 voices version
205     if (activeConstraints[SP2_1P1_3V]) {
206         P1_2_3v_noDirectMotionFromPerfectConsonance(home, this);
207     }
208
209     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "
          ↪  direct", "penult"};
210     //need to set cost[0] to be off cost
211     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
          ↪  at(SECOND_SPECIES)), offCostArray.size())), costs);
212     //set cost[1] to be fifth cost
213     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
          ↪  .at(SECOND_SPECIES)), fifthCostArray.size())), costs);
214     //set cost[2] to be octave cost
215     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/
          ↪  notesPerMeasure.at(SECOND_SPECIES)), octaveCostArray.size())), costs);
216     //need to set cost[3] to be variety cost
217     add_cost(home, 3, varietyCostArray, costs);
218     //set cost[4] to be motion cost
219     add_cost(home, 4, secondSpeciesRealMotionCosts, costs);
220     //set cost[5] to be melodic cost
221     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪  notesPerMeasure.at(SECOND_SPECIES)), melodicDegreeCost.size())), costs)
          ↪  ;
222     //need to set cost[6] to be direct cost
223     add_cost(home, 6, directCostArray, costs);
```

```
224     //set cost[7] to be penult sixth cost
225     add_cost(home, 7, penultCostArray, costs);
226 
227 }
228 
229 /**
230  * 4 VOICES CONSTRUCTOR
231  */
232 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<
      ↪ int> cf,int lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
233     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
        ↪ nV1, int nV2, int nV3) :
234     SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c,
        ↪ v_type, m_costs, g_costs, s_costs, bm, nV3)
235 {
236 
237     costs = IntVarArray(home, 8, 0, 1000000);
238     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "variety", "
        ↪ direct", "penult"};
239 
240     varietyCostArray = IntVarArray(home, 3*(secondSpeciesHarmonicIntervals.size()
        ↪ -2), IntSet({0, varietyCost}));
241     directCostArray = IntVarArray(home, secondSpeciesRealMotions.size()-1,IntSet
        ↪ ({0, 2, directMoveCost}));
242 
243     // DISABLED
244     // 2.H3 : penult cost
245     // if (activeConstraints[SP2_2H3_4V]) {
246     //     H3_2_penultimateNoteDomain(home, this);
247     // }
248 
249     //P1 4 voices version
250     if (activeConstraints[SP2_1P1_4V]) {
251         P1_2_4v_noDirectMotionFromPerfectConsonance(home, this);
252     }
253 
254     //set cost[0] to be fifth cost
255     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
        ↪ .at(SECOND_SPECIES), fifthCostArray.size())), costs);
256     //set cost[1] to be octave cost
257     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/
        ↪ notesPerMeasure.at(SECOND_SPECIES), octaveCostArray.size())), costs);
258     //set cost[2] to be motion cost
259     add_cost(home, 2, secondSpeciesRealMotionCosts, costs);
260     //set cost[3] to be melodic cost
261     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/
        ↪ notesPerMeasure.at(SECOND_SPECIES), melodicDegreeCost.size())), costs)
        ↪ ;
262     //need to set cost[4] to be off cost
263     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
        ↪ at(SECOND_SPECIES), offCostArray.size())), costs);
264     //need to set cost[5] to be variety cost
265     add_cost(home, 5, varietyCostArray, costs);
266     //need to set cost[6] to be direct cost
267     add_cost(home, 6, directCostArray, costs);
268     //set cost[7] to be penult sixth cost
269     add_cost(home, 7, penultCostArray, costs);
270 }
271 
272 string SecondSpeciesCounterpoint::to_string() const {
273     string text = "\nSecond␣species␣diminution␣array␣:␣" + boolVarArray_to_string
        ↪ (isDiminution) + "\n";
274     text += "Second␣species␣isLowest␣array␣:␣" + boolVarArray_to_string(
        ↪ isNotLowest) + "\n";
275     text += "First␣species␣m␣intervals␣:␣" + intVarArray_to_string(
        ↪ firstSpeciesMelodicIntervals) + "\n";
276     text += "Second␣species␣m␣intervals␣:␣" + intVarArray_to_string(
        ↪ secondSpeciesMelodicIntervals) + "\n";
```

```
    text += "Second species h intervals : " + intVarArray_to_string(
      ↪ secondSpeciesHarmonicIntervals) + "\n";
    text += "Second species is Consonance : " + boolVarArray_to_string(
      ↪ isConsonance) + "\n";
    text += "Second species notes : " + intVarArray_to_string(notes) + "\n";
    text += "First species : " + FirstSpeciesCounterpoint::to_string() + "\n";
    return text;
}


SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home,
  ↪ SecondSpeciesCounterpoint &s): FirstSpeciesCounterpoint(home, s){
    secondSpeciesNotesCp.update(home, s.secondSpeciesNotesCp);
    secondSpeciesHarmonicIntervals.update(home, s.secondSpeciesHarmonicIntervals)
      ↪ ;
    secondSpeciesMelodicIntervals.update(home, s.secondSpeciesMelodicIntervals);
    secondSpeciesMotions.update(home, s.secondSpeciesMotions);
    secondSpeciesMotionCosts.update(home, s.secondSpeciesMotionCosts);
    secondSpeciesRealMotionCosts.update(home, s.secondSpeciesRealMotionCosts);

    secondSpeciesArsisArray.update(home, s.secondSpeciesArsisArray);
}

SecondSpeciesCounterpoint* SecondSpeciesCounterpoint::clone(Home home){
    return new SecondSpeciesCounterpoint(home, *this);
}

IntVarArray SecondSpeciesCounterpoint::getFirstHInterval(){
    return firstSpeciesHarmonicIntervals;
}

IntVarArray SecondSpeciesCounterpoint::getMotions(){
    return secondSpeciesMotions;
}

IntVarArray SecondSpeciesCounterpoint::getFirstMInterval(){
    return firstSpeciesMelodicIntervals;
}

IntVarArray SecondSpeciesCounterpoint::getBranchingNotes(){
    return secondSpeciesNotesCp;
}

int SecondSpeciesCounterpoint::getHIntervalSize(){
    return secondSpeciesHarmonicIntervals.size();
}
```

FirstSpeciesCounterpoint.cpp

```
    //
// Created by Damien Sprockeels on 11/06/2024.
// Extended and developed by Luc Cleenewerk and Diego de Patoul up to August
  ↪ 2024.
//

#include "../../headers/Parts/FirstSpeciesCounterpoint.hpp"

/**
 * GENERAL CONSTRUCTOR
 */

FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<
  ↪ int> cf, int lb, int ub, Species mSpecies, Stratum* low, CantusFirmus* c,
    int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs,
      ↪ int bm, int nV):
```

```
14          Part(home, nMes, mSpecies, cf, lb, ub, v_type, m_costs, g_costs, s_costs,
              ↪  nV, bm) { /// super constructor
15
16      motherSpecies =         mSpecies;
17      for(int i = lowerBound; i <= upperBound; i++){
18          cp_range.push_back(i);
19      }
20      /*
21      if borrowMode is enabled, the extended domain is extended to make the
          ↪  inclusion of borrowed notes possible. We can see from Fux's examples
22      that he does like to borrow notes, so the borrow cost should just do the job
          ↪  and we should still allow borrowed notes, not outright forbid them
23      */
24      if(borrowMode==1){
25          extended_domain = vector_union(cp_range, vector_union(scale,
              ↪  borrowed_scale));
26      } else {
27          extended_domain = vector_intersection(cp_range, vector_union(scale,
              ↪  borrowed_scale));
28      }
29      off_domain = vector_difference(vector_intersection(cp_range, scale),
          ↪  lowerBound, upperBound);
30
31      /// First species notes in the counterpoint
32      firstSpeciesNotesCp = IntVarArray(home, nMeasures * notesPerMeasure.at(
          ↪  FIRST_SPECIES), IntSet(IntArgs(vector_intersection(cp_range,
          ↪  extended_domain))));
33
34      if(borrowMode==1 && motherSpecies==FIRST_SPECIES){
35          firstSpeciesNotesCp[firstSpeciesNotesCp.size()-2] = IntVar(home, IntSet(
              ↪  IntArgs(vector_intersection(cp_range, chromatic_scale))));
36      } else {
37          firstSpeciesNotesCp[firstSpeciesNotesCp.size()-2] = IntVar(home, IntSet(
              ↪  IntArgs(vector_intersection(cp_range, extended_domain))));
38      }
39
40      rel(home, firstSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(
          ↪  FIRST_SPECIES),notes.size()));
41
42      //Harmonic intervals
43      for (int i = 0; i < h_intervals.size(); i++) {
44          rel(home, (h_intervals[i])==(notes[i]-low->getNotes()[i])%12);
45      }
46
47      /// Harmonic intervals for the first species notes
48      firstSpeciesHarmonicIntervals = IntVarArray(home, nMeasures* notesPerMeasure.
          ↪  at(FIRST_SPECIES), -PERFECT_OCTAVE, PERFECT_OCTAVE);
49
50      for(int i = 0; i < firstSpeciesHarmonicIntervals.size(); i++){
51          rel(home, (firstSpeciesHarmonicIntervals[i])==((firstSpeciesNotesCp[i]-
              ↪  low->getNotes()[i*4])%12));
52      }
53
54      rel(home, firstSpeciesHarmonicIntervals, IRT_EQ, h_intervals.slice(0,4/
          ↪  notesPerMeasure.at(FIRST_SPECIES),h_intervals.size()));
55
56      /// Melodic intervals for the first species notes
57      firstSpeciesMelodicIntervals = IntVarArray(home, nMeasures* notesPerMeasure.
          ↪  at(FIRST_SPECIES) -1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
58      for(int i = 0; i < firstSpeciesMelodicIntervals.size(); i++)
59        rel(home, firstSpeciesMelodicIntervals[i], IRT_EQ, expr(home,
              ↪  firstSpeciesNotesCp[i+1] - firstSpeciesNotesCp[i]));
60
61      //create the is off array which determines if a notes is borrowed or not
62      is_off = BoolVarArray(home, notes.size(), 0, 1);
63      initializeIsOffArray(home, this);
64
65      //create off_cost array
```

```
66      offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
67
68      //create the melodic degree cost array
69      melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({
          ↪  secondCost, thirdCost, fourthCost, tritoneCost, fifthCost,
70          sixthCost, seventhCost, octaveCost}));
71
72      //create motions arrays
73      firstSpeciesMotions = IntVarArray(home, nMeasures* notesPerMeasure.at(
          ↪  FIRST_SPECIES) -1, IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION,
          ↪  PARALLEL_MOTION});
74      firstSpeciesMotionCosts = IntVarArray(home, firstSpeciesMotions.size(),
          ↪  IntSet{0, directCost, obliqueCost, contraryCost});
75
76      //create motions
77
78      for(int i = 0; i < firstSpeciesMotions.size(); i++){
79
80          //direct motions help creation
81          BoolVar both_up = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low->
              ↪  getMelodicIntervals()[i]>0)); //if both parts are going in the
              ↪  same direction
82          BoolVar both_stay = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(low
              ↪  ->getMelodicIntervals()[i]==0)); //if both parts are staying
83          BoolVar both_down = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low
              ↪  ->getMelodicIntervals()[i]<0)); //if both parts are going down
84          //oblique motions help creation
85          BoolVar cf_stays_1 = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low
              ↪  ->getMelodicIntervals()[i]==0)); //if the lowest part stays and
              ↪  one goes up
86          BoolVar cf_stays_2 = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low
              ↪  ->getMelodicIntervals()[i]==0)); //if the lowest part stays and
              ↪  one goes down
87          BoolVar cp_stays_1 = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(
              ↪  low->getMelodicIntervals()[i]>0)); //if the lowest part goes up
              ↪  and one stays
88          BoolVar cp_stays_2 = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(
              ↪  low->getMelodicIntervals()[i]<0)); //if the lowest part goes down
              ↪  and one stays
89          //contrary motions help creation
90          BoolVar cpd_cfu = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low->
              ↪  getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
91          BoolVar cpu_cfd = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low->
              ↪  getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
92
93          //direct constraints
94          rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i
              ↪  ]==1)) >> (firstSpeciesMotions[i]==PARALLEL_MOTION));
95          rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i
              ↪  ]==1)) >> (firstSpeciesMotionCosts[i]==directCost));
96          //oblique constraints
97          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪  this->isNotLowest[i]==1)) >> (firstSpeciesMotions[i]==
              ↪  OBLIQUE_MOTION));
98          rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (
              ↪  this->isNotLowest[i]==1)) >> (firstSpeciesMotionCosts[i]==
              ↪  obliqueCost));
99          //contrary constraints
100         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
              ↪  firstSpeciesMotions[i]==CONTRARY_MOTION));
101         rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
              ↪  firstSpeciesMotionCosts[i]==contraryCost));
102         //bass constraints
103         rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotions[i]==-1));
104         rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotionCosts[i]==0));
105
106     }
107
```

```
108     // create pefectConsArray
109     fifthCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_fifthCost
          ↪ }));
110     octaveCostArray = IntVarArray(home, h_intervals.size(), IntSet({0,
          ↪ h_octaveCost}));
111
112     isConsonance = BoolVarArray(home, h_intervals.size(), 0, 1);
113     //this loop checks that every harmonic interval is a consonance or not
114     for(int i = 0; i < isConsonance.size(); i++){
115         rel(home, expr(home, (h_intervals[i]==UNISSON)||(h_intervals[i]==
              ↪ MINOR_THIRD)||(h_intervals[i]==MAJOR_THIRD)||(h_intervals[i]==
              ↪ PERFECT_FIFTH)||
116           (h_intervals[i]==MINOR_SIXTH)||(h_intervals[i]==MAJOR_SIXTH)||(
                ↪ h_intervals[i]==PERFECT_OCTAVE)), IRT_EQ, isConsonance[i]);
117     }
118     /// General rules
119
120     //G4 :
121     if (activeConstraints[SP1_G4]) {
122         G4_counterpointMustBeInTheSameKey(home, this);
123     }
124
125     // G7 : melodic intervals should be small (works for 1st, 2nd and 3rd species
          ↪ )
126     if (activeConstraints[SP1_G7]) {
127         G7_melodicIntervalsShouldBeSmall(home, this, motherSpecies);
128     }
129
130     /// Harmonic rules
131     /// H1 from Thibault: All harmonic intervals must be consonances
132     if (activeConstraints[SP1_1H1]) {
133         H1_1_harmonicIntervalsAreConsonances(home, this);
134         // disArray = IntVarArray(home, firstSpeciesHarmonicIntervals.size(),
              ↪ IntSet{0, 1});
135         // // Define the set of consonant intervals
136         // IntSet consonantIntervals({UNISSON, MINOR_THIRD, MAJOR_THIRD,
              ↪ PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
137         //     -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -
              ↪ MAJOR_SIXTH, -PERFECT_OCTAVE});
138
139         // // Loop through each harmonic interval
140         // for (size_t i = 0; i < firstSpeciesHarmonicIntervals.size(); i++) {
141         //     // Create a Boolean variable to check if h_intervals[i] is in
              ↪ consonantIntervals
142         //     BoolVar isConsonant(home, 0, 1);
143         //     dom(home, firstSpeciesHarmonicIntervals[i], consonantIntervals,
              ↪ isConsonant);
144
145         //     // If the interval is consonant, set disArray[i] to 0
146         //     rel(home, isConsonant >> (disArray[i] == 0));
147
148         //     // Otherwise, set disArray[i] to H1_1_cost
149         //     rel(home, !isConsonant >> (disArray[i] == 1));
150         // }
151
152     }
153
154     //H2 and H3 are found in the TwoVoiceCounterpoint class, since these rules
          ↪ are 2 voice specific
155
156     // H4 : applied as rule G9 in the Two, Three and FourVoice
          ↪ Counterpointproblem
157
158     // H6 from Thibault : Imperfect consonances are preferred
159     if (activeConstraints[SP1_1H6]) {
160         H6_1_preferImperfectConsonances(home, this);
161     }
162
```

```
163     // combined costs
164     toCombineCosts = IntVarArray(home, 1, 0, 10000);
165     // toCombineCostNames = {"1H1"};
166     // //need to set constraintCosts[0]
167     // add_toCombineCost(home, 0, disArray, toCombineCosts);
168 }
169
170 /**
171  * 2 VOICES CONSTRUCTOR
172  */
173
174 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<
      ↪ int> cf, int lb, int ub, Stratum* low, CantusFirmus* c, int v_type
175     , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
        ↪ nV) :
176         FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c,
              ↪ v_type, m_costs, g_costs, s_costs, bm, nV) ///call the general
              ↪ constructor
177 {
178     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/
          ↪ notesPerMeasure.at(FIRST_SPECIES),m_intervals_brut.size()));
179
180     //H7,H8 from Thibault : penultimate note major sixth or minor third
181     // cout << "H7_1_2v_penultimateSixthOrThird" << endl;
182     if (activeConstraints[SP1_1H7_2V]) {
183         H7_1_2v_penultimateSixthOrThird(home, this);
184     }
185
186     //1.M2 from Thibault: Melodic intervals cannot exceed a minor sixth
187     if (activeConstraints[SP1_1M2_2V]) {
188         M1_1_2v_melodicIntervalsNotExceedMinorSixth(home, this);
189     }
190
191     // Motion rules
192     //1.P1 from Thibault : Perfect consonances cannot be reached by direct motion
193     if (activeConstraints[SP1_1P1_2V]) {
194         P1_1_2v_noDirectMotionFromPerfectConsonance(home, this);
195     }
196
197     //P2 from Thibault : already done when creating motions array
198
199     //P3 from Thibault : no battuta
200     if (activeConstraints[SP1_1P3_2V]) {
201         P3_1_noBattuta(home, this);
202     }
203
204     costs = IntVarArray(home, 5, 0, 1000000);
205     cost_names = {"fifth", "octave", "motion", "melodic", "borrow"};
206
207     //set cost[0] to be fifth cost
208     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
          ↪ .at(FIRST_SPECIES), fifthCostArray.size())), costs);
209     //set cost[1] to be octave cost
210     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/
          ↪ notesPerMeasure.at(FIRST_SPECIES), octaveCostArray.size())), costs);
211     //set cost[2] to be motion cost
212     add_cost(home, 2, firstSpeciesMotionCosts, costs);
213     //set cost[3] to be melodic cost
214     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪ notesPerMeasure.at(FIRST_SPECIES), melodicDegreeCost.size())), costs);
215     //set cost[4] to be off cost
216     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
          ↪ at(FIRST_SPECIES), offCostArray.size())), costs);
217 }
218
219 /**
220  * 3 VOICES CONSTRUCTOR
221  */
```

```
222
223 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<
      ↪ int> cf, int lb, int ub,  Stratum* low, CantusFirmus* c,  int v_type,
224     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
          ↪ nV1, int nV2) :
225     FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c,
          ↪ v_type, m_costs, g_costs, s_costs, bm, nV2) ///call the general
          ↪ constructor
226 {
227     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/
          ↪ notesPerMeasure.at(FIRST_SPECIES),m_intervals_brut.size()));
228
229     varietyCostArray = IntVarArray(home, 3*(firstSpeciesHarmonicIntervals.size()
          ↪ -2), IntSet({0, varietyCost}));
230     directCostArray = IntVarArray(home, firstSpeciesMotions.size()-1,IntSet({0,
          ↪ directMoveCost}));
231
232     //1.H7 -- after careful testing, this constraint does not work Fux's examples
233     if (activeConstraints[SP1_1H7_3V]) {
234         H7_1_3v_penultimateSixthOrThird(home, this);
235     }
236
237     //1.M2 from Thibault: Melodic intervals cannot exceed a minor sixth (also
          ↪ include octave?)
238     if (activeConstraints[SP1_1M2_3V]) {
239         M1_1_3v_melodicIntervalsNotExceedMinorSixth(home, this);
240     }
241
242     //1.P1
243     if (activeConstraints[SP1_1P1_3V]) {
244         P1_1_3v_noDirectMotionFromPerfectConsonance(home, this);
245     }
246
247     //1.P3 from Thibault : no battuta
248     if (activeConstraints[SP1_1P3_3V]) {
249         P3_1_noBattuta(home, this);
250     }
251
252     costs = IntVarArray(home, 7, 0, 1000000);
253     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "
          ↪ direct"};
254     //need to set cost[0] to be off cost
255     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
          ↪ at(FIRST_SPECIES), offCostArray.size())), costs);
256     //set cost[1] to be fifth cost
257     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
          ↪ .at(FIRST_SPECIES), fifthCostArray.size())), costs);
258     //set cost[2] to be octave cost
259     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/
          ↪ notesPerMeasure.at(FIRST_SPECIES), octaveCostArray.size())), costs);
260     //need to set cost[3] to be variety cost
261     add_cost(home, 3, varietyCostArray, costs);
262     //set cost[4] to be motion cost
263     add_cost(home, 4, firstSpeciesMotionCosts, costs);
264     //set cost[5] to be melodic cost
265     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪ notesPerMeasure.at(FIRST_SPECIES), melodicDegreeCost.size())), costs);
266     //need to set cost[6] to be direct cost
267     add_cost(home, 6, directCostArray, costs);
268 }
269
270 /**
271  * 4 VOICES CONSTRUCTOR
272  */
273
274 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<
      ↪ int> cf, int lb, int ub, Stratum* low, CantusFirmus* c,  int v_type,
```

```
275     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int
          ↪ nV1, int nV2, int nV3):
276     FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c,
          ↪ v_type, m_costs, g_costs, s_costs, bm, nV3)
277 {
278     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/
          ↪ notesPerMeasure.at(FIRST_SPECIES),m_intervals_brut.size()));
279
280     varietyCostArray = IntVarArray(home, 3*(firstSpeciesHarmonicIntervals.size()
          ↪ -2), IntSet({0, varietyCost}));
281     directCostArray = IntVarArray(home, firstSpeciesMotions.size()-1,IntSet({0,
          ↪ 2, directMoveCost}));
282
283     /// M2 from Thibault: Melodic intervals cannot exceed a minor sixth (also
          ↪ include octave?)
284     if (activeConstraints[SP1_1M2_4V]) {
285         M1_1_3v_melodicIntervalsNotExceedMinorSixth(home, this);
286     }
287
288     //P1 4 voices version
289     if (activeConstraints[SP1_1P1_4V]) {
290         P1_1_4v_noDirectMotionFromPerfectConsonance(home, this);
291     }
292
293     //P3 from Thibault : no battuta
294     if (activeConstraints[SP1_1P3_4V]) {
295         P3_1_noBattuta(home, this);
296     }
297
298     costs = IntVarArray(home, 7, 0, 1000000);
299     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "
          ↪ direct"};
300     //need to set cost[0] to be off cost
301     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.
          ↪ at(FIRST_SPECIES), offCostArray.size())), costs);
302     //set cost[1] to be fifth cost
303     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure
          ↪ .at(FIRST_SPECIES), fifthCostArray.size())), costs);
304     //set cost[2] to be octave cost
305     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/
          ↪ notesPerMeasure.at(FIRST_SPECIES), octaveCostArray.size())), costs);
306     //need to set cost[3] to be variety cost
307     add_cost(home, 3, varietyCostArray, costs);
308     //set cost[4] to be motion cost
309     add_cost(home, 4, firstSpeciesMotionCosts, costs);
310     //set cost[5] to be melodic cost
311     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/
          ↪ notesPerMeasure.at(FIRST_SPECIES), melodicDegreeCost.size())), costs);
312     //need to set cost[6] to be direct cost
313     add_cost(home, 6, directCostArray, costs);
314 }
315
316 /**
317  * This function returns a string with the characteristics of the counterpoint.
          ↪ It calls the to_string() method from
318  * the Part class and adds 1st species specific characteristics.
319  * @return a string representation of the current instance of the
          ↪ FirstSpeciesCounterpoint class.
320  */
321 string FirstSpeciesCounterpoint::to_string() const {
322     string text = Part::to_string() + "\nFirst␣species␣:\n";
323     text += "First␣species␣first␣notes:␣" + intVarArray_to_string(
          ↪ firstSpeciesNotesCp) + "\n";
324     text += "First␣species␣isLowest␣array␣:␣" + boolVarArray_to_string(
          ↪ isNotLowest) + "\n";
325     text += "First␣species␣motions␣:␣" + intVarArray_to_string(
          ↪ firstSpeciesMotions) + "\n";
```

```
326     text += "First species h intervals : " + intVarArray_to_string(
          ↪ firstSpeciesHarmonicIntervals) + "\n";
327     return text;
328 }
329
330 // clone constructor
331 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home,
      ↪ FirstSpeciesCounterpoint &s) : Part(home, s){
332     motherSpecies = s.motherSpecies;
333     //cantus = s.cantus;
334 }
335
336 FirstSpeciesCounterpoint* FirstSpeciesCounterpoint::clone(Home home){
337     return new FirstSpeciesCounterpoint(home, *this);
338 }
339
340 IntVarArray FirstSpeciesCounterpoint::getBranchingNotes(){
341     return firstSpeciesNotesCp;
342 }
343
344 IntVarArray FirstSpeciesCounterpoint::getFirstHInterval(){
345     return firstSpeciesHarmonicIntervals;
346 }
347
348 IntVarArray FirstSpeciesCounterpoint::getMotions(){
349     return firstSpeciesMotions;
350 }
351
352 IntVarArray FirstSpeciesCounterpoint::getFirstMInterval(){
353     return firstSpeciesMelodicIntervals;
354 }
355
356 int FirstSpeciesCounterpoint::getHIntervalSize(){
357     return firstSpeciesHarmonicIntervals.size();
358 }
```

A.6 Other source files

constraints.cpp

```
1     //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file contains the implementations of the functions that post the
      ↪ constraints.
4 //
5
6 #include "../headers/constraints.hpp"
7
8 void initializeIsOffArray(Home home, Part* part){
9     for(int i = 0; i < part->getIsOffArray().size(); i++){
        ↪                                 //loop goes through every note of the
        ↪ counterpoint
10        IntVarArray res = IntVarArray(home, part->getOffDomain().size(), 0, 1);
11        IntVar sm = IntVar(home, 0, part->getOffDomain().size());
12        for(int l = 0; l < part->getOffDomain().size(); l++){
            ↪                              //loop goes through the domain of
            ↪ borrowed notes
13            BoolVar b1 = BoolVar(home, 0, 1);
14            rel(home, part->getNotes()[i], IRT_EQ, part->getOffDomain()[l], Reify
                ↪ (b1)); //checks if a note is borrowed
15            ite(home, b1, IntVar(home, 1, 1), IntVar(home, 0, 0), res[l]);
                ↪                 //if it is borrowed, set res[l] to one
16        }
17        IntVarArgs x(res.size());
```

```
        for(int t = 0; t < part->getOffDomain().size(); t++){
            x[t] = res[t];
              ↪
              ↪  //create an IntVarArgs array of the contents of res[l] (for
              ↪  sum)
        }
        rel(home, sm, IRT_EQ, expr(home, sum(x)));
          ↪                                          //set the sm variable to be
          ↪  the sum of the IntVarArgs
        rel(home, sm, IRT_GR, 0, Reify(part->getIsOffArray()[i]));
          ↪                        //if the sum is greater than 0, then the note
          ↪   is borrowed (off key)
    }
}

void R9_5_twoFifthSpeciesDiversity_3v(Home home, Part* cp1, Part* cp2){
    if(cp1->getSpecies()==FIFTH_SPECIES && cp2->getSpecies()==FIFTH_SPECIES){
        BoolVarArray isSameSpecies = BoolVarArray(home, cp1->getNotes().size(),
          ↪  0, 1);
        IntVarArray isSameSpeciesInt = IntVarArray(home, cp1->getNotes().size(),
          ↪  0, 1);
        IntVar percentageSame = IntVar(home, 0, cp1->getNotes().size());
        for(int i = 0; i < cp1->getNotes().size(); i++){
            rel(home, cp1->getSpeciesArray()[i], IRT_EQ, cp2->getSpeciesArray()[i
              ↪  ], Reify(isSameSpecies[i]));
            rel(home, isSameSpeciesInt[i], IRT_EQ, 1, Reify(isSameSpecies[i]));
        }
        rel(home, percentageSame, IRT_EQ, expr(home, sum(isSameSpeciesInt)));
        rel(home, percentageSame, IRT_LE, floor(cp1->getNotes().size()/2));
    }
}

void noMinorSecondBetweenUpper(Home home, vector<Stratum*> strata){
    // For all pairs of voices s1 < s2:
    for(int s1 = 0; s1 < strata.size(); s1++){
        for(int s2 = s1+1; s2 < strata.size(); s2++){
            // For each measure i:
            for(int i = 0; i < strata[1]->getNMeasures(); i++){
                rel(home, expr(home, abs(strata[s2]->getNotes()[i*4] - strata[s1
                  ↪  ]->getNotes()[i*4]) % 12), IRT_NQ, MINOR_SECOND);
            }
        }
    }
}

void G4_counterpointMustBeInTheSameKey(Home home, Part* part){
    for(int i = 0; i < part->getIsOffArray().size(); i++){
        rel(home, (part->getIsOffArray()[i]==0) >> (part->getOffCostArray()[i
          ↪  ]==0));                          //sets no cost if not borrowed
        rel(home, (part->getIsOffArray()[i]==1) >> (part->getOffCostArray()[i]==
          ↪  part->getBorrowCost())); //sets a cost if borrowed
    }
}

void G6_noChromaticMelodies(Home home, Part* part, int mSpec){

    //forbids that three notes one after the other increase by a step, making the
      ↪   melody chromatic

    for(int i = 0; i < part->getMelodicIntervals().size()-1; i+=4/notesPerMeasure
      ↪  .at(mSpec)){
        rel(home, expr(home, part->getMelodicIntervals()[i]==1), BOT_AND, expr(
          ↪  home, part->getMelodicIntervals()[i+1]==1), 0);
        rel(home, expr(home, part->getMelodicIntervals()[i]==-1), BOT_AND, expr(
          ↪  home, part->getMelodicIntervals()[i+1]==-1), 0);
    }
}

```

```
69  void G7_melodicIntervalsShouldBeSmall(Home home, Part* part, int mSpec){
70
71      //loop goes through every note of the melodic interval and sets the cost
72
73      for(int i = 0; i < part->getMelodicIntervals().size(); i+=4/notesPerMeasure.
          ↪ at(mSpec)){
74          rel(home, (abs(part->getMelodicIntervals()[i])<MINOR_THIRD) >> (part->
              ↪ getMelodicDegreeCost()[i]==part->getSecondCost()));
75          rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_THIRD || abs(part->
              ↪ getMelodicIntervals()[i])==MAJOR_THIRD) >> (part->
              ↪ getMelodicDegreeCost()[i]==part->getThirdCost()));
76          rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_FOURTH) >> (part
              ↪ ->getMelodicDegreeCost()[i]==part->getFourthCost()));
77          rel(home, (abs(part->getMelodicIntervals()[i])==TRITONE) >> (part->
              ↪ getMelodicDegreeCost()[i]==part->getTritoneCost()));
78          rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_FIFTH) >> (part->
              ↪ getMelodicDegreeCost()[i]==part->getFifthCost()));
79          rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_SIXTH || abs(part->
              ↪ getMelodicIntervals()[i])==MAJOR_SIXTH) >> (part->
              ↪ getMelodicDegreeCost()[i]==part->getSixthCost()));
80          rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_SEVENTH || abs(part
              ↪ ->getMelodicIntervals()[i])==MAJOR_SEVENTH) >> (part->
              ↪ getMelodicDegreeCost()[i]==part->getSeventhCost()));
81          rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_OCTAVE) >> (part
              ↪ ->getMelodicDegreeCost()[i]==part->getOctaveCost()));
82      }
83  }
84
85  void G9_lastChordSameAsFundamental(Home home, Stratum* lowest, Part* cantusFirmus
      ↪ ){
86
87      //we check the last note of the cantusFirmus to set the last note of the
          ↪ counterpoint to be in the same key
88
89      rel(home, expr(home, lowest->getNotes()[lowest->getNotes().size()-1]%12),
          ↪ IRT_EQ, expr(home, cantusFirmus->getNotes()[cantusFirmus->getNotes().
          ↪ size()-1]%12));
90      rel(home, expr(home, lowest->getNotes()[0]%12), IRT_EQ, expr(home,
          ↪ cantusFirmus->getNotes()[0]%12)); // Modified by Tom Lai
91  }
92
93  void H1_1_harmonicIntervalsAreConsonances(Home home, Part* part){
94
95      //we also include negative values, since the fourth species could render the
          ↪ harmonic intervals to be negative because of its deplacement
96      //mostly the harmonic intervals are going to be positive
97
98      dom(home, part->getFirstSpeciesHIntervals(), IntSet({UNISSON, MINOR_THIRD,
          ↪ MAJOR_THIRD, PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
99          -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -
              ↪ PERFECT_OCTAVE}));
100 }
101
102 void H1_3_fiveConsecutiveNotesByJointDegree(Home home, Part* part){
103     for(int i = 0; i < part->getIs5QNArray().size(); i++){
104         rel(home, (part->getIs5QNArray()[i]==1) >> (part->getConsonance()[(i*4)
              ↪ +2]==1));
105     }
106 }
107
108 void H2_1_startWithPerfectConsonance(Home home, Part* part){
109     dom(home, part->getHIntervals()[0], IntSet(IntArgs(PERFECT_CONSONANCES)));
110 }
111
112 void H2_2_arsisHarmoniesCannotBeDisonnant(Home home, Part* part){
113
114     for(int i = 0; i < part->getNMeasures()-1; i++){
115         if(i != part->getNMeasures()-2){ //if it is the penultimate measure
```

```
116             for(int d : DISONANCE){
117                 rel(home, part->getHIntervals()[(i*4)+2], IRT_EQ, d, Reify(part->
                      ↪ getIsDiminution()[i], RM_PMI));
118             }
119         }
120     }
121 }
122
123 void H2_3_disonanceImpliesDiminution(Home home, Part* part){
124     for(int i = 0; i < part->getIsDiminution().size(); i++){
125         BoolVar band1 = BoolVar(home, 0, 1);
126         BoolVar band2 = BoolVar(home, 0, 1);
127
128         rel(home, part->getConsonance()[(i*4)+2], BOT_OR, part->getIsDiminution()
              ↪ [i], 1);
129     }
130 }
131
132 void H3_1_endWithPerfectConsonance(Home home, Part* part){
133     dom(home, part->getHIntervals()[part->getHIntervals().size()-1], IntSet(
          ↪ IntArgs(PERFECT_CONSONANCES)));
134 }
135
136 void H3_2_penultimateNoteDomain(Home home, Part* part){
137     dom(home, expr(home, abs(part->getHIntervals()[part->getHIntervals().size()
          ↪ -5])), IntSet({UNISSON, PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH}));
138
139     rel(home, (part->getHIntervals()[part->getHIntervals().size()-5]!=
          ↪ PERFECT_FIFTH) >> (part->getPenultCostArray()[0]==part->getPenultCost
          ↪ ()));
140     rel(home, (part->getHIntervals()[part->getHIntervals().size()-5]==
          ↪ PERFECT_FIFTH) >> (part->getPenultCostArray()[0]==0));
141 }
142
143 void H3_3_cambiataCost(Home home, Part* part){
144     for(int i = 0; i < part->getCambiataCostArray().size(); i++){
145         rel(home, ((part->getThirdSpeciesHIntervals()[(i*4)+1]==UNISSON || part->
              ↪ getThirdSpeciesHIntervals()[(i*4)+1]==PERFECT_FIFTH)&&(part->
              ↪ getThirdSpeciesHIntervals()[(i*4)+2]==UNISSON || part->
              ↪ getThirdSpeciesHIntervals()[(i*4)+2]==PERFECT_FIFTH)
146             &&(abs(part->getThirdSpeciesMIntervals()[(i*4)+1])<=2)) >> (part->
                  ↪ getCambiataCostArray()[i]==part->getCambiataCost()));
147         rel(home, ((part->getThirdSpeciesHIntervals()[(i*4)+1]!=UNISSON && part->
              ↪ getThirdSpeciesHIntervals()[(i*4)+1]!=PERFECT_FIFTH)||(part->
              ↪ getThirdSpeciesHIntervals()[(i*4)+2]!=UNISSON && part->
              ↪ getThirdSpeciesHIntervals()[(i*4)+2]!=PERFECT_FIFTH)
148             ||(abs(part->getThirdSpeciesMIntervals()[(i*4)+1])>2)) >> (part->
                  ↪ getCambiataCostArray()[i]==0));
149     }
150 }
151
152 void H5_1_cpAndCfDifferentNotes(Home home, Part* part, Part* cf){
153     // cout << part->getFirstSpeciesNotes() << endl;
154     for(int i = 1; i < part->getFirstSpeciesNotes().size()-1; i++){
155         rel(home, part->getFirstSpeciesNotes()[i], IRT_NQ, cf->getNotes()[i]);
156     }
157 }
158
159 /**
160  * Modified by Tom Lai
161  * Check that voices do not play the same notes (except for the first and last
       ↪ measure)
162  * @param home
163  * @param parts vector of parts with parts[0] being the cantusFirmus
164  */
165 void H5_1_differentNotes(Home home, vector<Part*> parts){
166     // iterate over all pairs of parts
167     for(int v1 = 0; v1 < parts.size(); v1++){
```

```
168         for(int v2 = v1+1; v2 < parts.size(); v2++){
169             if(parts[v1]->getSpecies()==CANTUS_FIRMUS){
170                 // check voice v2 doesn't play same note as cantusFirmus
171                 for(int i = 1; i < parts[v1]->getNotes().size()-1; i++){
172                     rel(home, parts[v1]->getNotes()[i], IRT_NQ, parts[v2]->
                        ↪ getNotes()[i*4]);
173                 }
174             } else {
175                 // check voice v1 doesn't play same note as voice v2 with v1 and
                    ↪ v2 are counterpoints
176                 for(int i = 4; i < parts[v1]->getNotes().size()-1; i+=4){
177                     rel(home, parts[v1]->getNotes()[i], IRT_NQ, parts[v2]->
                        ↪ getNotes()[i]);
178                 }
179             }
180         }
181     }
182 }
183
184 void H6_1_preferImperfectConsonances(Home home, Part* part){
185     for(int i = 0; i < part->getHIntervals().size(); i++){
186         //set the octave cost
187         rel(home, part->getOctaveCostArray()[i], IRT_EQ, part->getHOctaveCost(),
              ↪ Reify(expr(home, part->getHIntervals()[i]==UNISSON), RM_PMI));
188         rel(home, part->getOctaveCostArray()[i], IRT_EQ, 0, Reify(expr(home, part
              ↪ ->getHIntervals()[i]!=UNISSON), RM_PMI));
189
190         //set the fifth cost
191         rel(home, part->getFifthCostArray()[i], IRT_EQ, part->getHFifthCost(),
              ↪ Reify(expr(home, part->getHIntervals()[i]==PERFECT_FIFTH), RM_PMI)
              ↪ );
192         rel(home, part->getFifthCostArray()[i], IRT_EQ, 0, Reify(expr(home, part
              ↪ ->getHIntervals()[i]!=PERFECT_FIFTH), RM_PMI));
193     }
194 }
195
196 void H7_1_2v_penultimateSixthOrThird(Home home, Part* part){
197     // If the part is not the lowest voice, then it must have a major sixth in
          ↪ the penultimate
198     IntVar penultimateInterval(home, -PERFECT_OCTAVE, PERFECT_OCTAVE);
199     if(part->getSpecies()==FIRST_SPECIES){
200         penultimateInterval = part->getHIntervals()[part->getHIntervals().size()
              ↪ -5];
201     } else if(part->getSpecies()==SECOND_SPECIES || part->getSpecies()==
          ↪ FOURTH_SPECIES){
202         penultimateInterval = part->getHIntervals()[part->getHIntervals().size()
              ↪ -3];
203     } else if(part->getSpecies()==THIRD_SPECIES){
204         penultimateInterval = part->getHIntervals()[part->getHIntervals().size()
              ↪ -2];
205     } else if(part->getSpecies()==FIFTH_SPECIES){
206         rel(home, expr(home, part->getSpeciesArray()[part->getSpeciesArray().size
              ↪ ()-3] == FOURTH_SPECIES) >>
207                 (penultimateInterval == part->getHIntervals()[part->getHIntervals
                      ↪ ().size()-3]));
208         rel(home, expr(home, part->getSpeciesArray()[part->getSpeciesArray().size
              ↪ ()-2] == THIRD_SPECIES) >>
209                 (penultimateInterval == part->getHIntervals()[part->getHIntervals
                      ↪ ().size()-2]));
210     }
211     rel(home, (part->getIsNotLowest()[part->getIsNotLowest().size()-2]==1) >> (
          ↪ penultimateInterval==MAJOR_SIXTH));
212 }
213
214 void H7_1_3v_penultimateSixthOrThird(Home home, Part* part){
215     // If the part is not the lowest voice, then it must have a major sixth in
          ↪ the penultimate measure
```

```
    // The penultimate measure is the second to last measure, so we check the
      ↪ second to last interval
    // We also check that the penultimate interval is a minor third, perfect
      ↪ fifth, or major sixth
    IntVar penultimateInterval(home, -PERFECT_OCTAVE, PERFECT_OCTAVE);
    if(part->getSpecies()==FIRST_SPECIES){
        penultimateInterval = part->getFirstSpeciesHIntervals()[part->
          ↪ getFirstSpeciesHIntervals().size()-2];
    } else if(part->getSpecies()==SECOND_SPECIES || part->getSpecies()==
      ↪ FOURTH_SPECIES){
        penultimateInterval = part->getSecondHInterval()[part->getSecondHInterval
          ↪ ().size()-2];
    } else if(part->getSpecies()==THIRD_SPECIES){
        penultimateInterval = part->getThirdSpeciesHIntervals()[part->
          ↪ getThirdSpeciesHIntervals().size()-2];
    } else if(part->getSpecies()==FIFTH_SPECIES){
        rel(home, expr(home, part->getSpeciesArray()[part->getSpeciesArray().size
          ↪ ()-3] == FOURTH_SPECIES) >>
             (penultimateInterval == part->getHIntervals()[part->getHIntervals
               ↪ ().size()-3]));
        rel(home, expr(home, part->getSpeciesArray()[part->getSpeciesArray().size
          ↪ ()-2] == THIRD_SPECIES) >>
             (penultimateInterval == part->getHIntervals()[part->getHIntervals
               ↪ ().size()-2]));
    }
    rel(home, (part->getIsNotLowest()[part->getIsNotLowest().size()-2]==1) >> (
      ↪ penultimateInterval==MAJOR_SIXTH || penultimateInterval==MINOR_THIRD
      ↪ || penultimateInterval==MAJOR_THIRD || penultimateInterval==
      ↪ PERFECT_FIFTH));
}

void H8_3v_preferHarmonicTriad(Home home, Part* part, IntVarArray triadCostArray,
  ↪   Stratum* upper1, Stratum* upper2){
    for(int i = 0; i < triadCostArray.size(); i++){
        BoolVar h1_3 = BoolVar(home, 0 ,1);         //check if the first interval
          ↪   is a minor third
        BoolVar h1_4 = BoolVar(home, 0 ,1);         //check if the first interval
          ↪   is a major third
        BoolVar h1_third = BoolVar(home, 0 ,1);     //check is the first interval
          ↪   is a third
        BoolVar h1_7 = BoolVar(home, 0 ,1);         //check if the first interval
          ↪   is a perfect fifth

        BoolVar h2_3 = BoolVar(home, 0 ,1);         //check if the second
          ↪ interval is a minor third
        BoolVar h2_4 = BoolVar(home, 0 ,1);         //check if the second
          ↪ interval is a major third
        BoolVar h2_third = BoolVar(home, 0 ,1);     //check if the second
          ↪ interval is a third
        BoolVar h2_7 = BoolVar(home, 0 ,1);         //check if the second
          ↪ interval is a perfect fifth

        BoolVar h_firstPoss = BoolVar(home, 0, 1);  //check if the first interval
          ↪   is a third and the second a perfect fifth
        BoolVar h_secondPoss = BoolVar(home, 0, 1); //check if the second
          ↪ interval is a third and the first a perfect fifth
        BoolVar triad = BoolVar(home, 0, 1);        //check if it is a triad
        BoolVar not_triad = BoolVar(home, 0, 1);    //check if it is not a triad

        //check for the first possibility
        rel(home, expr(home, abs(upper1->getHIntervals()[i*4])), IRT_EQ, 3, Reify
          ↪ (h1_3));
        rel(home, expr(home, abs(upper1->getHIntervals()[i*4])), IRT_EQ, 4, Reify
          ↪ (h1_4));
        rel(home, expr(home, abs(upper2->getHIntervals()[i*4])), IRT_EQ, 7, Reify
          ↪ (h2_7));
        rel(home, h1_3, BOT_OR, h1_4, h1_third);
        rel(home, h1_third, BOT_AND, h2_7, h_firstPoss);
```

```
257
258         //check for the second possibility
259         rel(home, expr(home, abs(upper2->getHIntervals()[i*4])), IRT_EQ, 3, Reify
              ↪  (h2_3));
260         rel(home, expr(home, abs(upper2->getHIntervals()[i*4])), IRT_EQ, 4, Reify
              ↪  (h2_4));
261         rel(home, expr(home, abs(upper1->getHIntervals()[i*4])), IRT_EQ, 7, Reify
              ↪  (h1_7));
262         rel(home, h2_3, BOT_OR, h2_4, h2_third);
263         rel(home, h2_third, BOT_AND, h1_7, h_secondPoss);
264
265         rel(home, h_firstPoss, BOT_OR, h_secondPoss, triad);   //is a triad if it
              ↪   is either the first or the second possibility
266         rel(home, triad, BOT_XOR, not_triad, 1);               //set not triad
267         rel(home, triadCostArray[i], IRT_EQ, 0, Reify(triad, RM_IMP));
268         rel(home, triadCostArray[i], IRT_EQ, part->getTriadCost(), Reify(
              ↪  not_triad, RM_IMP));
269     }
270 }
271
272 void H8_4v_preferHarmonicTriad(Home home, IntVarArray triadCostArray, Stratum*
      ↪  upper1, Stratum* upper2, Stratum* upper3){
273     // cout << upper1->getHIntervals().size() << endl;
274     // cout << upper2->getHIntervals().size() << endl;
275     // cout << upper3->getHIntervals().size() << endl;
276     for(int i = 0; i < triadCostArray.size(); i++){
277
278         IntVar H_b = upper1->getHIntervals()[i*4];
279         IntVar H_c = upper2->getHIntervals()[i*4];
280         IntVar H_d = upper3->getHIntervals()[i*4];
281
282
283         BoolVar H_b_is_third    = expr(home, H_b==MINOR_THIRD || H_b==MAJOR_THIRD
              ↪  );
284         BoolVar H_b_is_fifth    = expr(home, H_b==PERFECT_FIFTH);
285         BoolVar H_b_is_octave   = expr(home, H_b==UNISSON);
286
287         BoolVar H_c_is_third    = expr(home, H_c==MINOR_THIRD || H_c==MAJOR_THIRD
              ↪  );
288         BoolVar H_c_is_fifth    = expr(home, H_c==PERFECT_FIFTH);
289         BoolVar H_c_is_octave   = expr(home, H_c==UNISSON);
290
291         BoolVar H_d_is_third    = expr(home, H_d==MINOR_THIRD || H_d==MAJOR_THIRD
              ↪  );
292         BoolVar H_d_is_fifth    = expr(home, H_d==PERFECT_FIFTH);
293         BoolVar H_d_is_octave   = expr(home, H_d==UNISSON);
294
295
296         // worst case : not a harmonic triad with at least a third and a fifth
297         BoolVar no_fifth_or_no_third = expr(home, H_b_is_third + H_c_is_third +
              ↪  H_d_is_third == 0 || H_b_is_fifth + H_c_is_fifth + H_d_is_fifth ==
              ↪   0);
298         BoolVar note_outside_harmonic_triad = expr(home, H_b_is_third +
              ↪  H_b_is_fifth + H_b_is_octave == 0 || H_c_is_third + H_c_is_fifth +
              ↪   H_c_is_octave == 0 || H_d_is_third + H_d_is_fifth + H_d_is_octave
              ↪   == 0);
299
300         rel(home, (note_outside_harmonic_triad || no_fifth_or_no_third) >> (
              ↪  triadCostArray[i] == not_harmonic_triad_cost));
301
302         // now we are left with only combinations with at a third, a fifth, and
              ↪  another note of the harmonic triad.
303         // Doubling the fifth
304         rel(home, expr(home, (H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 2) &&
              ↪   !note_outside_harmonic_triad && !no_fifth_or_no_third) >> (
              ↪  triadCostArray[i] == double_fifths_cost));
305
```

```
306         // Doubling the third, and ensure it is the same type of third (not a
              ↪ major and a minor)
307         rel(home, expr(home, (H_b_is_third + H_c_is_third + H_d_is_third == 2) &&
              ↪  !note_outside_harmonic_triad && !no_fifth_or_no_third) >> (
              ↪ triadCostArray[i] == double_thirds_cost));
308         rel(home, (H_b_is_third && H_c_is_third) >> (H_b == H_c));
309         rel(home, (H_b_is_third && H_d_is_third) >> (H_b == H_d));
310         rel(home, (H_c_is_third && H_d_is_third) >> (H_c == H_d));
311 
312         // triad with octave
313         rel(home, (H_b_is_fifth && H_c_is_third && H_d_is_octave) >> (
              ↪ triadCostArray[i] == triad_with_octave_cost)); // 5 3 8
314         rel(home, (H_b_is_third && H_c_is_octave && H_d_is_fifth) >> (
              ↪ triadCostArray[i] == triad_with_octave_cost)); // 3 8 5
315         rel(home, (H_b_is_third && H_c_is_fifth && H_d_is_octave) >> (
              ↪ triadCostArray[i] == triad_with_octave_cost)); // 3 5 8
316         rel(home, (H_b_is_octave && H_c_is_third && H_d_is_fifth) >> (
              ↪ triadCostArray[i] == triad_with_octave_cost)); // 8 3 5
317         rel(home, (H_b_is_octave && H_c_is_fifth && H_d_is_third) >> (
              ↪ triadCostArray[i] == triad_with_octave_cost)); // 8 5 3
318 
319         // Best case : 5 8 3
320         rel(home, (H_b_is_fifth && H_c_is_octave && H_d_is_third) >> (
              ↪ triadCostArray[i] == 0));
321 
322     }
323 }
324 
325 void M1_1_2v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part){
326     rel(home, part->getFirstSpeciesMIntervals(), IRT_LQ, MINOR_SIXTH);
327     rel(home, part->getFirstSpeciesMIntervals(), IRT_GQ, -MINOR_SIXTH);
328 }
329 
330 void M1_1_3v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part){
331     dom(home, part->getMelodicIntervals(), IntSet({-UNISSON, -MINOR_SECOND, -
          ↪ MAJOR_SECOND, -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FOURTH, -TRITONE,
332             -PERFECT_FIFTH, -MINOR_SIXTH, -PERFECT_OCTAVE, UNISSON, MINOR_SECOND,
                  ↪  MAJOR_SECOND, MINOR_THIRD, MAJOR_THIRD, PERFECT_FOURTH,
                  ↪ TRITONE,
333             PERFECT_FIFTH, MINOR_SIXTH, PERFECT_OCTAVE}));
334 }
335 
336 void M1_2_octaveLeap(Home home, Part* part, Stratum* low){
337     for(int j = 0; j < part->getFirstSpeciesMIntervals().size(); j++){
338         rel(home, part->getFirstSpeciesMIntervals()[j], IRT_EQ, 12, Reify(expr(
              ↪ home,(abs(part->getFirstSpeciesHIntervals()[j])>=4)&&
339             (expr(home, expr(home, abs(low->getMelodicIntervals()[j])>0) == part
                  ↪ ->getIsNotLowest()[j])), RM_PMI));
340     }
341 }
342 
343 void M2_2_2v_twoConsecutiveNotesAreNotTheSame(Home home, Part* part){
344     rel(home, part->getSecondSpeciesMIntervals(), IRT_NQ, 0);
345 }
346 
347 void M2_2_3v_melodicIntervalsNotExceedMinorSixth(Home home, vector<Part*> parts,
      ↪  bool containsThirdSpecies){
348     for(int i = 1; i < parts.size(); i++){
349         if(parts[i]->getSpecies()==THIRD_SPECIES){
350             containsThirdSpecies=1;
351             break;
352         }
353     }
354     for(int p1 = 1; p1 < parts.size(); p1++){
355         for(int p2 = 1; p2 < parts.size(); p2++){
356             if(p1!=p2 && parts[p1]->getSpecies()==SECOND_SPECIES){
357                 if(!containsThirdSpecies){
```

```
                    for(int i = 0; i < parts[p1]->getBranchingNotes().size()-4; i
                      ↪  ++){
                        rel(home, parts[p1]->getMelodicIntervals().slice(0,
                          ↪  notesPerMeasure.at(SECOND_SPECIES), parts[p1]->
                          ↪  getMelodicIntervals().size())[i],
                            IRT_NQ, 0);
                    }
                } else {
                    for(int i = 0; i < parts[p1]->getBranchingNotes().size()-4; i
                      ↪  ++){
                        rel(home, parts[p1]->getMelodicIntervals().slice(0,
                          ↪  notesPerMeasure.at(SECOND_SPECIES), parts[p1]->
                          ↪  getMelodicIntervals().size())[i],
                            IRT_NQ, 0);
                    }
                    //no unison in the two last notes
                    rel(home, parts[p1]->getBranchingNotes()[parts[p1]->
                      ↪  getBranchingNotes().size()-2], IRT_NQ, parts[p1]->
                      ↪  getBranchingNotes()[parts[p1]->getBranchingNotes().
                      ↪  size()-1]);
                }
            }
        }
    }
}

void M2_1_varietyCost(Home home, vector<Part*> parts){
    for(int i = 1; i < parts.size(); i++){
        Part* p = parts[i];
        // cout << "Branching notes size : " << endl;
        // cout << p->getBranchingNotes().size() << endl;
        int temp = 0;
        IntVarArray notes = p->getBranchingNotes();
        for(int j = 0; j < p->getHIntervalSize()-1; j++){
            // cout << j << endl;
            int upbnd = 0;
            if(j+3<p->getHIntervalSize()){
                upbnd = j+4;
            } else {upbnd = p->getHIntervalSize();}
            for(int k = j+1; k < upbnd;k++){
                //setting a cost if notes inside a window are the same in a part
                rel(home, (notes[j]==notes[k])>>(p->getVarietyArray(temp)==p->
                  ↪  getVarietyCost()));
                rel(home, (notes[j]!=notes[k])>>(p->getVarietyArray(temp)==0));
                temp++;
            }

        }
    }
}

void P1_1_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
    for(int j = 0; j < part->getFirstSpeciesMotions().size(); j++){

        rel(home, ((part->getFirstSpeciesHIntervals()[j+1]==UNISSON || part->
          ↪  getFirstSpeciesHIntervals()[j+1]==PERFECT_FIFTH)&&part->
          ↪  getIsNotLowest()[j+1]==1) >>
            (part->getFirstSpeciesMotions()[j]!=PARALLEL_MOTION));

    }
}

void P1_1_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
    for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){

        //if the counterpoint is the lowest part -> the cost is 0
        rel(home, (!part->getIsNotLowest()[j]) >> (part->getDirectCostArray()[j
          ↪  ]==0));
```

```
413
414         //if the counterpoint is neither the lowest nor the highest part, and the
              ↪  condition is true, then add a cost of 2
415         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part
              ↪  ->getFirstSpeciesMotions()[j]==2&&(part->getFirstSpeciesHIntervals
              ↪  ()[j+1]==0||
416             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))==7))) >> (
                  ↪  part->getDirectCostArray()[j]==2));
417         //else the cost is 0
418         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part
              ↪  ->getFirstSpeciesMotions()[j]!=2||(part->getFirstSpeciesHIntervals
              ↪  ()[j+1]!=0
419             && expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))!=7))) >> (
                  ↪  part->getDirectCostArray()[j]==0));
420
421         //if the counterpoint is the highest part and the condition is true, then
              ↪  add a direct move cost (8 in this case)
422         rel(home, (part->getIsHighest()[j] && (part->getFirstSpeciesMotions()[j
              ↪  ]==2&&(part->getFirstSpeciesHIntervals()[j+1]==0||
423             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))==7))) >> (
                  ↪  part->getDirectCostArray()[j]==part->getDirectMoveCost()));
424         //else the cost is 0
425         rel(home, (part->getIsHighest()[j] && (part->getFirstSpeciesMotions()[j
              ↪  ]!=2||(part->getFirstSpeciesHIntervals()[j+1]!=0 &&
426             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))!=7))) >> (
                  ↪  part->getDirectCostArray()[j]==0));
427
428     }
429 }
430
431 void P1_2_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
432     for(int j = 0; j < part->getSecondSpeciesRealMotions().size(); j++){
433         rel(home, ((part->getFirstSpeciesHIntervals()[j+1]==UNISSON || part->
              ↪  getFirstSpeciesHIntervals()[j+1]==PERFECT_FIFTH)&&part->
              ↪  getIsNotLowest()[j+1]==1) >>
434             (part->getSecondSpeciesRealMotions()[j]!=PARALLEL_MOTION));
435     }
436 }
437
438 void P1_2_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
439
440     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
441         rel(home, (part->getSecondSpeciesRealMotions()[j]==2&&(part->
              ↪  getFirstSpeciesHIntervals()[j+1]==0||part->
              ↪  getFirstSpeciesHIntervals()[j+1]==7))>>
442             (part->getDirectCostArray()[j]==part->getDirectMoveCost()));
443         rel(home, (part->getSecondSpeciesRealMotions()[j]!=2||(part->
              ↪  getFirstSpeciesHIntervals()[j+1]!=0&&part->
              ↪  getFirstSpeciesHIntervals()[j+1]!=7))>>
444             (part->getDirectCostArray()[j]==0));
445     }
446
447 }
448
449 void P1_2_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
450     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
451
452         rel(home, (!part->getIsNotLowest()[j]) >> (part->getDirectCostArray()[j
              ↪  ]==0));
453
454         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part
              ↪  ->getSecondSpeciesRealMotions()[j]==2&&(part->
              ↪  getFirstSpeciesHIntervals()[j+1]==0||
455             part->getFirstSpeciesHIntervals()[j+1]==7))) >> (part->
                  ↪  getDirectCostArray()[j]==2));
456         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part
              ↪  ->getSecondSpeciesRealMotions()[j]!=2||(part->
              ↪  getFirstSpeciesHIntervals()[j+1]!=0 &&
```

```
457             part->getFirstSpeciesHIntervals()[j+1]!=7))) >> (part->
                  ↪ getDirectCostArray()[j]==0));
458
459         rel(home, (part->getIsHighest()[j] && (part->getSecondSpeciesRealMotions
                  ↪ ()[j]==2&&(part->getFirstSpeciesHIntervals()[j+1]==0||
460             part->getFirstSpeciesHIntervals()[j+1]==7))) >> (part->
                  ↪ getDirectCostArray()[j]==part->getDirectMoveCost()));
461         rel(home, (part->getIsHighest()[j] && (part->getSecondSpeciesRealMotions
                  ↪ ()[j]!=2||(part->getFirstSpeciesHIntervals()[j+1]!=0 &&
462             part->getFirstSpeciesHIntervals()[j+1]!=7))) >> (part->
                  ↪ getDirectCostArray()[j]==0));
463
464     }
465 }
466
467 void P3_0_noBattuta(Home home, Part* part){
468     for(int j = 0; j < part->getMotions().size(); j++){
469         rel(home, expr(home, part->getMotions()[j]==CONTRARY_MOTION && part->
                  ↪ getHIntervals()[j+1]==0 &&
470             part->getMelodicIntervals()[j]<-4), BOT_AND, part->getIsNotLowest()[j
                  ↪ ], 0);
471     }
472 }
473
474 void P3_1_noBattuta(Home home, Part* part){
475     for(int j = 0; j < part->getFirstSpeciesMotions().size(); j++){
476         rel(home, expr(home, part->getFirstSpeciesMotions()[j]==CONTRARY_MOTION
                  ↪ && part->getFirstSpeciesHIntervals()[j+1]==0 &&
477             part->getFirstSpeciesMIntervals()[j]<-4), BOT_AND, part->
                  ↪ getIsNotLowest()[j], 0);
478     }
479 }
480
481 void P3_2_noBattuta(Home home, Part* part){
482     for(int j = 0; j < part->getSecondSpeciesMotions().size(); j++){
483         rel(home, expr(home, part->getSecondSpeciesMotions()[j]==CONTRARY_MOTION
                  ↪ && part->getFirstSpeciesHIntervals()[j+1]==0 &&
484             part->getFirstSpeciesMIntervals()[j]<-4), BOT_AND, part->
                  ↪ getIsNotLowest()[j], 0);
485     }
486 }
487
488 void P4_successiveCost(Home home, vector<Part*> parts, int scc_cz, IntVarArray
      ↪ successiveCostArray, vector<Species> species){
489     int idx = 0;
490     for(int v1 = 1; v1 < parts.size(); v1++){
491         for(int v2 = v1+1; v2 < parts.size(); v2++){
492             IntVarArray hIntervals12 = IntVarArray(home, parts[v1]->getNMeasures
                      ↪ (), 0, MAJOR_SEVENTH);
493             BoolVarArray isPCons12 = BoolVarArray(home, parts[v1]->getNMeasures()
                      ↪ , 0, 1);
494
495             if(parts[v1]->getSpecies()==FOURTH_SPECIES || parts[v2]->getSpecies()
                      ↪ ==FOURTH_SPECIES){
496                 if(parts[v1]->getSpecies()==FOURTH_SPECIES && parts[v2]->
                          ↪ getSpecies()==FOURTH_SPECIES){
497                     for(int i = 0; i < parts[v1]->getNMeasures()-1; i++){
498                         rel(home, hIntervals12[i]==(abs(parts[v1]->getNotes()[i
                                  ↪ *4+2]-parts[v2]->getNotes()[i*4+2])%12));
499                     }
500                 } else if(parts[v1]->getSpecies()==FOURTH_SPECIES && parts[v2]->
                          ↪ getSpecies()!=FOURTH_SPECIES){
501                     for(int i = 0; i < parts[v1]->getNMeasures()-1; i++){
502                         rel(home, hIntervals12[i]==(abs(parts[v1]->getNotes()[i
                                  ↪ *4+2]-parts[v2]->getNotes()[i*4])%12));
503                     }
504                 } else{
505                     for(int i = 0; i < parts[v1]->getNMeasures()-1; i++){
```

```
506                         rel(home, hIntervals12[i]==(abs(parts[v1]->getNotes()[i
                              ↪ *4]-parts[v2]->getNotes()[i*4+2])%12));
507                     }
508                 }
509                 rel(home, hIntervals12[hIntervals12.size()-1]==(abs(parts[v1]->
                      ↪ getNotes()[parts[v1]->getNotes().size()-1]-
510                     parts[v2]->getNotes()[parts[v2]->getNotes().size()-1])%12));
511             } else {
512                 for(int i = 0; i < parts[v1]->getNMeasures(); i++){
513                     rel(home, hIntervals12[i]==(abs(parts[v1]->getNotes()[i*4]-
                          ↪ parts[v2]->getNotes()[i*4])%12));
514                 }
515             }
516
517             for(int i = 0; i < hIntervals12.size(); i++){
518                 rel(home, expr(home, hIntervals12[i]==UNISSON), BOT_OR, expr(home
                      ↪ , hIntervals12[i]==PERFECT_FIFTH), isPCons12[i]);
519             }
520
521             if(parts[v1]->getSpecies()!=SECOND_SPECIES && parts[v2]->getSpecies()
                  ↪ !=SECOND_SPECIES && parts[v1]->getSpecies()!=FOURTH_SPECIES &&
522                 parts[v2]->getSpecies()!=FOURTH_SPECIES)
523             {
524                 for(int i = 0; i < isPCons12.size()-1; i++){
525                     BoolVar succPCons = BoolVar(home, 0, 1);
526                     rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], succPCons);
527                     rel(home, (succPCons==1) >> (successiveCostArray[idx]==parts[
                          ↪ v1]->getSuccCost()));
528                     rel(home, (succPCons==0) >> (successiveCostArray[idx]==0));
529                     idx++;
530                 }
531             }
532             else if(parts[v1]->getSpecies()==SECOND_SPECIES){
533                 for(int i = 0; i < isPCons12.size()-1; i++){
534                     BoolVar firstNotFifth = BoolVar(home, 0, 1);
535                     BoolVar secondNotFifth = BoolVar(home, 0, 1);
536                     BoolVar notSuccessiveFifths = BoolVar(home, 0, 1);
537                     BoolVar succPCons = BoolVar(home, 0, 1);
538                     BoolVar succPConsAndNotSuccFifths = BoolVar(home, 0, 1);
539
540                     BoolVar mNotThird1 = BoolVar(home, 0, 1);
541                     BoolVar mNotThird2 = BoolVar(home, 0, 1);
542                     BoolVar mNotThird = BoolVar(home, 0, 1);
543                     BoolVar firstFifth = BoolVar(home, 0, 1);
544                     BoolVar secondFifth = BoolVar(home, 0, 1);
545                     BoolVar succFifth = BoolVar(home, 0, 1);
546                     BoolVar succFifthNotThird = BoolVar(home, 0, 1);
547
548                     BoolVar applyCost = BoolVar(home, 0, 1);
549
550                     rel(home, hIntervals12[i], IRT_NQ, 7, Reify(firstNotFifth));
551                     rel(home, hIntervals12[i+1], IRT_NQ, 7, Reify(secondNotFifth)
                          ↪ );
552                     rel(home, firstNotFifth, BOT_OR, secondNotFifth,
                          ↪ notSuccessiveFifths);
553
554                     rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], succPCons);
555                     rel(home, succPCons, BOT_AND, notSuccessiveFifths,
                          ↪ succPConsAndNotSuccFifths);
556
557                     rel(home, parts[v1]->getMelodicIntervals()[i*2], IRT_NQ, 3,
                          ↪ mNotThird1);
558                     rel(home, parts[v1]->getMelodicIntervals()[i*2], IRT_NQ, 4,
                          ↪ mNotThird2);
559                     rel(home, mNotThird1, BOT_AND, mNotThird2, mNotThird);
560
561                     rel(home, hIntervals12[i], IRT_EQ, 7, Reify(firstFifth));
562                     rel(home, hIntervals12[i+1], IRT_EQ, 7, Reify(secondFifth));
```

```
563                     rel(home, firstFifth, BOT_AND, secondFifth, succFifth);
564                     rel(home, mNotThird, BOT_AND, succFifth, succFifthNotThird);
565
566                     rel(home, succPConsAndNotSuccFifths, BOT_OR,
                          ↪ succFifthNotThird, applyCost);
567                     rel(home, (applyCost==1) >> (successiveCostArray[idx]==parts[
                          ↪ v1]->getSuccCost()));
568                     rel(home, (applyCost==0) >> (successiveCostArray[idx]==0));
569
570                     idx++;
571                 }
572             }
573             else if(parts[v2]->getSpecies()==SECOND_SPECIES){
574                 for(int i = 0; i < isPCons12.size()-1; i++){
575                     BoolVar firstNotFifth = BoolVar(home, 0, 1);
576                     BoolVar secondNotFifth = BoolVar(home, 0, 1);
577                     BoolVar notSuccessiveFifths = BoolVar(home, 0, 1);
578                     BoolVar succPCons = BoolVar(home, 0, 1);
579                     BoolVar succPConsAndNotSuccFifths = BoolVar(home, 0, 1);
580
581                     BoolVar mNotThird1 = BoolVar(home, 0, 1);
582                     BoolVar mNotThird2 = BoolVar(home, 0, 1);
583                     BoolVar mNotThird = BoolVar(home, 0, 1);
584                     BoolVar firstFifth = BoolVar(home, 0, 1);
585                     BoolVar secondFifth = BoolVar(home, 0, 1);
586                     BoolVar succFifth = BoolVar(home, 0, 1);
587                     BoolVar succFifthNotThird = BoolVar(home, 0, 1);
588
589                     BoolVar applyCost = BoolVar(home, 0, 1);
590
591                     rel(home, hIntervals12[i], IRT_NQ, 7, Reify(firstNotFifth));
592                     rel(home, hIntervals12[i+1], IRT_NQ, 7, Reify(secondNotFifth)
                          ↪ );
593                     rel(home, firstNotFifth, BOT_OR, secondNotFifth,
                          ↪ notSuccessiveFifths);
594
595                     rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], succPCons);
596                     rel(home, succPCons, BOT_AND, notSuccessiveFifths,
                          ↪ succPConsAndNotSuccFifths);
597
598                     rel(home, parts[v2]->getMelodicIntervals()[i*2], IRT_NQ, 3,
                          ↪ mNotThird1);
599                     rel(home, parts[v2]->getMelodicIntervals()[i*2], IRT_NQ, 4,
                          ↪ mNotThird2);
600                     rel(home, mNotThird1, BOT_AND, mNotThird2, mNotThird);
601
602                     rel(home, hIntervals12[i], IRT_EQ, 7, Reify(firstFifth));
603                     rel(home, hIntervals12[i+1], IRT_EQ, 7, Reify(secondFifth));
604                     rel(home, firstFifth, BOT_AND, secondFifth, succFifth);
605                     rel(home, mNotThird, BOT_AND, succFifth, succFifthNotThird);
606
607                     rel(home, succPConsAndNotSuccFifths, BOT_OR,
                          ↪ succFifthNotThird, applyCost);
608                     rel(home, (applyCost==1) >> (successiveCostArray[idx]==parts[
                          ↪ v1]->getSuccCost()));
609                     rel(home, (applyCost==0) >> (successiveCostArray[idx]==0));
610
611                     idx++;
612                 }
613             } else if(parts[v1]->getSpecies()==FOURTH_SPECIES || parts[v2]->
                  ↪ getSpecies()==FOURTH_SPECIES){
614                 for(int i = 0; i < hIntervals12.size()-1; i++){
615                     BoolVar firstNotFifth = BoolVar(home, 0, 1);
616                     BoolVar secondNotFifth = BoolVar(home, 0, 1);
617                     BoolVar notSuccessiveFifths = BoolVar(home, 0, 1);
618
619                     BoolVar succPCons = BoolVar(home, 0, 1);
620                     BoolVar succPConsNotFifths = BoolVar(home, 0, 1);
```

```
621
622                     rel(home, hIntervals12[i], IRT_NQ, 7, Reify(firstNotFifth));
623                     rel(home, hIntervals12[i+1], IRT_NQ, 7, Reify(secondNotFifth)
                          ↪ );
624                     rel(home, firstNotFifth, BOT_OR, secondNotFifth,
                          ↪ notSuccessiveFifths);
625
626                     rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], succPCons);
627                     rel(home, succPCons, BOT_AND, notSuccessiveFifths,
                          ↪ succPConsNotFifths);
628                     rel(home, (succPConsNotFifths==1) >> (successiveCostArray[idx
                          ↪ ]==parts[v1]->getSuccCost()));
629                     rel(home, (succPConsNotFifths==0) >> (successiveCostArray[idx
                          ↪ ]==0));
630                     idx++;
631                 }
632             }
633         }
634     }
635 }
636
637 void P6_3v_noMoveInSameDirection(Home home, vector<Part*> parts){
638     for(int i = 0; i < parts[0]->getFirstSpeciesMotions().size(); i++){
639         rel(home, expr(home, parts[0]->getFirstSpeciesMotions()[i]==2 && parts
              ↪ [1]->getFirstSpeciesMotions()[i]==2), BOT_AND, expr(home, parts
              ↪ [2]->getFirstSpeciesMotions()[i]==2), 0);
640     }
641 }
642
643 void P6_4v_noMoveInSameDirection(Home home, vector<Part*> parts){
644     for(int i = 0; i < parts[0]->getFirstSpeciesMotions().size(); i++){
645         rel(home, expr(home, parts[0]->getFirstSpeciesMotions()[i]==2 && parts
              ↪ [1]->getFirstSpeciesMotions()[i]==2 && parts[2]->
              ↪ getFirstSpeciesMotions()[i]==2), BOT_AND, expr(home, parts[3]->
              ↪ getFirstSpeciesMotions()[i]==2), 0);
646     }
647 }
648
649 void P7_noSuccessiveAscendingSixths(Home home, vector<Part*> parts){
650     for(int p1 = 0; p1 < parts.size(); p1++){
651         for(int p2 = p1+1; p2 < parts.size(); p2++){
652             for(int j = 1; j < parts[p1]->getFirstHInterval().size()-1; j++){
653                 rel(home, ((parts[p1]->getFirstHInterval()[j-1]!=MINOR_SIXTH &&
                      ↪ parts[p1]->getFirstHInterval()[j-1]!=MAJOR_SIXTH) &&
654                     (parts[p2]->getFirstHInterval()[j-1]!=MINOR_SIXTH && parts[p2
                          ↪ ]->getFirstHInterval()[j-1]!=MAJOR_SIXTH)) || (
655                         (parts[p1]->getFirstHInterval()[j]!=-MINOR_SIXTH && parts
                              ↪ [p1]->getFirstHInterval()[j]!=-MAJOR_SIXTH) &&
656                     (parts[p2]->getFirstHInterval()[j]!=- MINOR_SIXTH && parts[p2
                          ↪ ]->getFirstHInterval()[j]!=-MAJOR_SIXTH)) || (
657                         parts[p1]->getFirstMInterval()[j]>0 || parts[p2]->
                              ↪ getFirstMInterval()[j] > 0));
658             }
659         }
660     }
661 }
662
663 void P1_1_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
664     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
665         //set a cost when it is reached through direct motion, it is 0 when not
666         rel(home, (part->getFirstSpeciesMotions()[j]==2&&(part->
              ↪ getFirstSpeciesHIntervals()[j+1]==0||part->
              ↪ getFirstSpeciesHIntervals()[j+1]==7))>>
667             (part->getDirectCostArray()[j]==part->getDirectCost()));
668         rel(home, (part->getFirstSpeciesMotions()[j]!=2||(part->
              ↪ getFirstSpeciesHIntervals()[j+1]!=0&&part->
              ↪ getFirstSpeciesHIntervals()[j+1]!=7))>>
669             (part->getDirectCostArray()[j]==0));
```

```
    }
}
```

Utilities.cpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
// This file contains the implementations of general utility functions.
//

#include "../headers/Utilities.hpp"

vector<bool> activeConstraints = std::vector<bool>(consSize, false);

string get_constraint_name(int constraint) {
    if (constraint < 0 || constraint >= constraintNames.size()) {
        return "Unknown constraint";
    }
    return constraintNames[constraint];
}

/**
 * For a given set of intervals between notes that loops and a starting note, returns all the possible notes
 * @param note the starting note
 * @param intervals the set of intervals between notes. It must make a loop. For example, to get all notes from a major
 * scale from note, use {2, 2, 1, 2, 2, 2, 1}. To get all notes from a minor chord, use {3, 4, 5}.
 * @return vector<int> all the notes
 */
vector<int> get_all_notes_from_interval_loop(int n, vector<int> intervals)
{
    int note = n % PERFECT_OCTAVE; // bring the root back to [12,23] in case the argument is wrong
    vector<int> notes;

    int i = 0;
    while (note <= 127)
    {
        notes.push_back(note);
        note += intervals[i % intervals.size()];
        ++i;
    }
    return notes;
}

/**
 * For a given tonality (root + mode), returns all the possible notes
 * @param root the root of the tonality (in [0,11])
 * @param scale the set of tones and semitones that define the scale
 * @return vector<int> all the possible notes from that tonality
 */
vector<int> get_all_notes_from_scale(int root, vector<int> scale)
{
    return get_all_notes_from_interval_loop(root, scale);
}

/**
 * For a given chord (root + mode), returns all the possible notes
 * @param root the root of the chord
 * @param quality the set of tones and semitones that define the chord
 * @return vector<int> all the possible notes from that chord
 */
vector<int> get_all_notes_from_chord(int root, vector<int> quality)
```

```
{
    return get_all_notes_from_interval_loop(root, quality);
}

/**
 * Get all values for a given note
 * @param note a note
 * @return vector<int> a vector containing all the given notes
 */
vector<int> get_all_given_note(int note)
{
    int current = note % PERFECT_OCTAVE;
    vector<int> notes;
    while (current < 127)
    {
        notes.push_back(current);
        current += 12;
    }
    return notes;
}

/**
 * Transforms a vector of integers into a string
 * @param vector a vector of integers
 * @return string the string representation of the vector
 */
string int_vector_to_string(vector<int> vector){
    string s;
    for (int i = 0; i < vector.size(); ++i) {
        s += to_string(vector[i]);
        if(i != vector.size() - 1)
            s += ", ";
    }
    return s;
}

/**
 * Transforms an int* into a vector<int>
 * @param ptr an int* pointer
 * @param size the size of the array
 * @return a vector<int> containing the same values as the array
 */
vector<int> int_pointer_to_vector(int* ptr, int size){
    vector<int> v;
    for(int i = 0; i < size; i++){
        v.push_back(ptr[i]);
    }
    return v;
}

/**
 * Union algorithm found and adapted from GeeksForGeeks.com
 */
vector<int> vector_union(vector<int> v1, vector<int> v2){
    vector<int> v(v1.size()
                  + v2.size());
    vector<int>::iterator it, st;

    it = set_union(v1.begin(), v1.end(),
                   v2.begin(),
                   v2.end(), v.begin());

    vector<int> res = {};
    for(st = v.begin(); st != it; ++st){
        res.push_back(*st);
    }
    return res;
}
```

```
125
126 /**
127  * Intersection algorithm found and adapted from GeeksForGeeks.com
128  */
129 vector<int> vector_intersection(vector<int> v1, vector<int> v2){
130     vector<int> v(v1.size()
131                   + v2.size());
132     vector<int>::iterator it, st;
133
134     it = set_intersection(v1.begin(), v1.end(),
135                   v2.begin(),
136                   v2.end(), v.begin());
137
138     vector<int> res = {};
139     for(st = v.begin(); st != it; ++st){
140         res.push_back(*st);
141     }
142     return res;
143 }
144
145 /**
146  * Difference algorithm found and adapted from GeeksForGeeks.com
147  */
148 vector<int> vector_difference(vector<int> v1, int lb, int ub){
149     vector<int> v = {};
150     for(int i = lb; i <= ub; i++){
151         int t = 0;
152         for(int j = 0; j < v1.size(); j++){
153             if(i==v1[j]){
154                 t=1;
155                 break;
156             }
157         }
158         if(t==0){
159             v.push_back(i);
160         }
161     }
162     return v;
163 }
164
165 /**
166  * Prints the Search::Statistics object into a readable format
167  * @param stats a Search::Statistics object representing the statistics of a
    ↪  search
168  * @return The string representation of the statistics object
169  */
170 string statistics_to_string(Search::Statistics stats){
171     string s = "Nodes␣traversed:␣" + to_string(stats.node) + "\n";
172     s += "Failed␣nodes␣explored:␣" + to_string(stats.fail) + "\n";
173     s += "Restarts␣performed:␣" + to_string(stats.restart) + "\n";
174     s += "Propagators␣executed:␣" + to_string(stats.propagate) + "\n";
175     s += "No␣goods␣generated:␣" + to_string(stats.nogood) + "\n";
176     s += "Maximal␣depth␣of␣explored␣tree:␣" + to_string(stats.depth) + "\n";
177     return s;
178 }
179
180 /**
181  * Prints the Search::Statistics object into a csv format (coma separated)
182  * @param stats a Search::Statistics object representing the statistics of a
    ↪  search
183  * @return The string representation of the statistics object
184  */
185 string statistics_to_csv_string(Search::Statistics stats){
186     string s = to_string(stats.node) + ",";
187     s += to_string(stats.fail) + ",";
188     s += to_string(stats.restart) + ",";
189     s += to_string(stats.propagate) + ",";
190     s += to_string(stats.nogood) + ",";
```

```
191     s += to_string(stats.depth) + ",";
192     return s;
193 }
194 
195 /**
196  * Returns the value of a variable as a string
197  * @param var an integer variable
198  * @param absolute a boolean indicating if the value should be returned as an
    ↪  absolute value (default is false)
199  * @return a string representing the value of the variable
200  */
201 string intVar_to_string(const IntVar &var, bool absolute) {
202     if (var.assigned()){
203         if(absolute)
204             return to_string(abs(var.val()));
205         return to_string(var.val());
206     }
207     return "<not␣assigned>";
208 }
209 
210 /**
211  * Returns the values of an array of variables as a string
212  * @param vars an array of integer variables
213  * @return a string representing the values of the variables
214  */
215 string intVarArray_to_string(IntVarArray vars){
216     int s = vars.size();
217     string res = "{";
218     for(int i = 0; i < s; i++){
219         res += intVar_to_string(vars[i]);
220         if(i != s - 1)
221             res += ",␣";
222     }
223     res += "}";
224     return res;
225 }
226 
227 /**
228  * Returns the value of a variable as a string
229  * @param var an integer variable
230  * @param absolute a boolean indicating if the value should be returned as an
    ↪  absolute value (default is false)
231  * @return a string representing the value of the variable
232  */
233 string boolVar_to_string(const BoolVar &var, bool absolute) {
234     if (var.assigned()){
235         if(absolute)
236             return to_string(abs(var.val()));
237         return to_string(var.val());
238     }
239     return "<not␣assigned>";
240 }
241 
242 /**
243  * Returns the values of an array of variables as a string
244  * @param vars an array of integer variables
245  * @return a string representing the values of the variables
246  */
247 string boolVarArray_to_string(BoolVarArray vars){
248     int s = vars.size();
249     string res = "{";
250     for(int i = 0; i < s; i++){
251         res += boolVar_to_string(vars[i]);
252         if(i != s - 1)
253             res += ",␣";
254     }
255     res += "}";
256     return res;
```

```
}

/**
 * Returns the values of an array of variables as a string
 * @param vars an array of integer variables
 * @return a string representing the values of the variables, ignoring unassigned variables
 */
string cleanIntVarArray_to_string(IntVarArray vars){
    int s = vars.size();
    string res = "{";
    for(int i = 0; i < s; i++){
        if (vars[i].assigned()){
            res += intVar_to_string(vars[i]);
            if(i != s - 1)
                res += ", ";
        }
    }
    res += "}";
    return res;
}

/**
 * Returns the values of an IntVarArgs as a string
 * @param args an IntVarArgs
 * @return a string representing the values
 */
string intVarArgs_to_string(IntVarArgs args){
    int s = args.size();
    string res;
    for(int i = 0; i < s; i++){
        res += intVar_to_string(args[i], 0);
        if(i != s - 1)
            res += ", ";
    }
    return res;
}



/**
 * Returns the name of a note based on its MIDI value
 * @param note an integer
 */
string midi_to_letter(int note){
    return noteNames[note % PERFECT_OCTAVE];
}

/**
 * Returns the name of a mode based on its integer value
 * @param mode an integer
 * @return a string representing the name of the mode
 */
string mode_int_to_name(int mode){
    return modeNames[mode];
}

/**
 * returns a string with the time
 * @return a string with the time
 */
string time(){
    /// date and time for logs
    std::time_t currentTime = std::time(nullptr); // Get the current time
    std::string timeString = std::asctime(std::localtime(&currentTime)); // Convert to string
    return timeString;
}
```

```
323
324 /**
325  * Write a text into a log file
326  * Useful for debugging in the OM environment
327  * @param message the text to write
328  */
329 void write_to_log_file(const char *message, const string& filename) { // TODO :
      ↪ TO ACTIVATE LOGGING, UNCOOMMENT THE CONTENTS OF THIS FUNCTION AND THE NEXT
      ↪  FUNCTION.
330     // const char* homeDir = std::getenv("HOME"); // Get the user's home
          ↪ directory
331     // if (homeDir) {
332     //     std::string filePath(homeDir);
333     //     filePath += "/Documents/Libraries/MusicConstraints/out/" + filename;
          ↪ // Specify the desired file path, such as $HOME/log.txt
334
335     //     std::ofstream myfile(filePath, std::ios::app); // append mode
336     //     if (myfile.is_open()) {
337     //         myfile << message << endl;
338     //         myfile.close();
339     //     }
340     // }
341 }
342
343 /**
344  * Write a text into a log file
345  * @param message the text to write
346  */
347 void writeToLogFile(const char* message){ // TODO : TO ACTIVATE LOGGING,
      ↪ UNCOOMMENT THE CONTENTS OF THIS FUNCTION AND THE PREVIOUS FUNCTION.
348     // std::time_t currentTime = std::time(nullptr); // Get the current time
349     // std::string timeString = std::asctime(std::localtime(&currentTime)); //
          ↪ Convert to string
350
351     // const char* homeDir = std::getenv("HOME"); // Get the user's home
          ↪ directory
352     // if (homeDir) {
353     //     std::string filePath(homeDir);
354     //     filePath += "/log.txt"; // Specify the desired file path, such as
          ↪ $HOME/log.txt
355
356     //     std::ofstream myfile(filePath, std::ios::app); // append mode
357     //     if (myfile.is_open()) {
358     //         myfile <<timeString<< endl << message << endl;
359     //         myfile.close();
360     //     }
361     // }
362 }
```

CounterpointUtils.cpp

```
1     //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file contains the implementations of the main functions to create
      ↪ problems and counterpoints.
4 //
5
6 #include "../headers/CounterpointUtils.hpp"
7
8
9 Part* create_counterpoint(Home home, int species, int nMeasures, vector<int>
      ↪ cantusFirmus, int lowerBound, int upperBound, Stratum* low,
10     CantusFirmus* c, int v_type, vector<int> m_costs, vector<int> g_costs, vector
          ↪ <int> s_costs, int bm, int nV){
11     switch(nV) {
```

```
12          case TWO_VOICES :
13              switch (species) { /// call the appropriate constructor for the
                    ↪ counterpoint
14              case FIRST_SPECIES:
15                  return new FirstSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm, TWO_VOICES);
16                  break;
17              case SECOND_SPECIES:
18                  return new SecondSpeciesCounterpoint(home, nMeasures,
                        ↪ cantusFirmus, lowerBound, upperBound, low, c, v_type,
                        ↪ m_costs, g_costs, s_costs, bm, TWO_VOICES);
19                  break;
20              case THIRD_SPECIES:
21                  return new ThirdSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm, TWO_VOICES);
22                  break;
23              case FOURTH_SPECIES:
24                  return new FourthSpeciesCounterpoint(home, nMeasures,
                        ↪ cantusFirmus, lowerBound, upperBound, low, c, v_type,
                        ↪ m_costs, g_costs, s_costs, bm, TWO_VOICES);
25                  break;
26              case FIFTH_SPECIES:
27                  return new FifthSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm, TWO_VOICES);
28                  break;
29              default:
30                  throw std::invalid_argument("Species not implemented");
31              }
32              break;
33          case THREE_VOICES :
34              switch (species) { /// call the appropriate constructor for the
                    ↪ counterpoint
35              case FIRST_SPECIES:
36                  return new FirstSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm,
37                      TWO_VOICES, THREE_VOICES);
38                  break;
39              case SECOND_SPECIES:
40                  return new SecondSpeciesCounterpoint(home, nMeasures,
                        ↪ cantusFirmus, lowerBound, upperBound, low, c, v_type,
                        ↪ m_costs, g_costs, s_costs, bm,
41                      TWO_VOICES, THREE_VOICES);
42                  break;
43              case THIRD_SPECIES:
44                  return new ThirdSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm,
45                      TWO_VOICES, THREE_VOICES);
46                  break;
47              case FOURTH_SPECIES:
48                  return new FourthSpeciesCounterpoint(home, nMeasures,
                        ↪ cantusFirmus, lowerBound, upperBound, low, c, v_type,
                        ↪ m_costs, g_costs, s_costs, bm,
49                      TWO_VOICES, THREE_VOICES);
50                  break;
51              case FIFTH_SPECIES:
52                  return new FifthSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm,
53                      TWO_VOICES, THREE_VOICES);
54                  break;
55              default:
56                  throw std::invalid_argument("Species not implemented");
57              }
```

```
58              break;
59          case FOUR_VOICES :
60              switch (species) { /// call the appropriate constructor for the
                    ↪ counterpoint
61              case FIRST_SPECIES:
62                  return new FirstSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm,
63                      TWO_VOICES, THREE_VOICES, FOUR_VOICES);
64                  break;
65              case SECOND_SPECIES:
66                  return new SecondSpeciesCounterpoint(home, nMeasures,
                        ↪ cantusFirmus, lowerBound, upperBound, low, c, v_type,
                        ↪ m_costs, g_costs, s_costs, bm,
67                      TWO_VOICES, THREE_VOICES, FOUR_VOICES);
68                  break;
69              case THIRD_SPECIES:
70                  return new ThirdSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm,
71                      TWO_VOICES, THREE_VOICES, FOUR_VOICES);
72                  break;
73              case FOURTH_SPECIES:
74                  return new FourthSpeciesCounterpoint(home, nMeasures,
                        ↪ cantusFirmus, lowerBound, upperBound, low, c, v_type,
                        ↪ m_costs, g_costs, s_costs, bm,
75                      TWO_VOICES, THREE_VOICES, FOUR_VOICES);
76                  break;
77              case FIFTH_SPECIES:
78                  return new FifthSpeciesCounterpoint(home, nMeasures, cantusFirmus
                        ↪ , lowerBound, upperBound, low, c, v_type, m_costs, g_costs
                        ↪ , s_costs, bm,
79                      TWO_VOICES, THREE_VOICES, FOUR_VOICES);
80                  break;
81              default:
82                  throw std::invalid_argument("Species not implemented");
83              }
84              break;
85          default :
86              throw std::invalid_argument("Number of voices not implemented");
87      }
88  };
89
90
91  CounterpointProblem* create_problem(vector<int> cf, vector<Species> spList,
        ↪ vector<int> v_type, vector<int> m_costs, vector<int> g_costs,
92      vector<int> s_costs, vector<int> imp, int bm){
93
94      switch (spList.size())
95      {
96      case 1:
97          return new TwoVoiceCounterpoint(cf, spList[0], v_type[0], m_costs,
                ↪ g_costs, s_costs, imp, bm);
98          break;
99      case 2:
100         return new ThreeVoiceCounterpoint(cf, spList, v_type, m_costs, g_costs,
                ↪ s_costs, imp, bm);
101         break;
102     case 3:
103         return new FourVoiceCounterpoint(cf, spList, v_type, m_costs, g_costs,
                ↪ s_costs, imp, bm);
104         break;
105     default:
106         throw std::invalid_argument("The number of voices you asked for is not
                ↪ implemented (yet).");
107         break;
108     }
109 }
```

```
110
111 bool notInt(char* argv){
112     bool noInt = false;
113     for(int i = 0; i < strlen(argv); i++){
114         if(!isdigit(argv[i])){
115             noInt = true;
116             break;
117         }
118     }
119     return noInt;
120 }
121
122 bool has_solution(CounterpointProblem* problem) {
123     BAB<CounterpointProblem> e(problem);
124     if (e.next()) {
125         // cout << problem->getCantusFirmus()->getHIntervals() << endl;
126         // cout << problem->getCounterpoint_1()->getFirstSpeciesHIntervals() <<
              ↪ endl;
127         // cout << problem->getCounterpoint_2()->getFirstSpeciesHIntervals() <<
              ↪ endl;
128         return true;
129     }
130     return false;
131 }
```

fuxTest.cpp

```
1     //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // Modified by Tom Lai.
4 // This file contains the testing framework implementation.
5 //
6 #include <iostream>
7 #include <fstream>  // For file operations
8 #include <cmath>
9 #include "../headers/fuxTest.hpp"
10 #include <gecode/int.hh> // Ensure you include the necessary Gecode headers
11
12 // ========== Modified by Tom ====================
13
14 FuxTest::FuxTest(char* test){
15     cantusFirmus = {60,   62,   65,   64,   67,   65,   64,   62,   60};
16     cfSize = cantusFirmus.size();
17     melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
18     general_params = {4, 1, 1, 2, 2, 2, 8, 1};
19     specific_params = {8 , 4 , 0 , 2 , 1 , 8 , 50};
20     importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
21     borrowMode = 1;
22     if(strcmp(test, "all")==0){
23         cout << "Running all tests..." << endl;
24         test_1H1();
25         test_1H2();
26         test_1H3();
27         test_1H4();
28         test_1H5();
29         // test_1H6();
30         test_1H7();
31         test_1H10();
32         test_1M2();
33
34         test_1H2();
35         test_2M2();
36         cout << "Tests ended." << endl;
37     } else if(strcmp(test, "1H1")==0){
38         test_1H1();
```

```
39      } else if(strcmp(test, "1H2")==0){
40          test_1H2();
41      } else if(strcmp(test, "1H3")==0){
42          test_1H3();
43      } else if(strcmp(test, "1H4")==0){
44          test_1H4();
45      } else if(strcmp(test, "1H5")==0){
46          test_1H5();
47      // } else if(strcmp(test, "1H6")==0){
48      //     test_1H6();
49      } else if(strcmp(test, "1H7")==0){
50          test_1H7();
51      } else if(strcmp(test, "1H10")==0){
52          test_1H10();
53      } else if(strcmp(test, "1M2")==0){
54          test_1M2();
55      } else if(strcmp(test, "2H2")==0){
56          test_2H2();
57      } else if(strcmp(test, "2M2")==0){
58          test_2M2();
59      } else if(strcmp(test, "figs")==0){
60          test_2v_1sp_fig22();
61          test_2v_1sp_fig23();
62          test_2v_2sp_fig38();
63          test_2v_2sp_fig39();
64          test_2v_2sp_fig40();
65          test_2v_2sp_fig41();
66          test_2v_2sp_fig42();
67          test_2v_2sp_fig43();
68          test_2v_2sp_fig44();
69          test_2v_2sp_fig45();
70          test_2v_3sp_fig55();
71          test_2v_3sp_fig56();
72          test_2v_3sp_fig57();
73          test_2v_3sp_fig58();
74          test_2v_3sp_fig59();
75          test_2v_3sp_fig60();
76          test_2v_4sp_fig74();
77          test_2v_4sp_fig75();
78          test_2v_4sp_fig76();
79          test_2v_4sp_fig77();
80          test_2v_4sp_fig78();
81          test_3v_1sp_fig108();
82          test_3v_1sp_fig109();
83          test_3v_1sp_fig110();
84          test_3v_1sp_fig111();
85          test_3v_1sp_fig112();
86          test_3v_1sp_fig113();
87          test_3v_1sp_fig114();
88          test_3v_1sp_fig115();
89          test_3v_1sp_fig116();
90          test_3v_1sp_fig117();
91          test_3v_1sp_fig118();
92          test_3v_1sp_fig119();
93          test_3v_2sp_fig125();
94          test_3v_2sp_fig128();
95          test_3v_2sp_fig129();
96          test_3v_3sp_fig132();
97          test_3v_3sp_fig133();
98          test_4v_2sp_fig176();
99      } else {
100         std::invalid_argument("Test␣for␣constraint␣not␣found!");
101     }
102     //     CounterpointProblem* problem;
103     //     problem = dispatcher(test);
104     //     BAB<CounterpointProblem> e(problem);
105     //     int nb_sol = 0;
106     //     while(CounterpointProblem* pb = e.next()){
```

```
107     //         nb_sol++;
108     //         cout << pb->to_string() << endl;
109     //         delete pb;
110     //         if (nb_sol >= 1)
111     //             break;
112     //     }
113     //     if(nb_sol==0){
114     //         cout << "This constraint works. It prohibits a forbidden
          ↪  configuration." << endl;
115     //     } else {
116     //         cout << "This constraint does NOT work. It allows a forbidden
          ↪  configuration." << endl;
117     //     }
118     // }
119
120 }
121
122 /*
123 Chromatic melodies are forbidden for two and three voice composition, and to be
      ↪  avoided for four voice composition.
124 */
125 void FuxTest::test_G6() {
126     cout << "Start tests G6..." << endl;
127     test_G6_2v_1sp();
128     cout << "End tests G6." << endl;
129 }
130
131 void FuxTest::test_G6_2v_1sp() {
132     // TODO
133     // spList = {FIRST_SPECIES};
134     // v_type = {0};
135     // auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪  melodic_params, general_params, specific_params, importance,
          ↪  borrowMode);
136     // std::vector<CounterpointProblem*> solutions = get_all_solutions(problem);
137     // for (CounterpointProblem* solution: solutions) {
138     //     if (solution->getSolutionArray()[0].val() % 12 != cantusFirmus[0] %
          ↪  12) {
139     //         std::cerr   << "/!\\ ERROR test 1H4_2v_1sp: It exists solution but
          ↪   it shouldn't with the following configuration:\n"
140     //                     << "cantus firmus: ";
141     //         printVector(cantusFirmus);
142     //         std::cerr   << "solution array: ";
143     //         printIntVarArray(problem->getSolutionArray());
144     //         std::cerr   << "> First note of lowest stratum is not the tonic"
          ↪  << std::endl;
145     //     }
146     //     if (solution->getSolutionArray()[cfSize-1].val() % 12 != cantusFirmus
          ↪  [0] % 12) {
147     //         std::cerr   << "/!\\ ERROR test 1H4_2v_1sp: It exists solution but
          ↪   it shouldn't with the following configuration:\n"
148     //                     << "cantus firmus: ";
149     //         printVector(cantusFirmus);
150     //         std::cerr   << "solution array: ";
151     //         printIntVarArray(problem->getSolutionArray());
152     //         std::cerr   << "> Last note of lowest stratum is not the tonic" <<
          ↪   std::endl;
153     //     }
154     // }
155     // delete problem;
156 }
157
158 /*
159 All notes on the downbeat are consonant with the notes (on the downbeat) of the
      ↪  lowest stratum.
160 */
161 void FuxTest::test_1H1() {
162     cout << "Start tests 1H1..." << endl;
```

```
163     test_1H1_2v_1sp();
164     test_1H1_3v_1sp();
165     test_1H1_4v_1sp();
166     test_1H1_2v_2sp();
167     test_1H1_3v_2sp();
168     test_1H1_4v_2sp();
169     test_1H1_2v_3sp();
170     test_1H1_3v_3sp();
171     test_1H1_4v_3sp();
172     test_1H1_2v_4sp();
173     test_1H1_3v_4sp();
174     test_1H1_4v_4sp();
175     test_1H1_2v_5sp();
176     cout << "End␣tests␣1H1." << endl;
177 }
178 
179 void FuxTest::test_1H1_2v_1sp() {
180     cout << "Start␣test␣1H1_2v_1sp..." << endl;
181     // define dissonant intervals
182     int dis[] = {1, 2, 5, 6, 10, 11};
183     // define the species
184     spList = {FIRST_SPECIES};
185     // define the voice types (=pitch range):
186     // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
187     v_type = {3};
188     // Test that dissonant notes are forbidden
189     for (int interval : dis) {
190         for (size_t i = 0; i < cfSize; i++) {
191             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪  v_type, melodic_params, general_params, specific_params,
                  ↪  importance, borrowMode);
192             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
193             rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, note)
                  ↪  ; // fix the note i
194             if (has_solution(problem)) {
195                 std::cerr   << "/!\\␣ERROR␣test␣1H1_2v_1sp:␣It␣exists␣a␣solution␣
                      ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
196                             << "Cantus␣firmus:␣";
197                 printVector(cantusFirmus);
198                 std::cerr   << "Solution␣array:␣";
199                 printIntVarArray(problem->getSolutionArray());
200                 std::cerr   << ">␣Dissonant␣harmonic␣at␣the␣mesure␣" << i << std
                      ↪  ::endl;
201             }
202             delete problem;
203         }
204     }
205     cout << "End␣test␣1H1_2v_1sp..." << endl;
206 }
207 
208 void FuxTest::test_1H1_3v_1sp() {
209     cout << "Start␣test␣1H1_3v_1sp..." << endl;
210     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
211     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
212     spList = {FIRST_SPECIES, FIRST_SPECIES};
213     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
214     // Test that dissonant notes are forbidden
215     for (int interval : dis) {
216         for (size_t j = 0; j < 2; j++) { // iterate over each solution line
217             for (size_t i = 0; i < cfSize; i++) {
218                 CounterpointProblem* problem = create_problem(cantusFirmus,
                      ↪  spList, v_type, melodic_params, general_params,
                      ↪  specific_params, importance, borrowMode);
219                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
220                 rel(problem->getHome(), problem->getSolutionArray()[j*cfSize+i],
                      ↪  IRT_EQ, note); // fix the note i
221                 if (has_solution(problem)) {
```

```
222                     std::cerr   << "/!\\ ERROR test 1H1_3v_1sp: It exists a
                          ↪ solution but it shouldn't with the following
                          ↪ configuration:\n"
223                                 << "Cantus firmus: ";
224                     printVector(cantusFirmus);
225                     std::cerr   << "Solution array: ";
226                     printIntVarArray(problem->getSolutionArray());
227                     std::cerr   << "> Dissonant harmonic at the mesure " << i <<
                          ↪ std::endl;
228                 }
229                 delete problem;
230             }
231         }
232     }
233     cout << "End test 1H1_3v_1sp..." << endl;
234 }
235
236 void FuxTest::test_1H1_4v_1sp() {
237     cout << "Start test 1H1_4v_1sp..." << endl;
238     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
239     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
240     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
241     v_type = {3, 3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
242     // Test that dissonant notes are forbidden
243     for (int interval : dis) {
244         for (size_t j = 0; j < 2; j++) { // iterate over each solution line
245             for (size_t i = 0; i < cfSize; i++) {
246                 CounterpointProblem* problem = create_problem(cantusFirmus,
                      ↪ spList, v_type, melodic_params, general_params,
                      ↪ specific_params, importance, borrowMode);
247                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
248                 rel(problem->getHome(), problem->getSolutionArray()[j*cfSize+i],
                      ↪ IRT_EQ, note); // fix the note i
249                 if (has_solution(problem)) {
250                     std::cerr   << "/!\\ ERROR test 1H1_4v_1sp: It exists a
                          ↪ solution but it shouldn't with the following
                          ↪ configuration:\n"
251                                 << "Cantus firmus: ";
252                     printVector(cantusFirmus);
253                     std::cerr   << "Solution array: ";
254                     printIntVarArray(problem->getSolutionArray());
255                     std::cerr   << "> Dissonant harmonic at the mesure " << i <<
                          ↪ std::endl;
256                 }
257                 delete problem;
258             }
259         }
260     }
261     cout << "End test 1H1_4v_1sp..." << endl;
262 }
263
264 void FuxTest::test_1H1_2v_2sp() {
265     cout << "Start test 1H1_2v_2sp..." << endl;
266     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
267     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
268     spList = {SECOND_SPECIES};
269     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
270     // Test that dissonant notes are forbidden
271     for (int interval : dis) {
272         for (size_t i = 1; i < cfSize; i++) {
273             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪ v_type, melodic_params, general_params, specific_params,
                  ↪ importance, borrowMode);
274             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
275             rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ,
                  ↪ note); // fix the note i (downbeat)
276             if (has_solution(problem)) {
```

```
277                 std::cerr   << "/!\\ ERROR test 1H1_2v_2sp: It exists a solution
                      ↪  but it shouldn't with the following configuration:\n"
278                         << "Cantus firmus: ";
279                 printVector(cantusFirmus);
280                 std::cerr   << "Solution array: ";
281                 printIntVarArray(problem->getSolutionArray());
282                 std::cerr   << ">Dissonant harmonic at the mesure " << i << std
                      ↪  ::endl;
283             }
284             delete problem;
285         }
286     }
287     cout << "End test 1H1_2v_2sp..." << endl;
288 }
289
290 void FuxTest::test_1H1_3v_2sp() {
291     cout << "Start test 1H1_3v_2sp..." << endl;
292     int cpSize = cfSize*2-1; // size of a counterpoint
293     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
294     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
295     spList = {SECOND_SPECIES, SECOND_SPECIES};
296     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
297     // Test that dissonant notes are forbidden
298     for (int interval : dis) {
299         for (size_t j = 0; j < 2; j++) { // iterate over each solution line
300             for (size_t i = 1; i < cfSize; i++) {
301                 CounterpointProblem* problem = create_problem(cantusFirmus,
                      ↪  spList, v_type, melodic_params, general_params,
                      ↪  specific_params, importance, borrowMode);
302                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
303                 rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i
                      ↪  *2)], IRT_EQ, note); // fix the note i
304                 if (has_solution(problem)) {
305                     std::cerr   << "/!\\ ERROR test 1H1_3v_2sp: It exists a
                          ↪  solution but it shouldn't with the following
                          ↪  configuration:\n"
306                                 << "Cantus firmus: ";
307                     printVector(cantusFirmus);
308                     std::cerr   << "Solution array: ";
309                     printIntVarArray(problem->getSolutionArray());
310                     std::cerr   << ">Dissonant harmonic at the mesure " << i <<
                          ↪  std::endl;
311                 }
312                 delete problem;
313             }
314         }
315     }
316     cout << "End test 1H1_3v_2sp..." << endl;
317 }
318
319 void FuxTest::test_1H1_4v_2sp() {
320     cout << "Start test 1H1_4v_2sp..." << endl;
321     int cpSize = cfSize*2-1; // size of a counterpoint
322     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
323     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
324     spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
325     v_type = {3, 3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
326     // Test that dissonant notes are forbidden
327     for (int interval : dis) {
328         for (size_t j = 0; j < 3; j++) { // iterate over each solution line
329             for (size_t i = 1; i < cfSize; i++) {
330                 CounterpointProblem* problem = create_problem(cantusFirmus,
                      ↪  spList, v_type, melodic_params, general_params,
                      ↪  specific_params, importance, borrowMode);
331                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
332                 rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize)+(i
                      ↪  *2)], IRT_EQ, note); // fix the note i
333                 if (has_solution(problem)) {
```

```
334                     std::cerr    << "/!\\ ERROR test 1H1_4v_2sp: It exists a
                          ↪ solution but it shouldn't with the following
                          ↪ configuration:\n"
335                                  << "Cantus firmus: ";
336                     printVector(cantusFirmus);
337                     std::cerr    << "Solution array: ";
338                     printIntVarArray(problem->getSolutionArray());
339                     std::cerr    << "> Dissonant harmonic at the mesure " << i <<
                          ↪ std::endl;
340                 }
341                 delete problem;
342             }
343         }
344     }
345     cout << "End test 1H1_4v_2sp..." << endl;
346 }
347
348 void FuxTest::test_1H1_2v_3sp() {
349     cout << "Start test 1H1_2v_3sp..." << endl;
350     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
351     spList = {THIRD_SPECIES};
352     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
353     // Test that dissonant notes are forbidden
354     for (int interval : dis) {
355         for (size_t i = 0; i < cfSize; i++) {
356             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                      ↪ v_type, melodic_params, general_params, specific_params,
                      ↪ importance, borrowMode);
357             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
358             rel(problem->getHome(), problem->getSolutionArray()[i*4], IRT_EQ,
                      ↪ note); // fix the note i (downbeat)
359             if (has_solution(problem)) {
360                 std::cerr    << "/!\\ ERROR test 1H1_2v_2sp: It exists a solution
                          ↪ but it shouldn't with the following configuration:\n"
361                              << "Cantus firmus: ";
362                 printVector(cantusFirmus);
363                 std::cerr    << "Solution array: ";
364                 printIntVarArray(problem->getSolutionArray());
365                 std::cerr    << "> Dissonant harmonic at the mesure " << i << std
                          ↪ ::endl;
366             }
367             delete problem;
368         }
369     }
370     cout << "End test 1H1_2v_3sp..." << endl;
371 }
372
373 void FuxTest::test_1H1_3v_3sp() {
374     cout << "Start test 1H1_3v_3sp..." << endl;
375     int cpSize = cfSize*4-3; // size of a counterpoint
376     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
377     spList = {THIRD_SPECIES, THIRD_SPECIES};
378     v_type = {3, 3};
379     // Test that dissonant notes are forbidden
380     for (int interval : dis) {
381         for (size_t i = 0; i < cfSize; i++) {
382             for (size_t j = 0; j < 2; j++) {
383                 CounterpointProblem* problem = create_problem(cantusFirmus,
                          ↪ spList, v_type, melodic_params, general_params,
                          ↪ specific_params, importance, borrowMode);
384                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
385                 rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) +
                          ↪ (i * 4)], IRT_EQ, note); // fix the downbeat of each
                          ↪ mesure
386                 if (has_solution(problem)) {
387                     std::cerr    << "/!\\ ERROR test 1H1_3v_3sp: It exists a
                          ↪ solution but it shouldn't with the following
                          ↪ configuration:\n"
```

```
                                << "Cantus␣firmus:␣";
                    printVector(cantusFirmus);
                    std::cerr   << "Solution␣array:␣";
                    printIntVarArray(problem->getSolutionArray());
                    std::cerr   << ">␣Dissonant␣harmonic␣at␣the␣measure␣" << i <<
                      ↪  std::endl;
                }
                delete problem;
            }
        }
    }
    cout << "End␣test␣1H1_3v_3sp..." << endl;
}

void FuxTest::test_1H1_4v_3sp() {
    cout << "Start␣test␣1H1_4v_3sp..." << endl;
    int cpSize = cfSize*4-3; // size of a counterpoint
    int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
    spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
    v_type = {3, 3, 3};
    // Test that dissonant notes are forbidden
    for (int interval : dis) {
        for (size_t i = 0; i < cfSize; i++) {
            for (size_t j = 0; j < 3; j++) {
                CounterpointProblem* problem = create_problem(cantusFirmus,
                  ↪  spList, v_type, melodic_params, general_params,
                  ↪  specific_params, importance, borrowMode);
                int note = cantusFirmus[i] + 12 + interval; // note is dissonant
                rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) +
                  ↪  (i * 4)], IRT_EQ, note); // fix the downbeat of each
                  ↪  mesure
                if (has_solution(problem)) {
                    std::cerr   << "/!\\␣ERROR␣test␣1H1_4v_3sp:␣It␣exists␣a␣
                      ↪  solution␣but␣it␣shouldn't␣with␣the␣following␣
                      ↪  configuration:\n"
                                << "Cantus␣firmus:␣";
                    printVector(cantusFirmus);
                    std::cerr   << "Solution␣array:␣";
                    printIntVarArray(problem->getSolutionArray());
                    std::cerr   << ">␣Dissonant␣harmonic␣at␣the␣measure␣" << i <<
                      ↪  std::endl;
                }
                delete problem;
            }
        }
    }
    cout << "End␣test␣1H1_4v_3sp..." << endl;
}

void FuxTest::test_1H1_2v_4sp() {
    cout << "Start␣test␣1H1_2v_4sp..." << endl;
    int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
    spList = {FOURTH_SPECIES};
    v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
    // Test that dissonant notes are forbidden
    for (int interval : dis) {
        for (size_t i = 1; i < cfSize; i++) {
            CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪  v_type, melodic_params, general_params, specific_params,
              ↪  importance, borrowMode);
            if (i == cfSize-1) { // last note
                int note = cantusFirmus[i] + 12 + interval; // note is dissonant
                rel(problem->getHome(), problem->getSolutionArray()[(i*2)-1],
                  ↪  IRT_EQ, note); // fix the note i (downbeat)
            } else { // other notes
                int note = cantusFirmus[i] + 12 + interval; // note is dissonant
                rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ,
                  ↪  note); // fix the note i (downbeat)
```

```
            }
            if (has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H1_2v_4sp: It exists a solution
                  ↪  but it shouldn't with the following configuration:\n"
                            << "Cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "Solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Dissonant harmonic at the mesure " << i << std
                  ↪  ::endl;
            }
            delete problem;
        }
    }
    cout << "End test 1H1_2v_4sp..." << endl;
}

void FuxTest::test_1H1_3v_4sp() {
    cout << "Start test 1H1_3v_4sp..." << endl;
    int cpSize = cfSize*2-2; // size of a counterpoint
    int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
    spList = {FOURTH_SPECIES, FOURTH_SPECIES};
    v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
    // Test that dissonant notes are forbidden
    for (int interval : dis) {
        for (size_t i = 1; i < cfSize; i++) {
            for (size_t j = 0; j < 2; j++) {
                CounterpointProblem* problem = create_problem(cantusFirmus,
                  ↪  spList, v_type, melodic_params, general_params,
                  ↪  specific_params, importance, borrowMode);
                if (i == cfSize-1) { // last note
                    int note = cantusFirmus[i] + 12 + interval; // note is
                      ↪  dissonant
                    rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize
                      ↪  )+(i*2)-1], IRT_EQ, note); // fix the note i (downbeat
                      ↪  )
                } else { // other notes
                    int note = cantusFirmus[i] + 12 + interval; // note is
                      ↪  dissonant
                    rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize
                      ↪  )+(i*2)], IRT_EQ, note); // fix the note i (downbeat)
                }
                if (has_solution(problem)) {
                    std::cerr   << "/!\\ ERROR test 1H1_3v_4sp: It exists a
                      ↪  solution but it shouldn't with the following
                      ↪  configuration:\n"
                                << "Cantus firmus: ";
                    printVector(cantusFirmus);
                    std::cerr   << "Solution array: ";
                    printIntVarArray(problem->getSolutionArray());
                    std::cerr   << "> Dissonant harmonic at the mesure " << i <<
                      ↪  std::endl;
                }
                delete problem;
            }
        }
    }
    cout << "End test 1H1_3v_4sp..." << endl;
}

void FuxTest::test_1H1_4v_4sp() {
    cout << "Start test 1H1_4v_4sp..." << endl;
    int cpSize = cfSize*2-2; // size of a counterpoint
    int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
    spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
    v_type = {3, 3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
    // Test that dissonant notes are forbidden
    for (int interval : dis) {
```

```
500         for (size_t i = 1; i < cfSize; i++) {
501             for (size_t j = 0; j < 3; j++) {
502                 CounterpointProblem* problem = create_problem(cantusFirmus,
                      ↪ spList, v_type, melodic_params, general_params,
                      ↪ specific_params, importance, borrowMode);
503                 if (i == cfSize-1) { // last note
504                     int note = cantusFirmus[i] + 12 + interval; // note is
                          ↪ dissonant
505                     rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize
                          ↪ )+(i*2)-1], IRT_EQ, note); // fix the note i (downbeat
                          ↪ )
506                 } else { // other notes
507                     int note = cantusFirmus[i] + 12 + interval; // note is
                          ↪ dissonant
508                     rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize
                          ↪ )+(i*2)], IRT_EQ, note); // fix the note i (downbeat)
509                 }
510                 if (has_solution(problem)) {
511                     std::cerr   << "/!\\ ERROR test 1H1_4v_4sp: It exists a
                          ↪ solution but it shouldn't with the following
                          ↪ configuration:\n"
512                                 << "Cantus firmus: ";
513                     printVector(cantusFirmus);
514                     std::cerr   << "Solution array: ";
515                     printIntVarArray(problem->getSolutionArray());
516                     std::cerr   << "> Dissonant harmonic at the mesure " << i <<
                          ↪ std::endl;
517                 }
518                 delete problem;
519             }
520         }
521     }
522     cout << "End test 1H1_4v_4sp..." << endl;
523 }
524
525 void FuxTest::test_1H1_2v_5sp() {
526     cout << "Start test 1H1_2v_5sp..." << endl;
527     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
528     spList = {FIFTH_SPECIES};
529     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
530     // Test that dissonant notes are forbidden
531     for (int interval : dis) {
532         for (size_t i = 0; i < cfSize; i++) {
533             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪ v_type, melodic_params, general_params, specific_params,
                  ↪ importance, borrowMode);
534             int note = cantusFirmus[i] + 12 + interval; // note is dissonant
535             rel(problem->getHome(), problem->getSolutionArray()[i*4], IRT_EQ,
                  ↪ note); // fix the note i (downbeat)
536             if (has_solution(problem)) {
537                 std::cerr   << "/!\\ ERROR test 1H1_2v_5sp: It exists a solution
                      ↪ but it shouldn't with the following configuration:\n"
538                             << "Cantus firmus: ";
539                 printVector(cantusFirmus);
540                 std::cerr   << "Solution array: ";
541                 printIntVarArray(problem->getSolutionArray());
542                 std::cerr   << "> Dissonant harmonic at the mesure " << i << std
                      ↪ ::endl;
543             }
544             delete problem;
545         }
546     }
547     cout << "End test 1H1_2v_5sp..." << endl;
548 }
549
550 // void FuxTest::test_1H1_3v_5sp() {
551 //     cout << "Start test 1H1_3v_5sp..." << endl;
552 //     int cpSize = cfSize*2-2; // size of a counterpoint
```

```
553 //     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
554 //     spList = {FOURTH_SPECIES, FOURTH_SPECIES};
555 //     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
556 //     // Test that dissonant notes are forbidden
557 //     for (int interval : dis) {
558 //         for (size_t i = 1; i < cfSize; i++) {
559 //             for (size_t j = 0; j < 2; j++) {
560 //                 CounterpointProblem* problem = create_problem(cantusFirmus,
      ↪ spList, v_type, melodic_params, general_params, specific_params,
      ↪ importance, borrowMode);
561 //                 if (i == cfSize-1) { // last note
562 //                     int note = cantusFirmus[i] + 12 + interval; // note is
      ↪ dissonant
563 //                     rel(problem->getHome(), problem->getSolutionArray()[(j*
      ↪ cpSize)+(i*2)-1], IRT_EQ, note); // fix the note i (downbeat)
564 //                 } else { // other notes
565 //                     int note = cantusFirmus[i] + 12 + interval; // note is
      ↪ dissonant
566 //                     rel(problem->getHome(), problem->getSolutionArray()[(j*
      ↪ cpSize)+(i*2)], IRT_EQ, note); // fix the note i (downbeat)
567 //                 }
568 //                 if (has_solution(problem)) {
569 //                     std::cerr   << "/!\\ ERROR test 1H1_3v_4sp: It exists a
      ↪ solution but it shouldn't with the following configuration:\n"
570 //                                 << "Cantus firmus: ";
571 //                     printVector(cantusFirmus);
572 //                     std::cerr   << "Solution array: ";
573 //                     printIntVarArray(problem->getSolutionArray());
574 //                     std::cerr   << "> Dissonant harmonic at the mesure " << i
      ↪ << std::endl;
575 //                 }
576 //                 delete problem;
577 //             }
578 //         }
579 //     }
580 //     cout << "End test 1H1_3v_5sp..." << endl;
581 // }
582
583
584 /*
585 In two voice composition, the first harmonic interval must be a perfect
      ↪ consonance.
586 */
587 void FuxTest::test_1H2() {
588     cout << "Start tests 1H2..." << endl;
589     test_1H2_2v_1sp();
590     test_1H2_2v_2sp();
591     test_1H2_2v_3sp();
592     test_1H2_2v_4sp();
593     cout << "End tests 1H2." << endl;
594 }
595
596 void FuxTest::test_1H2_2v_1sp() {
597     cout << "Start test 1H2_2v_1sp..." << endl;
598     int p_cons[] = {0, 7}; // perfect conssonant intervals
599     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect
        ↪ conssonant intervals
600     spList = {FIRST_SPECIES};
601     v_type = {3};
602     for (int interval : p_cons) {
603         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
          ↪ v_type, melodic_params, general_params, specific_params,
          ↪ importance, borrowMode);
604         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ,
          ↪ cantusFirmus[0]+12+interval); // fix the first solution note as
          ↪ perfect consonant with cf
605         if (!has_solution(problem)) {
```

```
606             std::cerr   << "/!\\ ERROR test 1H2_2v_1sp: It doesn't exist a
                  ↪  solution but it should with the following configuration:\n"
607                     << "Cantus firmus: ";
608             printVector(cantusFirmus);
609             std::cerr   << "Solution array: ";
610             printIntVarArray(problem->getSolutionArray());
611             std::cerr   << "> First harmonic interval should be correct" << std::
                  ↪  endl;
612         }
613         delete problem;
614     }
615     for (int interval : non_p_cons) {
616         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪  v_type, melodic_params, general_params, specific_params,
              ↪  importance, borrowMode);
617         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ,
              ↪  cantusFirmus[0]+12+interval); // fix the first solution note as
              ↪  non-perfect consonant with cf
618         if (has_solution(problem)) {
619             std::cerr   << "/!\\ ERROR test 1H2_2v_1sp: It exists a solution but
                  ↪  it shouldn't with the following configuration:\n"
620                     << "Cantus firmus: ";
621             printVector(cantusFirmus);
622             std::cerr   << "Solution array: ";
623             printIntVarArray(problem->getSolutionArray());
624             std::cerr   << "> Non perfect consonance at the first mesure" << std
                  ↪  ::endl;
625         }
626         delete problem;
627     }
628     cout << "End test 1H2_2v_1sp..." << endl;
629 }
630
631 void FuxTest::test_1H2_2v_2sp() {
632     cout << "Start test 1H2_2v_2sp..." << endl;
633     int p_cons[] = {0, 7}; // perfect conssonant intervals
634     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect
          ↪  conssonant intervals
635     spList = {SECOND_SPECIES};
636     v_type = {3};
637     for (int interval : p_cons) {
638         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪  v_type, melodic_params, general_params, specific_params,
              ↪  importance, borrowMode);
639         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ,
              ↪  cantusFirmus[0]+12+interval); // fix the first solution note as
              ↪  perfect consonant with cf
640         if (!has_solution(problem)) {
641             std::cerr   << "/!\\ ERROR test 1H2_2v_2sp: It doesn't exist a
                  ↪  solution but it should with the following configuration:\n"
642                     << "Cantus firmus: ";
643             printVector(cantusFirmus);
644             std::cerr   << "Solution array: ";
645             printIntVarArray(problem->getSolutionArray());
646             std::cerr   << "> First harmonic interval should be correct" << std::
                  ↪  endl;
647         }
648         delete problem;
649     }
650     for (int interval : non_p_cons) {
651         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪  v_type, melodic_params, general_params, specific_params,
              ↪  importance, borrowMode);
652         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ,
              ↪  cantusFirmus[0]+12+interval); // fix the first solution note as
              ↪  non-perfect consonant with cf
653         if (has_solution(problem)) {
```

```
654             std::cerr   << "/!\\ ERROR test 1H2_2v_2sp: It exists a solution but
                  ↪  it shouldn't with the following configuration:\n"
655                     << "Cantus firmus: ";
656             printVector(cantusFirmus);
657             std::cerr   << "Solution array: ";
658             printIntVarArray(problem->getSolutionArray());
659             std::cerr   << ">Non perfect consonance at the first mesure" << std
                  ↪  ::endl;
660         }
661         delete problem;
662     }
663     cout << "End test 1H2_2v_2sp..." << endl;
664 }
665
666 void FuxTest::test_1H2_2v_3sp() {
667     cout << "Start test 1H2_2v_3sp..." << endl;
668     int p_cons[] = {0, 7}; // perfect conssonant intervals
669     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect
          ↪  conssonant intervals
670     spList = {THIRD_SPECIES};
671     v_type = {3};
672     // perfect conssonant are allowed
673     for (int interval : p_cons) {
674         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪  v_type, melodic_params, general_params, specific_params,
              ↪  importance, borrowMode);
675         int note = cantusFirmus[0] + 12 + interval; // note is perfect consonant
676         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
              ↪   fix the first solution note as perfect consonant with cf
677         if (!has_solution(problem)) {
678             std::cerr   << "/!\\ ERROR test 1H2_2v_3sp: It doesn't exist a
                  ↪  solution but it should with the following configuration:\n"
679                     << "Cantus firmus: ";
680             printVector(cantusFirmus);
681             std::cerr   << "Solution array: ";
682             printIntVarArray(problem->getSolutionArray());
683             std::cerr   << ">First harmonic interval should be correct" << std::
                  ↪  endl;
684         }
685         delete problem;
686     }
687     // non-perfect consonant are forbidden
688     for (int interval : non_p_cons) {
689         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪  v_type, melodic_params, general_params, specific_params,
              ↪  importance, borrowMode);
690         int note = cantusFirmus[0] + 12 + interval; // note is non-perfect
              ↪  consonant
691         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
              ↪   fix the first solution note as non-perfect consonant with cf
692         if (has_solution(problem)) {
693             std::cerr   << "/!\\ ERROR test 1H2_2v_3sp: It exists a solution but
                  ↪  it shouldn't with the following configuration:\n"
694                     << "Cantus firmus: ";
695             printVector(cantusFirmus);
696             std::cerr   << "Solution array: ";
697             printIntVarArray(problem->getSolutionArray());
698             std::cerr   << ">Non perfect consonance at the first mesure" << std
                  ↪  ::endl;
699         }
700         delete problem;
701     }
702     cout << "End test 1H2_2v_3sp..." << endl;
703 }
704
705 void FuxTest::test_1H2_2v_4sp() {
706     cout << "Start test 1H2_2v_4sp..." << endl;
707     int p_cons[] = {0, 7}; // perfect conssonant intervals
```

```
    int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect conssonant intervals
    spList = {FOURTH_SPECIES};
    v_type = {3};
    // perfect conssonant are allowed
    for (int interval : p_cons) {
        CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + 12 + interval; // note is perfect consonant
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first solution note as perfect consonant with cf
        if (!has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H2_2v_4sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First harmonic interval should be correct" << std::endl;
        }
        delete problem;
    }
    // non-perfect consonant are forbidden
    for (int interval : non_p_cons) {
        CounterpointProblem* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + 12 + interval; // note is non-perfect consonant
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first solution note as non-perfect consonant with cf
        if (has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H2_2v_4sp: It exists a solution but it shouldn't with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Non perfect consonance at the first mesure" << std::endl;
        }
        delete problem;
    }
    cout << "End test 1H2_2v_4sp..." << endl;
}


/*
In two voice composition, the last harmonic in-terval must be a perfect consonance.
When composing for three or four voices, the last chord should be composed only of notes of the harmonic triad.
*/
void FuxTest::test_1H3() {
    cout << "Start tests 1H3..." << endl;
    test_1H3_2v_1sp();
    test_1H3_3v_1sp();
    test_1H3_4v_1sp();
    test_1H3_2v_2sp();
    test_1H3_3v_2sp();
    test_1H3_4v_2sp();
    test_1H3_2v_3sp();
    test_1H3_3v_3sp();
    test_1H3_4v_3sp();
    test_1H3_2v_4sp();
    test_1H3_3v_4sp();
```

```
762     test_1H3_4v_4sp();
763     cout << "End tests 1H3." << endl;
764 }
765
766 void FuxTest::test_1H3_2v_1sp() {
767     cout << "Start test 1H3_2v_1sp..." << endl;
768     int p_cons[] = {0, 7}; // perfect conssonant intervals
769     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect
        ↪ conssonant intervals
770     spList = {FIRST_SPECIES};
771     v_type = {0};
772     // non-perfect consonant intervals are forbidden
773     for (int interval : non_p_cons) {
774         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
            ↪ v_type, melodic_params, general_params, specific_params,
            ↪ importance, borrowMode);
775         rel(problem->getHome(), problem->getSolutionArray()[cfSize-1], IRT_EQ,
            ↪ cantusFirmus[cfSize-1]+interval); // fix the last note with a not
            ↪ perfect consonance
776         if (has_solution(problem)) {
777             std::cerr   << "/!\\ ERROR test 1H3_2v_1sp: It exists solution but it
                ↪ shouldn't with the following configuration:\n"
778                         << "cantus firmus: ";
779             printVector(cantusFirmus);
780             std::cerr   << "solution array: ";
781             printIntVarArray(problem->getSolutionArray());
782             std::cerr   << "> Last harmonic interval must be a perfect consonance
                ↪ " << std::endl;
783         }
784         delete problem;
785     }
786     cout << "End test 1H3_2v_1sp..." << endl;
787 }
788
789 void FuxTest::test_1H3_3v_1sp() {
790     cout << "Start test 1H3_3v_1sp..." << endl;
791     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
792     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad
        ↪ intervals
793     spList = {FIRST_SPECIES, FIRST_SPECIES};
794     v_type = {3, 3};
795     // non-perfect consonant intervals are forbidden
796     for (size_t i = 0; i < 2; i++) { // iterate over each solution line
797         for (int interval : non_triad_cons) {
798             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                ↪ v_type, melodic_params, general_params, specific_params,
                ↪ importance, borrowMode);
799             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*i)+cfSize
                ↪ -1], IRT_EQ, cantusFirmus[cfSize-1] + 12 + interval); // fix
                ↪ the last note as a note not in harmonic triad intervals
800             if (has_solution(problem)) {
801                 std::cerr   << "/!\\ ERROR test 1H3_3v_1sp: It exists solution
                    ↪ but it shouldn't with the following configuration:\n"
802                             << "cantus firmus: ";
803                 printVector(cantusFirmus);
804                 std::cerr   << "solution array: ";
805                 printIntVarArray(problem->getSolutionArray());
806                 std::cerr   << "> Last harmonic intervals must be in harmonic
                    ↪ triad" << std::endl;
807             }
808             delete problem;
809         }
810     }
811     // perfect consonant intervals are allowed
812     for (int interval : triad_cons) {
813         for (size_t i = 0; i < 2; i++) { // iterate over each solution line
814             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                ↪ v_type, melodic_params, general_params, specific_params,
```

```
                ↪ importance, borrowMode);
815             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*i)+cfSize
                  ↪ -1], IRT_EQ, cantusFirmus[cfSize-1] + 12 + interval); // fix
                  ↪ the last note as a note in harmonic triad intervals
816             if (!has_solution(problem)) {
817                 std::cerr   << "/!\\ ERROR test 1H3_3v_1sp: It doesn't exist
                      ↪ solution but it should with the following configuration:\n
                      ↪ "
818                             << "cantus firmus: ";
819                 printVector(cantusFirmus);
820                 std::cerr   << "solution array: ";
821                 printIntVarArray(problem->getSolutionArray());
822                 std::cerr   << "> Last harmonic intervals must be in harmonic
                      ↪ triad" << std::endl;
823             }
824             delete problem;
825         }
826     }
827     cout << "End test 1H3_3v_1sp..." << endl;
828 }
829
830 void FuxTest::test_1H3_4v_1sp() {
831     cout << "Start test 1H3_4v_1sp..." << endl;
832     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
833     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad
          ↪ intervals
834     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
835     v_type = {3, 3, 3};
836     // non-perfect consonant intervals are forbidden
837     for (size_t i = 0; i < 3; i++) { // iterate over each solution line
838         for (int interval : non_triad_cons) {
839             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪ v_type, melodic_params, general_params, specific_params,
                  ↪ importance, borrowMode);
840             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*i)+cfSize
                  ↪ -1], IRT_EQ, cantusFirmus[cfSize-1]+12+interval); // fix the
                  ↪ last note as a note not in harmonic triad intervals
841             if (has_solution(problem)) {
842                 std::cerr   << "/!\\ ERROR test 1H3_4v_1sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
843                             << "cantus firmus: ";
844                 printVector(cantusFirmus);
845                 std::cerr   << "solution array: ";
846                 printIntVarArray(problem->getSolutionArray());
847                 std::cerr   << "> Last harmonic intervals must be in harmonic
                      ↪ triad" << std::endl;
848             }
849             delete problem;
850         }
851     }
852     // perfect consonant intervals are allowed
853     for (int interval : triad_cons) {
854         for (size_t i = 0; i < 3; i++) { // iterate over each solution line
855             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪ v_type, melodic_params, general_params, specific_params,
                  ↪ importance, borrowMode);
856             rel(problem->getHome(), problem->getSolutionArray()[(cfSize*i)+cfSize
                  ↪ -1], IRT_EQ, cantusFirmus[cfSize-1] + +12 + interval); // fix
                  ↪ the last note as a note in harmonic triad intervals
857             if (!has_solution(problem)) {
858                 std::cerr   << "/!\\ ERROR test 1H3_4v_1sp: It doesn't exist
                      ↪ solution but it should with the following configuration:\n
                      ↪ "
859                             << "cantus firmus: ";
860                 printVector(cantusFirmus);
861                 std::cerr   << "solution array: ";
862                 printIntVarArray(problem->getSolutionArray());
```

```
                std::cerr   << ">␣Last␣harmonic␣intervals␣must␣be␣in␣harmonic␣
                  ↪  triad" << std::endl;
            }
            delete problem;
        }
    }
    cout << "End␣test␣1H3_4v_1sp..." << endl;
}

void FuxTest::test_1H3_2v_2sp() {
    cout << "Start␣test␣1H3_2v_2sp..." << endl;
    int cpSize = cfSize*2-1; // size of a counterpoint
    int p_cons[] = {0, 7}; // perfect conssonant intervals
    int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect
      ↪  conssonant intervals
    spList = {SECOND_SPECIES};
    v_type = {0};
    // non-perfect consonant intervals are forbidden
    for (int interval : non_p_cons) {
        CounterpointProblem* problem = create_problem(cantusFirmus, spList,
          ↪  v_type, melodic_params, general_params, specific_params,
          ↪  importance, borrowMode);
        int note = cantusFirmus[cfSize-1] + interval;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
          ↪  note); // fix the last note with a not perfect consonance
        if (has_solution(problem)) {
            std::cerr   << "/!\\␣ERROR␣test␣1H3_2v_2sp:␣It␣exists␣solution␣but␣it
              ↪  ␣shouldn't␣with␣the␣following␣configuration:\n"
                    << "cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr   << "solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << ">␣Last␣harmonic␣interval␣must␣be␣a␣perfect␣consonance
              ↪  " << std::endl;
        }
        delete problem;
    }
    cout << "End␣test␣1H3_2v_1sp..." << endl;
}

void FuxTest::test_1H3_3v_2sp() {
    cout << "Start␣test␣1H3_3v_2sp..." << endl;
    int cpSize = cfSize*2-1; // size of a counterpoint
    int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
    int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad
      ↪  intervals
    spList = {SECOND_SPECIES, SECOND_SPECIES};
    v_type = {0, 0};
    // non-perfect consonant intervals are forbidden
    for (size_t i = 0; i < 2; i++) { // iterate over each solution line
        for (int interval : non_triad_cons) {
            CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪  v_type, melodic_params, general_params, specific_params,
              ↪  importance, borrowMode);
            int note = cantusFirmus[cfSize-1] + interval;
            rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize
              ↪  -1], IRT_EQ, note); // fix the last note as a note not in
              ↪  harmonic triad intervals
            if (has_solution(problem)) {
                std::cerr   << "/!\\␣ERROR␣test␣1H3_3v_2sp:␣It␣exists␣solution␣
                  ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                        << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << ">␣Last␣harmonic␣intervals␣must␣be␣in␣harmonic␣
                  ↪  triad" << std::endl;
            }
```

```
917             delete problem;
918         }
919     }
920     cout << "End test 1H3_3v_2sp..." << endl;
921 }
922
923 void FuxTest::test_1H3_4v_2sp() {
924     cout << "Start test 1H3_4v_2sp..." << endl;
925     int cpSize = cfSize*2-1; // size of a counterpoint
926     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
927     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad
        ↪ intervals
928     spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
929     v_type = {0, 0};
930     // non-perfect consonant intervals are forbidden
931     for (size_t i = 0; i < 3; i++) { // iterate over each solution line
932         for (int interval : non_triad_cons) {
933             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                ↪ v_type, melodic_params, general_params, specific_params,
                ↪ importance, borrowMode);
934             int note = cantusFirmus[cfSize-1] + interval;
935             rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize
                ↪ -1], IRT_EQ, note); // fix the last note as a note not in
                ↪ harmonic triad intervals
936             if (has_solution(problem)) {
937                 std::cerr   << "/!\\ ERROR test 1H3_4v_2sp: It exists solution
                    ↪ but it shouldn't with the following configuration:\n"
938                             << "cantus firmus: ";
939                 printVector(cantusFirmus);
940                 std::cerr   << "solution array: ";
941                 printIntVarArray(problem->getSolutionArray());
942                 std::cerr   << "> Last harmonic intervals must be in harmonic
                    ↪ triad" << std::endl;
943             }
944             delete problem;
945         }
946     }
947     cout << "End test 1H3_4v_2sp..." << endl;
948 }
949
950 void FuxTest::test_1H3_2v_3sp() {
951     cout << "Start test 1H3_2v_3sp..." << endl;
952     int cpSize = cfSize*4-3; // size of a counterpoint
953     int p_cons[] = {0, 7}; // perfect conssonant intervals
954     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect
        ↪ conssonant intervals
955     spList = {THIRD_SPECIES};
956     v_type = {0};
957     // non-perfect consonant intervals are forbidden
958     for (int interval : non_p_cons) {
959         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
            ↪ v_type, melodic_params, general_params, specific_params,
            ↪ importance, borrowMode);
960         int note = cantusFirmus[cfSize-1] + interval;
961         rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
            ↪ note); // fix the last note with a not perfect consonance
962         if (has_solution(problem)) {
963             std::cerr   << "/!\\ ERROR test 1H3_2v_3sp: It exists solution but it
                ↪ shouldn't with the following configuration:\n"
964                         << "cantus firmus: ";
965             printVector(cantusFirmus);
966             std::cerr   << "solution array: ";
967             printIntVarArray(problem->getSolutionArray());
968             std::cerr   << "> Last harmonic interval must be a perfect consonance
                ↪ " << std::endl;
969         }
970         delete problem;
971     }
```

```
    cout << "End test 1H3_2v_3sp..." << endl;
}

void FuxTest::test_1H3_3v_3sp() {
    cout << "Start test 1H3_3v_3sp..." << endl;
    int cpSize = cfSize*4-3; // size of a counterpoint
    int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
    int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad
      ↪ intervals
    spList = {THIRD_SPECIES, THIRD_SPECIES};
    v_type = {0, 0};
    // non-perfect consonant intervals are forbidden
    for (size_t i = 0; i < 2; i++) { // iterate over each solution line
        for (int interval : non_triad_cons) {
            CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪ v_type, melodic_params, general_params, specific_params,
              ↪ importance, borrowMode);
            int note = cantusFirmus[cfSize-1] + interval;
            rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize
              ↪ -1], IRT_EQ, note); // fix the last note as a note not in
              ↪ harmonic triad intervals
            if (has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H3_3v_3sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last harmonic intervals must be in harmonic
                  ↪ triad" << std::endl;
            }
            delete problem;
        }
    }
    cout << "End test 1H3_3v_3sp..." << endl;
}

void FuxTest::test_1H3_4v_3sp() {
    cout << "Start test 1H3_4v_3sp..." << endl;
    int cpSize = cfSize*4-3; // size of a counterpoint
    int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
    int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad
      ↪ intervals
    spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
    v_type = {0, 0, 0};
    // non-perfect consonant intervals are forbidden
    for (size_t i = 0; i < 3; i++) { // iterate over each solution line
        for (int interval : non_triad_cons) {
            CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪ v_type, melodic_params, general_params, specific_params,
              ↪ importance, borrowMode);
            int note = cantusFirmus[cfSize-1] + interval;
            rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize
              ↪ -1], IRT_EQ, note); // fix the last note as a note not in
              ↪ harmonic triad intervals
            if (has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H3_4v_3sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last harmonic intervals must be in harmonic
                  ↪ triad" << std::endl;
            }
            delete problem;
        }
    }
```

```
1026     cout << "End test 1H3_4v_3sp..." << endl;
1027 }
1028
1029 void FuxTest::test_1H3_2v_4sp() {
1030     cout << "Start test 1H3_2v_4sp..." << endl;
1031     int cpSize = cfSize*2-2; // size of a counterpoint
1032     int p_cons[] = {0, 7}; // perfect conssonant intervals
1033     int non_p_cons[] = {1, 2, 3, 4, 5, 6, 8, 9, 10, 11}; // non-perfect
          ↪ conssonant intervals
1034     spList = {FOURTH_SPECIES};
1035     v_type = {0};
1036     // non-perfect consonant intervals are forbidden
1037     for (int interval : non_p_cons) {
1038         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
              ↪ v_type, melodic_params, general_params, specific_params,
              ↪ importance, borrowMode);
1039         int note = cantusFirmus[cfSize-1] + interval;
1040         rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
              ↪ note); // fix the last note with a not perfect consonance
1041         if (has_solution(problem)) {
1042             std::cerr   << "/!\\ ERROR test 1H3_2v_4sp: It exists solution but it
                  ↪ shouldn't with the following configuration:\n"
1043                         << "cantus firmus: ";
1044             printVector(cantusFirmus);
1045             std::cerr   << "solution array: ";
1046             printIntVarArray(problem->getSolutionArray());
1047             std::cerr   << "> Last harmonic interval must be a perfect consonance
                  ↪ " << std::endl;
1048         }
1049         delete problem;
1050     }
1051     cout << "End test 1H3_2v_4sp..." << endl;
1052 }
1053
1054 void FuxTest::test_1H3_3v_4sp() {
1055     cout << "Start test 1H3_3v_4sp..." << endl;
1056     int cpSize = cfSize*2-2; // size of a counterpoint
1057     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
1058     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad
          ↪ intervals
1059     spList = {FOURTH_SPECIES, FOURTH_SPECIES};
1060     v_type = {0, 0};
1061     // non-perfect consonant intervals are forbidden
1062     for (size_t i = 0; i < 2; i++) { // iterate over each solution line
1063         for (int interval : non_triad_cons) {
1064             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪ v_type, melodic_params, general_params, specific_params,
                  ↪ importance, borrowMode);
1065             int note = cantusFirmus[cfSize-1] + interval;
1066             rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize
                  ↪ -1], IRT_EQ, note); // fix the last note as a note not in
                  ↪ harmonic triad intervals
1067             if (has_solution(problem)) {
1068                 std::cerr   << "/!\\ ERROR test 1H3_3v_4sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
1069                             << "cantus firmus: ";
1070                 printVector(cantusFirmus);
1071                 std::cerr   << "solution array: ";
1072                 printIntVarArray(problem->getSolutionArray());
1073                 std::cerr   << "> Last harmonic intervals must be in harmonic
                      ↪ triad" << std::endl;
1074             }
1075             delete problem;
1076         }
1077     }
1078     cout << "End test 1H3_3v_4sp..." << endl;
1079 }
1080
```

```
1081 void FuxTest::test_1H3_4v_4sp() {
1082     cout << "Start␣test␣1H3_4v_4sp..." << endl;
1083     int cpSize = cfSize*2-2; // size of a counterpoint
1084     int triad_cons[] = {0, 3, 4, 7}; // harmonic triad intervals
1085     int non_triad_cons[] = {1, 2, 5, 6, 8, 9, 10, 11}; // non-harmonic triad
          ↪ intervals
1086     spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
1087     v_type = {0, 0, 0};
1088     // non-perfect consonant intervals are forbidden
1089     for (size_t i = 0; i < 3; i++) { // iterate over each solution line
1090         for (int interval : non_triad_cons) {
1091             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪ v_type, melodic_params, general_params, specific_params,
                  ↪ importance, borrowMode);
1092             int note = cantusFirmus[cfSize-1] + interval;
1093             rel(problem->getHome(), problem->getSolutionArray()[(cpSize*i)+cpSize
                  ↪ -1], IRT_EQ, note); // fix the last note as a note not in
                  ↪ harmonic triad intervals
1094             if (has_solution(problem)) {
1095                 std::cerr   << "/!\\␣ERROR␣test␣1H3_4v_4sp:␣It␣exists␣solution␣
                      ↪ but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1096                             << "cantus␣firmus:␣";
1097                 printVector(cantusFirmus);
1098                 std::cerr   << "solution␣array:␣";
1099                 printIntVarArray(problem->getSolutionArray());
1100                 std::cerr   << ">␣Last␣harmonic␣intervals␣must␣be␣in␣harmonic␣
                      ↪ triad" << std::endl;
1101             }
1102             delete problem;
1103         }
1104     }
1105     cout << "End␣test␣1H3_4v_4sp..." << endl;
1106 }
1107
1108 /*The key tone is tuned according to the first note of the cantus firmus.
1109 In other words, the lowest stratum at the first and last notes must be the tonic.
      ↪ */
1110 void FuxTest::test_1H4() {
1111     cout << "Start␣tests␣1H4..." << endl;
1112     test_1H4_2v_1sp();
1113     test_1H4_3v_1sp();
1114     test_1H4_4v_1sp();
1115     test_1H4_2v_2sp();
1116     test_1H4_3v_2sp();
1117     test_1H4_4v_2sp();
1118     test_1H4_2v_3sp();
1119     test_1H4_3v_3sp();
1120     test_1H4_4v_3sp();
1121     test_1H4_2v_4sp();
1122     test_1H4_3v_4sp();
1123     test_1H4_4v_4sp();
1124     cout << "End␣tests␣1H4." << endl;
1125 }
1126
1127 void FuxTest::test_1H4_2v_1sp() {
1128     cout << "Start␣test␣1H4_2v_1sp..." << endl;
1129     spList = {FIRST_SPECIES};
1130     v_type = {-2};
1131     for (int i = -18; i < 1; i++) { //iterate over every note that can take
          ↪ solution array
1132         // fix the first note of the lowest stratum
1133         auto* problem = create_problem(cantusFirmus, spList, v_type,
              ↪ melodic_params, general_params, specific_params, importance,
              ↪ borrowMode);
1134         int note = cantusFirmus[0] + i;
1135         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
              ↪ fix the first note of the lowest stratum
1136         if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
```

```
1137            std::cerr    << "/!\\ ERROR test 1H4_2v_1sp: It doesn't exist a
                  ↪ solution but it should with the following configuration:\n"
1138                         << "Cantus firmus: ";
1139            printVector(cantusFirmus);
1140            std::cerr    << "Solution array: ";
1141            printIntVarArray(problem->getSolutionArray());
1142            std::cerr    << "> First note of lowest stratum is the tonic and must
                  ↪ be" << std::endl;
1143        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1144                std::cerr    << "/!\\ ERROR test 1H4_2v_1sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
1145                             << "cantus firmus: ";
1146                printVector(cantusFirmus);
1147                std::cerr    << "solution array: ";
1148                printIntVarArray(problem->getSolutionArray());
1149                std::cerr    << "> First note of lowest stratum is not the tonic"
                      ↪ << std::endl;
1150        }
1151        delete problem;
1152        // fix the last note of the lowest stratum
1153        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
                  ↪ general_params, specific_params, importance, borrowMode);
1154        note = cantusFirmus[cfSize-1] + i;
1155        rel(problem->getHome(), problem->getSolutionArray()[cfSize-1], IRT_EQ,
                  ↪ note); // fix the last note of the lowest stratum
1156        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1157            std::cerr    << "/!\\ ERROR test 1H4_2v_1sp: It doesn't exist a
                      ↪ solution but it should with the following configuration:\n"
1158                         << "Cantus firmus: ";
1159            printVector(cantusFirmus);
1160            std::cerr    << "Solution array: ";
1161            printIntVarArray(problem->getSolutionArray());
1162            std::cerr    << "> Last note of lowest stratum is the tonic and must
                      ↪ be" << std::endl;
1163        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
                  ↪ problem)) {
1164                std::cerr    << "/!\\ ERROR test 1H4_2v_1sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
1165                             << "cantus firmus: ";
1166                printVector(cantusFirmus);
1167                std::cerr    << "solution array: ";
1168                printIntVarArray(problem->getSolutionArray());
1169                std::cerr    << "> Last note of lowest stratum is not the tonic"
                      ↪ << std::endl;
1170        }
1171        delete problem;
1172    }
1173    cout << "End test 1H4_2v_1sp..." << endl;
1174 }
1175
1176 void FuxTest::test_1H4_3v_1sp() {
1177    cout << "Start test 1H4_3v_1sp..." << endl;
1178    // Lowest stratum is the first solution array
1179    spList = {FIRST_SPECIES, FIRST_SPECIES};
1180    v_type = {-2, 3};
1181    for (int i = -18; i < 1; i++) { //iterate over every note that can take
          ↪ solution array
1182        // fix the first note of the lowest stratum
1183        auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪ melodic_params, general_params, specific_params, importance,
                  ↪ borrowMode);
1184        int note = cantusFirmus[0] + i;
1185        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
                  ↪ fix the first note of the lowest stratum
1186        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1187            std::cerr    << "/!\\ ERROR test 1H4_3v_1sp: It doesn't exist a
                      ↪ solution but it should with the following configuration:\n"
1188                         << "Cantus firmus: ";
```

```
1189            printVector(cantusFirmus);
1190            std::cerr   << "Solution␣array:␣";
1191            printIntVarArray(problem->getSolutionArray());
1192            std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                  ↪ be" << std::endl;
1193        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1194                std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_1sp:␣It␣exists␣solution␣
                      ↪ but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1195                            << "cantus␣firmus:␣";
1196                printVector(cantusFirmus);
1197                std::cerr   << "solution␣array:␣";
1198                printIntVarArray(problem->getSolutionArray());
1199                std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                      ↪ << std::endl;
1200        }
1201        delete problem;
1202        // fix the last note of the lowest stratum
1203        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
                  ↪ general_params, specific_params, importance, borrowMode);
1204        note = cantusFirmus[cfSize-1] + i;
1205        rel(problem->getHome(), problem->getSolutionArray()[cfSize-1], IRT_EQ,
                  ↪ note); // fix the last note of the lowest stratum
1206        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1207            std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_1sp:␣It␣doesn't␣exist␣a␣
                      ↪ solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
1208                        << "Cantus␣firmus:␣";
1209            printVector(cantusFirmus);
1210            std::cerr   << "Solution␣array:␣";
1211            printIntVarArray(problem->getSolutionArray());
1212            std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                      ↪ be" << std::endl;
1213        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
                  ↪ problem)) {
1214                std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_1sp:␣It␣exists␣solution␣
                      ↪ but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1215                            << "cantus␣firmus:␣";
1216                printVector(cantusFirmus);
1217                std::cerr   << "solution␣array:␣";
1218                printIntVarArray(problem->getSolutionArray());
1219                std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                      ↪ << std::endl;
1220        }
1221        delete problem;
1222    }
1223    // Lowest stratum is the second solution array
1224    spList = {FIRST_SPECIES, FIRST_SPECIES};
1225    v_type = {3, -2};
1226    for (int i = -18; i < 1; i++) { //iterate over every note that can take
          ↪ solution array
1227        // fix the first note of the lowest stratum
1228        auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪ melodic_params, general_params, specific_params, importance,
                  ↪ borrowMode);
1229        int note = cantusFirmus[0] + i;
1230        rel(problem->getHome(), problem->getSolutionArray()[cfSize], IRT_EQ, note
                  ↪ ); // fix the first note of the lowest stratum
1231        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1232            std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_1sp:␣It␣doesn't␣exist␣a␣
                      ↪ solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
1233                        << "Cantus␣firmus:␣";
1234            printVector(cantusFirmus);
1235            std::cerr   << "Solution␣array:␣";
1236            printIntVarArray(problem->getSolutionArray());
1237            std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                      ↪ be" << std::endl;
1238        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1239                std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_1sp:␣It␣exists␣solution␣
                      ↪ but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
```

```
                            << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                  ↪  << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪  general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[(cfSize*2) -1],
          ↪  IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_1sp:␣It␣doesn't␣exist␣a␣
              ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
                        << "Cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr   << "Solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
              ↪  be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
          ↪  problem)) {
                std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_1sp:␣It␣exists␣solution␣
                  ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                            << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                  ↪  << std::endl;
        }
        delete problem;
    }
    cout << "End␣test␣1H4_3v_1sp." << endl;
}

void FuxTest::test_1H4_4v_1sp() {
    cout << "Start␣test␣1H4_4v_1sp..." << endl;
    // Lowest stratum is the first solution array
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    v_type = {-2, 3, 3};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take
      ↪  solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪  melodic_params, general_params, specific_params, importance,
          ↪  borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
          ↪   fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_1sp:␣It␣doesn't␣exist␣a␣
              ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
                        << "Cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr   << "Solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
              ↪  be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_1sp:␣It␣exists␣solution␣
                  ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                            << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
```

```
1293                printIntVarArray(problem->getSolutionArray());
1294                std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                      ↪  << std::endl;
1295            }
1296            delete problem;
1297            // fix the last note of the lowest stratum
1298            problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
                  ↪  general_params, specific_params, importance, borrowMode);
1299            note = cantusFirmus[cfSize-1] + i;
1300            rel(problem->getHome(), problem->getSolutionArray()[cfSize-1], IRT_EQ,
                  ↪  note); // fix the last note of the lowest stratum
1301            if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1302                std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_1sp:␣It␣doesn't␣exist␣a␣
                      ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
1303                            << "Cantus␣firmus:␣";
1304                printVector(cantusFirmus);
1305                std::cerr   << "Solution␣array:␣";
1306                printIntVarArray(problem->getSolutionArray());
1307                std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                      ↪  be" << std::endl;
1308            } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
                  ↪  problem)) {
1309                  std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_1sp:␣It␣exists␣solution␣
                        ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1310                              << "cantus␣firmus:␣";
1311                  printVector(cantusFirmus);
1312                  std::cerr   << "solution␣array:␣";
1313                  printIntVarArray(problem->getSolutionArray());
1314                  std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                        ↪  << std::endl;
1315            }
1316            delete problem;
1317        }
1318        // Lowest stratum is the second solution array
1319        spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
1320        v_type = {3, -2, 3};
1321        for (int i = -18; i < 1; i++) { //iterate over every note that can take
              ↪  solution array
1322            // fix the first note of the lowest stratum
1323            auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪  melodic_params, general_params, specific_params, importance,
                  ↪  borrowMode);
1324            int note = cantusFirmus[0] + i;
1325            rel(problem->getHome(), problem->getSolutionArray()[cfSize], IRT_EQ, note
                  ↪  ); // fix the first note of the lowest stratum
1326            if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1327                std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_1sp:␣It␣doesn't␣exist␣a␣
                      ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
1328                            << "Cantus␣firmus:␣";
1329                printVector(cantusFirmus);
1330                std::cerr   << "Solution␣array:␣";
1331                printIntVarArray(problem->getSolutionArray());
1332                std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                      ↪  be" << std::endl;
1333            } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1334                  std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_1sp:␣It␣exists␣solution␣
                        ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1335                              << "cantus␣firmus:␣";
1336                  printVector(cantusFirmus);
1337                  std::cerr   << "solution␣array:␣";
1338                  printIntVarArray(problem->getSolutionArray());
1339                  std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                        ↪  << std::endl;
1340            }
1341            delete problem;
1342            // fix the last note of the lowest stratum
1343            problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
                  ↪  general_params, specific_params, importance, borrowMode);
```

```
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[(cfSize*2) -1],
          ↪ IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
          ↪ problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It exists solution
                ↪ but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
              std::cerr   << "> Last note of lowest stratum is not the tonic"
                ↪ << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the third solution array
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    v_type = {3, 3, -2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take
      ↪ solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪ melodic_params, general_params, specific_params, importance,
          ↪ borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cfSize*2], IRT_EQ,
          ↪ note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It exists solution
                ↪ but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
              std::cerr   << "> First note of lowest stratum is not the tonic"
                ↪ << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪ general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[(cfSize*3) -1],
          ↪ IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_1sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
```

```
1395            std::cerr   << "Solution␣array:␣";
1396            printIntVarArray(problem->getSolutionArray());
1397            std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                  ↪ be" << std::endl;
1398        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
              ↪ problem)) {
1399              std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_1sp:␣It␣exists␣solution␣
                    ↪ but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1400                          << "cantus␣firmus:␣";
1401              printVector(cantusFirmus);
1402              std::cerr   << "solution␣array:␣";
1403              printIntVarArray(problem->getSolutionArray());
1404              std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                    ↪ << std::endl;
1405        }
1406        delete problem;
1407    }
1408    cout << "End␣test␣1H4_4v_1sp." << endl;
1409 }
1410
1411 void FuxTest::test_1H4_2v_2sp() {
1412    cout << "Start␣test␣1H4_2v_2sp..." << endl;
1413    int cpSize = cfSize*2-1; // size of a counterpoint
1414    spList = {SECOND_SPECIES};
1415    v_type = {-2};
1416    for (int i = -18; i < 1; i++) { //iterate over every note that can take
          ↪ solution array
1417        // fix the first note of the lowest stratum
1418        auto* problem = create_problem(cantusFirmus, spList, v_type,
              ↪ melodic_params, general_params, specific_params, importance,
              ↪ borrowMode);
1419        int note = cantusFirmus[0] + i;
1420        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
              ↪  fix the first note of the lowest stratum
1421        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1422            std::cerr   << "/!\\␣ERROR␣test␣1H4_2v_2sp:␣It␣doesn't␣exist␣a␣
                  ↪ solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
1423                        << "Cantus␣firmus:␣";
1424            printVector(cantusFirmus);
1425            std::cerr   << "Solution␣array:␣";
1426            printIntVarArray(problem->getSolutionArray());
1427            std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                  ↪ be" << std::endl;
1428        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1429              std::cerr   << "/!\\␣ERROR␣test␣1H4_2v_2sp:␣It␣exists␣solution␣
                    ↪ but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1430                          << "cantus␣firmus:␣";
1431              printVector(cantusFirmus);
1432              std::cerr   << "solution␣array:␣";
1433              printIntVarArray(problem->getSolutionArray());
1434              std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                    ↪ << std::endl;
1435        }
1436        delete problem;
1437        // fix the last note of the lowest stratum
1438        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
              ↪ general_params, specific_params, importance, borrowMode);
1439        note = cantusFirmus[cfSize-1] + i;
1440        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
              ↪ note); // fix the last note of the lowest stratum
1441        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1442            std::cerr   << "/!\\␣ERROR␣test␣1H4_2v_2sp:␣It␣doesn't␣exist␣a␣
                  ↪ solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
1443                        << "Cantus␣firmus:␣";
1444            printVector(cantusFirmus);
1445            std::cerr   << "Solution␣array:␣";
1446            printIntVarArray(problem->getSolutionArray());
```

```
1447            std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                  ↪  be" << std::endl;
1448        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
                  ↪  problem)) {
1449              std::cerr   << "/!\\␣ERROR␣test␣1H4_2v_2sp:␣It␣exists␣solution␣
                    ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1450                          << "cantus␣firmus:␣";
1451              printVector(cantusFirmus);
1452              std::cerr   << "solution␣array:␣";
1453              printIntVarArray(problem->getSolutionArray());
1454              std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                    ↪  << std::endl;
1455        }
1456        delete problem;
1457    }
1458    cout << "End␣test␣1H4_2v_2sp..." << endl;
1459 }
1460
1461 void FuxTest::test_1H4_3v_2sp() {
1462    cout << "Start␣test␣1H4_3v_2sp..." << endl;
1463    int cpSize = cfSize*2-1; // size of a counterpoint
1464    // Lowest stratum is the first solution array
1465    spList = {SECOND_SPECIES, SECOND_SPECIES};
1466    v_type = {-2, 3};
1467    for (int i = -18; i < 1; i++) { //iterate over every note that can take
                  ↪  solution array
1468        // fix the first note of the lowest stratum
1469        auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪  melodic_params, general_params, specific_params, importance,
                  ↪  borrowMode);
1470        int note = cantusFirmus[0] + i;
1471        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
                  ↪   fix the first note of the lowest stratum
1472        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1473            std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_2sp:␣It␣doesn't␣exist␣a␣
                    ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
1474                        << "Cantus␣firmus:␣";
1475            printVector(cantusFirmus);
1476            std::cerr   << "Solution␣array:␣";
1477            printIntVarArray(problem->getSolutionArray());
1478            std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                    ↪  be" << std::endl;
1479        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1480              std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_2sp:␣It␣exists␣solution␣
                    ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
1481                          << "cantus␣firmus:␣";
1482              printVector(cantusFirmus);
1483              std::cerr   << "solution␣array:␣";
1484              printIntVarArray(problem->getSolutionArray());
1485              std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                    ↪  << std::endl;
1486        }
1487        delete problem;
1488        // fix the last note of the lowest stratum
1489        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
                  ↪  general_params, specific_params, importance, borrowMode);
1490        note = cantusFirmus[cfSize-1] + i;
1491        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
                  ↪  note); // fix the last note of the lowest stratum
1492        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1493            std::cerr   << "/!\\␣ERROR␣test␣1H4_3v_2sp:␣It␣doesn't␣exist␣a␣
                    ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
1494                        << "Cantus␣firmus:␣";
1495            printVector(cantusFirmus);
1496            std::cerr   << "Solution␣array:␣";
1497            printIntVarArray(problem->getSolutionArray());
1498            std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
                    ↪  be" << std::endl;
```

```
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It exists solution but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
              std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the second solution array
    spList = {SECOND_SPECIES, SECOND_SPECIES};
    v_type = {3, -2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It exists solution but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
              std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2-1], IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_3v_2sp: It exists solution but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
```

```
                std::cerr    << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                  ↪  << std::endl;
        }
        delete problem;
    }
    cout << "End␣test␣1H4_3v_2sp." << endl;
}

void FuxTest::test_1H4_4v_2sp() {
    cout << "Start␣test␣1H4_4v_2sp..." << endl;
    int cpSize = cfSize*2-1; // size of a counterpoint
    // Lowest stratum is the first solution array
    spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
    v_type = {-2, 3, 3};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take
      ↪  solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪  melodic_params, general_params, specific_params, importance,
          ↪  borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
          ↪   fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr    << "/!\\␣ERROR␣test␣1H4_4v_2sp:␣It␣doesn't␣exist␣a␣
              ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
                         << "Cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr    << "Solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr    << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
              ↪  be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr    << "/!\\␣ERROR␣test␣1H4_4v_2sp:␣It␣exists␣solution␣
                  ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                             << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr    << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr    << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                  ↪  << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪  general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
          ↪  note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr    << "/!\\␣ERROR␣test␣1H4_4v_2sp:␣It␣doesn't␣exist␣a␣
              ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
                         << "Cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr    << "Solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr    << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
              ↪  be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
          ↪  problem)) {
                std::cerr    << "/!\\␣ERROR␣test␣1H4_4v_2sp:␣It␣exists␣solution␣
                  ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                             << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr    << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr    << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                  ↪  << std::endl;
```

```
        }
        delete problem;
    }
    // Lowest stratum is the second solution array
    spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
    v_type = {3, -2, 3};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2 -1], IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the third solution array
    spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
    v_type = {3, 3, -2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
```

```
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪ melodic_params, general_params, specific_params, importance,
          ↪ borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2], IRT_EQ,
          ↪ note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> First note of lowest stratum is not the tonic"
                  ↪ << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪ general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[(cpSize*3) -1],
          ↪ IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
          ↪ problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_4v_2sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic"
                  ↪ << std::endl;
        }
        delete problem;
    }
    cout << "End test 1H4_4v_2sp." << endl;
}

void FuxTest::test_1H4_2v_3sp() {
    cout << "Start test 1H4_2v_3sp..." << endl;
    int cpSize = cfSize*4-3; // size of a counterpoint
    spList = {THIRD_SPECIES};
    v_type = {-2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take
      ↪ solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪ melodic_params, general_params, specific_params, importance,
```

```
            ↪ borrowMode);
1706        int note = cantusFirmus[0] + i;
1707        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
              ↪ fix the first note of the lowest stratum
1708        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
1709            std::cerr << "/!\\ ERROR test 1H4_2v_3sp: It doesn't exist a
                ↪ solution but it should with the following configuration:\n"
1710                      << "Cantus firmus: ";
1711            printVector(cantusFirmus);
1712            std::cerr << "Solution array: ";
1713            printIntVarArray(problem->getSolutionArray());
1714            std::cerr << "> First note of lowest stratum is the tonic and must
                ↪ be" << std::endl;
1715        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
1716              std::cerr << "/!\\ ERROR test 1H4_2v_3sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
1717                        << "cantus firmus: ";
1718              printVector(cantusFirmus);
1719              std::cerr << "solution array: ";
1720              printIntVarArray(problem->getSolutionArray());
1721              std::cerr << "> First note of lowest stratum is not the tonic"
                  ↪ << std::endl;
1722        }
1723        delete problem;
1724        // fix the last note of the lowest stratum
1725        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
              ↪ general_params, specific_params, importance, borrowMode);
1726        note = cantusFirmus[cfSize-1] + i;
1727        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
              ↪ note); // fix the last note of the lowest stratum
1728        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
1729            std::cerr << "/!\\ ERROR test 1H4_2v_3sp: It doesn't exist a
                ↪ solution but it should with the following configuration:\n"
1730                      << "Cantus firmus: ";
1731            printVector(cantusFirmus);
1732            std::cerr << "Solution array: ";
1733            printIntVarArray(problem->getSolutionArray());
1734            std::cerr << "> Last note of lowest stratum is the tonic and must
                ↪ be" << std::endl;
1735        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
              ↪ problem)) {
1736              std::cerr << "/!\\ ERROR test 1H4_2v_3sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
1737                        << "cantus firmus: ";
1738              printVector(cantusFirmus);
1739              std::cerr << "solution array: ";
1740              printIntVarArray(problem->getSolutionArray());
1741              std::cerr << "> Last note of lowest stratum is not the tonic"
                  ↪ << std::endl;
1742        }
1743        delete problem;
1744    }
1745    cout << "End test 1H4_2v_3sp..." << endl;
1746 }
1747
1748 void FuxTest::test_1H4_3v_3sp() {
1749    cout << "Start test 1H4_3v_3sp..." << endl;
1750    int cpSize = cfSize*4-3; // size of a counterpoint
1751    // Lowest stratum is the first solution array
1752    spList = {THIRD_SPECIES, THIRD_SPECIES};
1753    v_type = {-2, 3};
1754    for (int i = -18; i < 1; i++) { //iterate over every note that can take
          ↪ solution array
1755        // fix the first note of the lowest stratum
1756        auto* problem = create_problem(cantusFirmus, spList, v_type,
              ↪ melodic_params, general_params, specific_params, importance,
              ↪ borrowMode);
1757        int note = cantusFirmus[0] + i;
```

```
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the second solution array
    spList = {THIRD_SPECIES, THIRD_SPECIES};
    v_type = {3, -2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
```

```
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2-1], IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_3v_3sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
    }
    cout << "End test 1H4_3v_3sp." << endl;
}

void FuxTest::test_1H4_4v_3sp() {
    cout << "Start test 1H4_4v_3sp..." << endl;
    int cpSize = cfSize*4-3; // size of a counterpoint
    // Lowest stratum is the first solution array
    spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
    v_type = {-2, 3, 3};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
```

```
            std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_3sp:␣It␣exists␣solution␣but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                            << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic" << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_2sp:␣It␣doesn't␣exist␣a␣solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
                        << "Cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr   << "Solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_3sp:␣It␣exists␣solution␣but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                            << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic" << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the second solution array
    spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
    v_type = {3, -2, 3};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note ); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_3sp:␣It␣doesn't␣exist␣a␣solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
                        << "Cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr   << "Solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_3sp:␣It␣exists␣solution␣but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                            << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
```

```
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                  ↪  << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪  general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2 -1], IRT_EQ,
          ↪   note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_3sp:␣It␣doesn't␣exist␣a␣
              ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
                        << "Cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr   << "Solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
              ↪  be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
          ↪  problem)) {
                std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_3sp:␣It␣exists␣solution␣
                  ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                            << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << ">␣Last␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                  ↪  << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the third solution array
    spList = {THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES};
    v_type = {3, 3, -2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take
      ↪  solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪  melodic_params, general_params, specific_params, importance,
          ↪  borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2], IRT_EQ,
          ↪  note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_3sp:␣It␣doesn't␣exist␣a␣
              ↪  solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
                        << "Cantus␣firmus:␣";
            printVector(cantusFirmus);
            std::cerr   << "Solution␣array:␣";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣the␣tonic␣and␣must␣
              ↪  be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\␣ERROR␣test␣1H4_4v_3sp:␣It␣exists␣solution␣
                  ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
                            << "cantus␣firmus:␣";
                printVector(cantusFirmus);
                std::cerr   << "solution␣array:␣";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << ">␣First␣note␣of␣lowest␣stratum␣is␣not␣the␣tonic"
                  ↪  << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪  general_params, specific_params, importance, borrowMode);
```

```
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[(cpSize*3) -1],
          ↪ IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
          ↪ problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_4v_3sp: It exists solution
                ↪ but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
              std::cerr   << "> Last note of lowest stratum is not the tonic"
                ↪ << std::endl;
        }
        delete problem;
    }
    cout << "End test 1H4_4v_3sp." << endl;
}

void FuxTest::test_1H4_2v_4sp() {
    cout << "Start test 1H4_2v_4sp..." << endl;
    int cpSize = cfSize*2-2; // size of a counterpoint
    spList = {FOURTH_SPECIES};
    v_type = {-2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take
      ↪ solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪ melodic_params, general_params, specific_params, importance,
          ↪ borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
          ↪  fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_2v_4sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_2v_4sp: It exists solution
                ↪ but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
              std::cerr   << "> First note of lowest stratum is not the tonic"
                ↪ << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪ general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
          ↪ note); // fix the last note of the lowest stratum
```

```
2015         if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
2016             std::cerr   << "/!\\ ERROR test 1H4_2v_4sp: It doesn't exist a
                 ↪ solution but it should with the following configuration:\n"
2017                         << "Cantus firmus: ";
2018             printVector(cantusFirmus);
2019             std::cerr   << "Solution array: ";
2020             printIntVarArray(problem->getSolutionArray());
2021             std::cerr   << "> Last note of lowest stratum is the tonic and must
                 ↪ be" << std::endl;
2022         } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
                 ↪ problem)) {
2023               std::cerr   << "/!\\ ERROR test 1H4_2v_4sp: It exists solution
                   ↪ but it shouldn't with the following configuration:\n"
2024                           << "cantus firmus: ";
2025               printVector(cantusFirmus);
2026               std::cerr   << "solution array: ";
2027               printIntVarArray(problem->getSolutionArray());
2028               std::cerr   << "> Last note of lowest stratum is not the tonic"
                     ↪ << std::endl;
2029         }
2030         delete problem;
2031     }
2032     cout << "End test 1H4_2v_4sp..." << endl;
2033 }
2034
2035 void FuxTest::test_1H4_3v_4sp() {
2036     cout << "Start test 1H4_3v_4sp..." << endl;
2037     int cpSize = cfSize*2-2; // size of a counterpoint
2038     // Lowest stratum is the first solution array
2039     spList = {FOURTH_SPECIES, FOURTH_SPECIES};
2040     v_type = {-2, 3};
2041     for (int i = -18; i < 1; i++) { //iterate over every note that can take
          ↪ solution array
2042         // fix the first note of the lowest stratum
2043         auto* problem = create_problem(cantusFirmus, spList, v_type,
                 ↪ melodic_params, general_params, specific_params, importance,
                 ↪ borrowMode);
2044         int note = cantusFirmus[0] + i;
2045         rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
                 ↪  fix the first note of the lowest stratum
2046         if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
2047             std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It doesn't exist a
                   ↪ solution but it should with the following configuration:\n"
2048                         << "Cantus firmus: ";
2049             printVector(cantusFirmus);
2050             std::cerr   << "Solution array: ";
2051             printIntVarArray(problem->getSolutionArray());
2052             std::cerr   << "> First note of lowest stratum is the tonic and must
                   ↪ be" << std::endl;
2053         } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
2054               std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It exists solution
                   ↪ but it shouldn't with the following configuration:\n"
2055                           << "cantus firmus: ";
2056               printVector(cantusFirmus);
2057               std::cerr   << "solution array: ";
2058               printIntVarArray(problem->getSolutionArray());
2059               std::cerr   << "> First note of lowest stratum is not the tonic"
                     ↪ << std::endl;
2060         }
2061         delete problem;
2062         // fix the last note of the lowest stratum
2063         problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
                 ↪ general_params, specific_params, importance, borrowMode);
2064         note = cantusFirmus[cfSize-1] + i;
2065         rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
                 ↪ note); // fix the last note of the lowest stratum
2066         if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
```

```
            std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
          ↪ problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It exists solution
                ↪ but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
              std::cerr   << "> Last note of lowest stratum is not the tonic"
                ↪ << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the second solution array
    spList = {FOURTH_SPECIES, FOURTH_SPECIES};
    v_type = {3, -2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take
      ↪ solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪ melodic_params, general_params, specific_params, importance,
          ↪ borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note
          ↪ ); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
              std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It exists solution
                ↪ but it shouldn't with the following configuration:\n"
                          << "cantus firmus: ";
              printVector(cantusFirmus);
              std::cerr   << "solution array: ";
              printIntVarArray(problem->getSolutionArray());
              std::cerr   << "> First note of lowest stratum is not the tonic"
                ↪ << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪ general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2-1], IRT_EQ,
          ↪ note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
```

```
2118        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
              ↪ problem)) {
2119            std::cerr   << "/!\\ ERROR test 1H4_3v_4sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
2120                        << "cantus firmus: ";
2121            printVector(cantusFirmus);
2122            std::cerr   << "solution array: ";
2123            printIntVarArray(problem->getSolutionArray());
2124            std::cerr   << "> Last note of lowest stratum is not the tonic"
                  ↪ << std::endl;
2125        }
2126        delete problem;
2127    }
2128    cout << "End test 1H4_3v_4sp." << endl;
2129 }
2130
2131 void FuxTest::test_1H4_4v_4sp() {
2132    cout << "Start test 1H4_4v_4sp..." << endl;
2133    int cpSize = cfSize*2-2; // size of a counterpoint
2134    // Lowest stratum is the first solution array
2135    spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
2136    v_type = {-2, 3, 3};
2137    for (int i = -18; i < 1; i++) { //iterate over every note that can take
         ↪ solution array
2138        // fix the first note of the lowest stratum
2139        auto* problem = create_problem(cantusFirmus, spList, v_type,
              ↪ melodic_params, general_params, specific_params, importance,
              ↪ borrowMode);
2140        int note = cantusFirmus[0] + i;
2141        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note); //
              ↪  fix the first note of the lowest stratum
2142        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
2143            std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a
                  ↪ solution but it should with the following configuration:\n"
2144                        << "Cantus firmus: ";
2145            printVector(cantusFirmus);
2146            std::cerr   << "Solution array: ";
2147            printIntVarArray(problem->getSolutionArray());
2148            std::cerr   << "> First note of lowest stratum is the tonic and must
                  ↪ be" << std::endl;
2149        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
2150            std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
2151                        << "cantus firmus: ";
2152            printVector(cantusFirmus);
2153            std::cerr   << "solution array: ";
2154            printIntVarArray(problem->getSolutionArray());
2155            std::cerr   << "> First note of lowest stratum is not the tonic"
                  ↪ << std::endl;
2156        }
2157        delete problem;
2158        // fix the last note of the lowest stratum
2159        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
              ↪ general_params, specific_params, importance, borrowMode);
2160        note = cantusFirmus[cfSize-1] + i;
2161        rel(problem->getHome(), problem->getSolutionArray()[cpSize-1], IRT_EQ,
              ↪ note); // fix the last note of the lowest stratum
2162        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
2163            std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a
                  ↪ solution but it should with the following configuration:\n"
2164                        << "Cantus firmus: ";
2165            printVector(cantusFirmus);
2166            std::cerr   << "Solution array: ";
2167            printIntVarArray(problem->getSolutionArray());
2168            std::cerr   << "> Last note of lowest stratum is the tonic and must
                  ↪ be" << std::endl;
2169        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
              ↪ problem)) {
```

```
                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic"
                  ↪ << std::endl;
        }
        delete problem;
    }
    // Lowest stratum is the second solution array
    spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
    v_type = {3, -2, 3};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take
      ↪ solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪ melodic_params, general_params, specific_params, importance,
          ↪ borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize], IRT_EQ, note
          ↪ ); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> First note of lowest stratum is not the tonic"
                  ↪ << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
          ↪ general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2 -1], IRT_EQ,
          ↪  note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a
              ↪ solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must
              ↪ be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(
          ↪ problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic"
                  ↪ << std::endl;
```

```
        }
        delete problem;
    }
    // Lowest stratum is the third solution array
    spList = {FOURTH_SPECIES, FOURTH_SPECIES, FOURTH_SPECIES};
    v_type = {3, 3, -2};
    for (int i = -18; i < 1; i++) { //iterate over every note that can take solution array
        // fix the first note of the lowest stratum
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        int note = cantusFirmus[0] + i;
        rel(problem->getHome(), problem->getSolutionArray()[cpSize*2], IRT_EQ, note); // fix the first note of the lowest stratum
        if (note % 12 == cantusFirmus[0] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> First note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[0] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> First note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
        // fix the last note of the lowest stratum
        problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        note = cantusFirmus[cfSize-1] + i;
        rel(problem->getHome(), problem->getSolutionArray()[(cpSize*3) -1], IRT_EQ, note); // fix the last note of the lowest stratum
        if (note % 12 == cantusFirmus[cfSize-1] % 12 && !has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It doesn't exist a solution but it should with the following configuration:\n"
                        << "Cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "Solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Last note of lowest stratum is the tonic and must be" << std::endl;
        } else if (note % 12 != cantusFirmus[cfSize-1] % 12 && has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H4_4v_4sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Last note of lowest stratum is not the tonic" << std::endl;
        }
        delete problem;
    }
    cout << "End test 1H4_4v_4sp." << endl;
}

/*
```

```
The voices cannot play the same note at the same time except in the first and last measure.
*/
void FuxTest::test_1H5() {
    cout << "Start tests 1H5..." << endl;
    test_1H5_2v_1sp();
    test_1H5_3v_1sp();
    test_1H5_2v_2sp();
    test_1H5_3v_2sp();
    test_1H5_2v_3sp();
    test_1H5_3v_3sp();
    test_1H5_2v_4sp();
    test_1H5_3v_4sp();
    cout << "End tests 1H5." << endl;
}

void FuxTest::test_1H5_2v_1sp() {
    cout << "Start test_1H5_2v_1sp ..." << endl;
    spList = {FIRST_SPECIES};
    v_type = {0};
    for (size_t i = 1; i < cfSize-1; i++) {
        auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
        rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, cantusFirmus[i]); // fix the solution note as the same note as the cf
        if (has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test_1H5_2v_1sp: It exists solution but it shouldn't with the following configuration:\n"
                        << "cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> Same note played at the mesure: " << i << std::endl;
        }
        delete problem;
    }
    cout << "End test_1H5_2v_1sp" << endl;
}

void FuxTest::test_1H5_3v_1sp() {
    cout << "Start test_1H5_3v_1sp..." << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    v_type = {0, 0};
    // // test solution lines != cf
    for (size_t j = 0; j < 2; j++) { // iterate over each solution line
        for (size_t i = 1; i < cfSize-1; i++) {
            auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
            rel(problem->getHome(), problem->getSolutionArray()[(j*cfSize) + i], IRT_EQ, cantusFirmus[i]); // fix the solution note as the same note as the cf
            if (has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test_1H5_3v_1sp: It exists solution but it shouldn't with the following configuration:\n"
                            << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> Same note played at the mesure: " << i << std::endl;
            }
            delete problem;
        }
    }
```

```
2328     // test notes of solutions lines are different
2329     for (size_t i = 1; i < cfSize-1; i++) {
2330         for (int j = -6; j < 13; j++) {
2331             auto* problem = create_problem(cantusFirmus, spList, v_type,
                     ↪ melodic_params, general_params, specific_params, importance,
                     ↪ borrowMode);
2332             int note = cantusFirmus[0] + j;
2333             rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, note)
                     ↪ ; // fix the note of the first solution line as the same note
                     ↪ as the note of the second solution line
2334             rel(problem->getHome(), problem->getSolutionArray()[cfSize+i], IRT_EQ
                     ↪ , note); // fix the note of the first solution line as the
                     ↪ same note as the note of the second solution line
2335             if (has_solution(problem)) {
2336                 std::cerr   << "/!\\ ERROR test 1H5_3v_1sp: It exists solution
                         ↪ but it shouldn't with the following configuration:\n"
2337                         << "cantus firmus: ";
2338                 printVector(cantusFirmus);
2339                 std::cerr   << "solution array: ";
2340                 printIntVarArray(problem->getSolutionArray());
2341                 std::cerr   << "> Same note played at the mesure: " << i << std::
                         ↪ endl;
2342             }
2343             delete problem;
2344         }
2345     }
2346     cout << "End test 1H5_3v_1sp." << endl;
2347 }
2348
2349 void FuxTest::test_1H5_2v_2sp() {
2350     cout << "Start test 1H5_2v_2sp ..." << endl;
2351     spList = {SECOND_SPECIES};
2352     v_type = {0};
2353     for (size_t i = 1; i < cfSize-1; i++) {
2354         auto* problem = create_problem(cantusFirmus, spList, v_type,
                 ↪ melodic_params, general_params, specific_params, importance,
                 ↪ borrowMode);
2355         rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ,
                 ↪ cantusFirmus[i]); // fix the solution note as the same note as the
                 ↪  cf
2356         if (has_solution(problem)) {
2357             std::cerr   << "/!\\ ERROR test 1H5_2v_2sp: It exists solution but it
                     ↪ shouldn't with the following configuration:\n"
2358                     << "cantus firmus: ";
2359             printVector(cantusFirmus);
2360             std::cerr   << "solution array: ";
2361             printIntVarArray(problem->getSolutionArray());
2362             std::cerr   << "> Same note played at the mesure: " << i << std::endl
                     ↪ ;
2363         }
2364         delete problem;
2365     }
2366     cout << "End test 1H5_2v_2sp" << endl;
2367 }
2368
2369 void FuxTest::test_1H5_3v_2sp() {
2370     cout << "Start test 1H5_3v_2sp..." << endl;
2371     int cpSize = cfSize*2-1; // size of a counterpoint
2372     spList = {SECOND_SPECIES, SECOND_SPECIES};
2373     v_type = {0, 0};
2374     // // test solution lines != cf
2375     for (size_t j = 0; j < 2; j++) { // iterate over each solution line
2376         for (size_t i = 1; i < cfSize-1; i++) { // iterate over each note of the
                 ↪ cantus firmus
2377             auto* problem = create_problem(cantusFirmus, spList, v_type,
                     ↪ melodic_params, general_params, specific_params, importance,
                     ↪ borrowMode);
```

```
2378            rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) + i
                  ↪  *2], IRT_EQ, cantusFirmus[i]); // fix the solution note as the
                  ↪   same note as the cf
2379            if (has_solution(problem)) {
2380                std::cerr   << "/!\\ ERROR test 1H5_3v_2sp: It exists solution
                      ↪  but it shouldn't with the following configuration:\n"
2381                            << "cantus firmus: ";
2382                printVector(cantusFirmus);
2383                std::cerr   << "solution array: ";
2384                printIntVarArray(problem->getSolutionArray());
2385                std::cerr   << "> Same note played at the mesure: " << i << std::
                      ↪  endl;
2386            }
2387            delete problem;
2388        }
2389    }
2390    // test notes of solutions lines are different
2391    for (size_t i = 1; i < cfSize-1; i++) {
2392        for (int j = -6; j < 13; j++) { // iterate over each possible note of
              ↪  solution lines
2393            auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪  melodic_params, general_params, specific_params, importance,
                  ↪  borrowMode);
2394            int note = cantusFirmus[0] + j;
2395            rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ,
                  ↪  note); // fix the note of the first solution line as the same
                  ↪  note as the note of the second solution line
2396            rel(problem->getHome(), problem->getSolutionArray()[cpSize+i*2],
                  ↪  IRT_EQ, note); // fix the note of the first solution line as
                  ↪  the same note as the note of the second solution line
2397            if (has_solution(problem)) {
2398                std::cerr   << "/!\\ ERROR test 1H5_3v_2sp: It exists solution
                      ↪  but it shouldn't with the following configuration:\n"
2399                            << "cantus firmus: ";
2400                printVector(cantusFirmus);
2401                std::cerr   << "solution array: ";
2402                printIntVarArray(problem->getSolutionArray());
2403                std::cerr   << "> Same note played at the mesure: " << i << std::
                      ↪  endl;
2404            }
2405            delete problem;
2406        }
2407    }
2408    cout << "End test 1H5_3v_2sp." << endl;
2409 }
2410
2411 void FuxTest::test_1H5_2v_3sp() {
2412    cout << "Start test 1H5_2v_3sp ..." << endl;
2413    spList = {THIRD_SPECIES};
2414    v_type = {0};
2415    for (size_t i = 1; i < cfSize-1; i++) {
2416        auto* problem = create_problem(cantusFirmus, spList, v_type,
              ↪  melodic_params, general_params, specific_params, importance,
              ↪  borrowMode);
2417        rel(problem->getHome(), problem->getSolutionArray()[i*4], IRT_EQ,
              ↪  cantusFirmus[i]); // fix the solution note as the same note as the
              ↪   cf
2418        if (has_solution(problem)) {
2419            std::cerr   << "/!\\ ERROR test 1H5_2v_3sp: It exists solution but it
                  ↪  shouldn't with the following configuration:\n"
2420                        << "cantus firmus: ";
2421            printVector(cantusFirmus);
2422            std::cerr   << "solution array: ";
2423            printIntVarArray(problem->getSolutionArray());
2424            std::cerr   << "> Same note played at the mesure: " << i << std::endl
                  ↪  ;
2425        }
2426        delete problem;
```

```
2427     }
2428     cout << "End test 1H5_2v_3sp" << endl;
2429 }
2430 
2431 void FuxTest::test_1H5_3v_3sp() {
2432     cout << "Start test 1H5_3v_3sp..." << endl;
2433     int cpSize = cfSize*4-3; // size of a counterpoint
2434     spList = {THIRD_SPECIES, THIRD_SPECIES};
2435     v_type = {0, 0};
2436     // // test solution lines != cf
2437     for (size_t j = 0; j < 2; j++) { // iterate over each solution line
2438         for (size_t i = 1; i < cfSize-1; i++) { // iterate over each note of the
            ↪  cantus firmus
2439             auto* problem = create_problem(cantusFirmus, spList, v_type,
                ↪  melodic_params, general_params, specific_params, importance,
                ↪  borrowMode);
2440             rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) + i
                ↪  *4], IRT_EQ, cantusFirmus[i]); // fix the solution note as the
                ↪   same note as the cf
2441             if (has_solution(problem)) {
2442                 std::cerr   << "/!\\ ERROR test 1H5_3v_3sp: It exists solution
                    ↪  but it shouldn't with the following configuration:\n"
2443                             << "cantus firmus: ";
2444                 printVector(cantusFirmus);
2445                 std::cerr   << "solution array: ";
2446                 printIntVarArray(problem->getSolutionArray());
2447                 std::cerr   << "> Same note played at the mesure: " << i << std::
                    ↪  endl;
2448             }
2449             delete problem;
2450         }
2451     }
2452     // test notes of solutions lines are different
2453     for (size_t i = 1; i < cfSize-1; i++) {
2454         for (int j = -6; j < 13; j++) { // iterate over each possible note of
            ↪  solution lines
2455             auto* problem = create_problem(cantusFirmus, spList, v_type,
                ↪  melodic_params, general_params, specific_params, importance,
                ↪  borrowMode);
2456             int note = cantusFirmus[0] + j;
2457             rel(problem->getHome(), problem->getSolutionArray()[i*4], IRT_EQ,
                ↪  note); // fix the note of the first solution line as the same
                ↪  note as the note of the second solution line
2458             rel(problem->getHome(), problem->getSolutionArray()[cpSize+i*4],
                ↪  IRT_EQ, note); // fix the note of the first solution line as
                ↪  the same note as the note of the second solution line
2459             if (has_solution(problem)) {
2460                 std::cerr   << "/!\\ ERROR test 1H5_3v_3sp: It exists solution
                    ↪  but it shouldn't with the following configuration:\n"
2461                             << "cantus firmus: ";
2462                 printVector(cantusFirmus);
2463                 std::cerr   << "solution array: ";
2464                 printIntVarArray(problem->getSolutionArray());
2465                 std::cerr   << "> Same note played at the mesure: " << i << std::
                    ↪  endl;
2466             }
2467             delete problem;
2468         }
2469     }
2470     cout << "End test 1H5_3v_3sp." << endl;
2471 }
2472 
2473 void FuxTest::test_1H5_2v_4sp() {
2474     cout << "Start test 1H5_2v_4sp ..." << endl;
2475     spList = {FOURTH_SPECIES};
2476     v_type = {0};
2477     for (size_t i = 1; i < cfSize-1; i++) {
```

```
2478        auto* problem = create_problem(cantusFirmus, spList, v_type,
              ↪ melodic_params, general_params, specific_params, importance,
              ↪ borrowMode);
2479        rel(problem->getHome(), problem->getSolutionArray()[i*2-1], IRT_EQ,
              ↪ cantusFirmus[i]); // fix the solution note as the same note as the
              ↪  cf
2480        if (has_solution(problem)) {
2481            std::cerr   << "/!\\ ERROR test 1H5_2v_4sp: It exists solution but it
                  ↪ shouldn't with the following configuration:\n"
2482                        << "cantus firmus: ";
2483            printVector(cantusFirmus);
2484            std::cerr   << "solution array: ";
2485            printIntVarArray(problem->getSolutionArray());
2486            std::cerr   << ">Same note played at the mesure: " << i << std::endl
                  ↪ ;
2487        }
2488        delete problem;
2489    }
2490    cout << "End test 1H5_2v_4sp" << endl;
2491 }
2492
2493 void FuxTest::test_1H5_3v_4sp() {
2494    cout << "Start test 1H5_3v_4sp..." << endl;
2495    int cpSize = cfSize*2-2; // size of a counterpoint
2496    spList = {FOURTH_SPECIES, FOURTH_SPECIES};
2497    v_type = {0, 0};
2498    // // test solution lines != cf
2499    for (size_t j = 0; j < 2; j++) { // iterate over each solution line
2500        for (size_t i = 1; i < cfSize-1; i++) { // iterate over each note of the
              ↪ cantus firmus
2501            auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪ melodic_params, general_params, specific_params, importance,
                  ↪ borrowMode);
2502            rel(problem->getHome(), problem->getSolutionArray()[(j*cpSize) + (i
                  ↪ *2)-2], IRT_EQ, cantusFirmus[i]); // fix the solution note as
                  ↪ the same note as the cf
2503            if (has_solution(problem)) {
2504                std::cerr   << "/!\\ ERROR test 1H5_3v_4sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
2505                            << "cantus firmus: ";
2506                printVector(cantusFirmus);
2507                std::cerr   << "solution array: ";
2508                printIntVarArray(problem->getSolutionArray());
2509                std::cerr   << ">Same note played at the mesure: " << i << std::
                      ↪ endl;
2510            }
2511            delete problem;
2512        }
2513    }
2514    // test notes of solutions lines are different
2515    for (size_t i = 1; i < cfSize-1; i++) {
2516        for (int j = -6; j < 13; j++) { // iterate over each possible note of
              ↪ solution lines
2517            auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪ melodic_params, general_params, specific_params, importance,
                  ↪ borrowMode);
2518            int note = cantusFirmus[0] + j;
2519            rel(problem->getHome(), problem->getSolutionArray()[i*2-2], IRT_EQ,
                  ↪ note); // fix the note of the first solution line as the same
                  ↪ note as the note of the second solution line
2520            rel(problem->getHome(), problem->getSolutionArray()[cpSize+i*2-2],
                  ↪ IRT_EQ, note); // fix the note of the first solution line as
                  ↪ the same note as the note of the second solution line
2521            if (has_solution(problem)) {
2522                std::cerr   << "/!\\ ERROR test 1H5_3v_4sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
2523                            << "cantus firmus: ";
2524                printVector(cantusFirmus);
```

```
2525                 std::cerr   << "solution array: ";
2526                 printIntVarArray(problem->getSolutionArray());
2527                 std::cerr   << "> Same note played at the mesure: " << i << std::
                       ↪  endl;
2528             }
2529             delete problem;
2530         }
2531     }
2532     cout << "End test 1H5_3v_4sp." << endl;
2533 }
2534
2535 // /*
2536 // For thesis notes, imperfect consonances are preferred to perfect consonances,
       ↪  and fifths are preferred to octaves.
2537 // */
2538 // void FuxTest::test_1H6() {
2539 //     cout << "Start test 1H6..." << endl;
2540 //     test_1H6_2v_1sp();
2541 //     cout << "End test 1H6." << endl;
2542 // }
2543
2544 // void FuxTest::test_1H6_2v_1sp() {
2545 //     cout << "test_1H6_2v_1sp" << endl;
2546 //     spList = {FIRST_SPECIES};
2547 //     v_type = {0};
2548 //     auto* problem = create_problem(cantusFirmus, spList, v_type,
       ↪  melodic_params, general_params, specific_params, importance, borrowMode);
2549 //     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 60);
2550 //     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 54);
2551 //     std::vector<CounterpointProblem*> solutions = get_all_solutions(problem);
2552 //     for (CounterpointProblem* solution : solutions) {
2553 //         cout << ">> Solution: " << solution->getSolutionArray() << endl;
2554 //         cout << ">> Costs: " << solution->cost() << endl;
2555 //     }
2556 //     delete problem;
2557 // }
2558
2559
2560 /*
2561 In two voice composition, the harmonic interval of the penultimate note must be
2562 a major sixth or a minor third depending on whether or not the cantus firmus is
       ↪  the lowest stratum.
2563 In three voice composition, that harmonic interval must be either a minor third,
       ↪  a perfect fifth, a major sixth or an octave.
2564 */
2565 void FuxTest::test_1H7() {
2566     cout << "Start test 1H7..." << endl;
2567     test_1H7_2v_1sp();
2568     test_1H7_3v_1sp();
2569     test_1H7_2v_2sp();
2570     test_1H7_3v_2sp();
2571     test_1H7_2v_3sp();
2572     test_1H7_3v_3sp();
2573     cout << "End test 1H7." << endl;
2574 }
2575
2576 void FuxTest::test_1H7_2v_1sp() {
2577     cout << "Start test 1H7_2v_1sp..." << endl;
2578     // Test with cf as the lowest stratum
2579     spList = {FIRST_SPECIES};
2580     v_type = {3};
2581     for (size_t interval = 0; interval < 12; interval++) {
2582         auto* problem = create_problem(cantusFirmus, spList, v_type,
           ↪  melodic_params, general_params, specific_params, importance,
           ↪  borrowMode);
2583         if (interval == 9) { // correct interval
2584             rel(problem->getHome(), problem->getSolutionArray()[cfSize-2], IRT_EQ
               ↪  , cantusFirmus[cfSize-2]+12+interval); // fix the penultimate
```

```
                ↪ solution note as the major sixth with the cf
2585            if (!has_solution(problem)) {
2586                std::cerr   << "/!\\ ERROR test 1H7_2v_1sp: It doesn't exist a
                      ↪ solution but it should with the following configuration:\n
                      ↪ "
2587                        << "Cantus firmus: ";
2588                printVector(cantusFirmus);
2589                std::cerr   << "Solution array: ";
2590                printIntVarArray(problem->getSolutionArray());
2591                std::cerr   << ">The penultimate note should be correct" << std
                      ↪ ::endl;
2592            }
2593        } else {
2594            rel(problem->getHome(), problem->getSolutionArray()[cfSize-2], IRT_EQ
                  ↪ , cantusFirmus[cfSize-2]+12+interval); // fix the penultimate
                  ↪ solution note as a wrong interval with the cf
2595            if (has_solution(problem)) {
2596                std::cerr   << "/!\\ ERROR test 1H7_2v_1sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
2597                        << "cantus firmus: ";
2598                printVector(cantusFirmus);
2599                std::cerr   << "solution array: ";
2600                printIntVarArray(problem->getSolutionArray());
2601                std::cerr   << ">The harmonic interval of the penultimate note
                      ↪ must be a major sixth" << std::endl;
2602            }
2603        }
2604        delete problem;
2605    }
2606    // Test with cf as the highest stratum
2607    v_type = {-2};
2608    for (size_t interval = 0; interval < 12; interval++) {
2609        auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪ melodic_params, general_params, specific_params, importance,
                  ↪ borrowMode);
2610        if (interval == 3) { // correct interval
2611            rel(problem->getHome(), problem->getSolutionArray()[cfSize-2], IRT_EQ
                  ↪ , cantusFirmus[cfSize-2]-interval); // fix the penultimate
                  ↪ solution note as the major sixth with the cf
2612            if (!has_solution(problem)) {
2613                std::cerr   << "/!\\ ERROR test 1H7_2v_1sp: It doesn't exist a
                      ↪ solution but it should with the following configuration:\n
                      ↪ "
2614                        << "Cantus firmus: ";
2615                printVector(cantusFirmus);
2616                std::cerr   << "Solution array: ";
2617                printIntVarArray(problem->getSolutionArray());
2618                std::cerr   << ">The penultimate note should be correct" << std
                      ↪ ::endl;
2619            }
2620        } else {
2621            rel(problem->getHome(), problem->getSolutionArray()[cfSize-2], IRT_EQ
                  ↪ , cantusFirmus[cfSize-2]-interval); // fix the penultimate
                  ↪ solution note as a wrong interval with the cf
2622            if (has_solution(problem)) {
2623                std::cerr   << "/!\\ ERROR test 1H7_2v_1sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
2624                        << "cantus firmus: ";
2625                printVector(cantusFirmus);
2626                std::cerr   << "solution array: ";
2627                printIntVarArray(problem->getSolutionArray());
2628                std::cerr   << ">The harmonic interval of the penultimate note
                      ↪ must be a minor third" << std::endl;
2629            }
2630        }
2631        delete problem;
2632    }
2633    cout << "End test 1H7_2v_1sp." << endl;
```

```
}

void FuxTest::test_1H7_3v_1sp() {
    cout << "Start test 1H7 3v 1sp..." << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    v_type = {3, 3};
    vector<int> intervals = {1, 2, 4, 5, 6, 8, 10, 11}; // forbidden intervals
    for (size_t i = 1; i < 3; i++) {
        for (int interval : intervals) {
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪ melodic_params, general_params, specific_params, importance,
          ↪ borrowMode);
        rel(problem->getHome(), problem->getSolutionArray()[cfSize*i-2], IRT_EQ,
          ↪ cantusFirmus[cfSize-2]+12+interval); // fix the penultimate
          ↪ solution note as a wrong interval with the cf
        if (has_solution(problem)) {
            std::cerr   << "/!\\ ERROR test 1H7 3v 1sp: It exists solution but it
              ↪ shouldn't with the following configuration:\n"
                  << "cantus firmus: ";
            printVector(cantusFirmus);
            std::cerr   << "solution array: ";
            printIntVarArray(problem->getSolutionArray());
            std::cerr   << "> The harmonic interval of the penultimate note must
              ↪ be either a minor third, a perfect fifth, a major sixth or an
              ↪ octave." << std::endl;
        }
        delete problem;
        }
    }
    cout << "End test 1H7 3v 1sp." << endl;
}

void FuxTest::test_1H7_2v_2sp() {
    cout << "Start test 1H7 2v 2sp..." << endl;
    int cpSize = cfSize*2-1; // size of a counterpoint
    // Test with cf as the lowest stratum
    spList = {SECOND_SPECIES};
    v_type = {3};
    for (size_t interval = 0; interval < 12; interval++) {
        auto* problem = create_problem(cantusFirmus, spList, v_type,
          ↪ melodic_params, general_params, specific_params, importance,
          ↪ borrowMode);
        if (interval == 9) { // correct interval
            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ
              ↪ , cantusFirmus[cfSize-2]+12+interval); // fix the penultimate
              ↪ solution note as the major sixth with the cf
            if (!has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H7 2v 2sp: It doesn't exist a
                  ↪ solution but it should with the following configuration:\n
                  ↪ "
                      << "Cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "Solution array: ";
                printIntVarArray(problem->getSolutionArray());
                std::cerr   << "> The penultimate note should be correct" << std
                  ↪ ::endl;
            }
        } else {
            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ
              ↪ , cantusFirmus[cfSize-2]+12+interval); // fix the penultimate
              ↪ solution note as a wrong interval with the cf
            if (has_solution(problem)) {
                std::cerr   << "/!\\ ERROR test 1H7 2v 2sp: It exists solution
                  ↪ but it shouldn't with the following configuration:\n"
                      << "cantus firmus: ";
                printVector(cantusFirmus);
                std::cerr   << "solution array: ";
                printIntVarArray(problem->getSolutionArray());
```

```
2685                 std::cerr   << ">␣The␣harmonic␣interval␣of␣the␣penultimate␣note␣must␣be␣a␣major␣sixth" << std::endl;
2686             }
2687         }
2688         delete problem;
2689     }
2690     // Test with cf as the highest stratum
2691     v_type = {-2};
2692     for (size_t interval = 0; interval < 12; interval++) {
2693         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
2694         if (interval == 3) { // correct interval
2695             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[cfSize-2]-interval); // fix the penultimate solution note as the major sixth with the cf
2696             if (!has_solution(problem)) {
2697                 std::cerr   << "/!\␣ERROR␣test␣1H7_2v_2sp:␣It␣doesn't␣exist␣a␣solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
2698                       << "Cantus␣firmus:␣";
2699                 printVector(cantusFirmus);
2700                 std::cerr   << "Solution␣array:␣";
2701                 printIntVarArray(problem->getSolutionArray());
2702                 std::cerr   << ">␣The␣penultimate␣note␣should␣be␣correct" << std::endl;
2703             }
2704         } else {
2705             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[cfSize-2]-interval); // fix the penultimate solution note as a wrong interval with the cf
2706             if (has_solution(problem)) {
2707                 std::cerr   << "/!\␣ERROR␣test␣1H7_2v_2sp:␣It␣exists␣solution␣but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
2708                       << "cantus␣firmus:␣";
2709                 printVector(cantusFirmus);
2710                 std::cerr   << "solution␣array:␣";
2711                 printIntVarArray(problem->getSolutionArray());
2712                 std::cerr   << ">␣The␣harmonic␣interval␣of␣the␣penultimate␣note␣must␣be␣a␣minor␣third" << std::endl;
2713             }
2714         }
2715         delete problem;
2716     }
2717     cout << "End␣test␣1H7_2v_2sp." << endl;
2718 }
2719
2720 void FuxTest::test_1H7_3v_2sp() {
2721     cout << "Start␣test␣1H7_3v_2sp..." << endl;
2722     int cpSize = cfSize*2-1; // size of a counterpoint
2723     spList = {SECOND_SPECIES, SECOND_SPECIES};
2724     v_type = {3, 3};
2725     vector<int> intervals = {1, 2, 4, 5, 6, 8, 10, 11}; // forbidden intervals
2726     for (size_t i = 1; i < 3; i++) {
2727         for (int interval : intervals) {
2728             auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
2729             rel(problem->getHome(), problem->getSolutionArray()[cpSize*i-2], IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as a wrong interval with the cf
2730             if (has_solution(problem)) {
2731                 std::cerr   << "/!\␣ERROR␣test␣1H7_3v_2sp:␣It␣exists␣solution␣but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
2732                       << "cantus␣firmus:␣";
2733                 printVector(cantusFirmus);
2734                 std::cerr   << "solution␣array:␣";
2735                 printIntVarArray(problem->getSolutionArray());
```

```
2736                 std::cerr   << ">␣The␣harmonic␣interval␣of␣the␣penultimate␣note␣must␣be␣either␣a␣minor␣third,␣a␣perfect␣fifth,␣a␣major␣sixth␣or␣an␣octave.␣" << std::endl;
2737             }
2738             delete problem;
2739         }
2740     }
2741     cout << "End␣test␣1H7_3v_2sp." << endl;
2742 }
2743
2744 void FuxTest::test_1H7_2v_3sp() {
2745     cout << "Start␣test␣1H7_2v_3sp..." << endl;
2746     int cpSize = cfSize*4-3; // size of a counterpoint
2747     // Test with cf as the lowest stratum
2748     spList = {THIRD_SPECIES};
2749     v_type = {3};
2750     for (size_t interval = 0; interval < 12; interval++) {
2751         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
2752         if (interval == 9) { // correct interval
2753             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as the major sixth with the cf
2754             if (!has_solution(problem)) {
2755                 std::cerr   << "/!\\␣ERROR␣test␣1H7_2v_3sp:␣It␣doesn't␣exist␣a␣solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
2756                       << "Cantus␣firmus:␣";
2757                 printVector(cantusFirmus);
2758                 std::cerr   << "Solution␣array:␣";
2759                 printIntVarArray(problem->getSolutionArray());
2760                 std::cerr   << ">␣The␣penultimate␣note␣should␣be␣correct" << std::endl;
2761             }
2762         } else {
2763             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the penultimate solution note as a wrong interval with the cf
2764             if (has_solution(problem)) {
2765                 std::cerr   << "/!\\␣ERROR␣test␣1H7_2v_3sp:␣It␣exists␣solution␣but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
2766                       << "cantus␣firmus:␣";
2767                 printVector(cantusFirmus);
2768                 std::cerr   << "solution␣array:␣";
2769                 printIntVarArray(problem->getSolutionArray());
2770                 std::cerr   << ">␣The␣harmonic␣interval␣of␣the␣penultimate␣note␣must␣be␣a␣major␣sixth" << std::endl;
2771             }
2772         }
2773         delete problem;
2774     }
2775     // Test with cf as the highest stratum
2776     v_type = {-2};
2777     for (size_t interval = 0; interval < 12; interval++) {
2778         auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params, general_params, specific_params, importance, borrowMode);
2779         if (interval == 3) { // correct interval
2780             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ, cantusFirmus[cfSize-2]-interval); // fix the penultimate solution note as the major sixth with the cf
2781             if (!has_solution(problem)) {
2782                 std::cerr   << "/!\\␣ERROR␣test␣1H7_2v_3sp:␣It␣doesn't␣exist␣a␣solution␣but␣it␣should␣with␣the␣following␣configuration:\n"
2783                       << "Cantus␣firmus:␣";
2784                 printVector(cantusFirmus);
```

```
2785                 std::cerr   << "Solution␣array:␣";
2786                 printIntVarArray(problem->getSolutionArray());
2787                 std::cerr   << ">␣The␣penultimate␣note␣should␣be␣correct" << std
                       ↪  ::endl;
2788             }
2789         } else {
2790             rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ
                   ↪  , cantusFirmus[cfSize-2]-interval); // fix the penultimate
                   ↪  solution note as a wrong interval with the cf
2791             if (has_solution(problem)) {
2792                 std::cerr   << "/!\\␣ERROR␣test␣1H7_2v_3sp:␣It␣exists␣solution␣
                       ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
2793                       << "cantus␣firmus:␣";
2794                 printVector(cantusFirmus);
2795                 std::cerr   << "solution␣array:␣";
2796                 printIntVarArray(problem->getSolutionArray());
2797                 std::cerr   << ">␣The␣harmonic␣interval␣of␣the␣penultimate␣note␣
                       ↪  must␣be␣a␣minor␣third" << std::endl;
2798             }
2799         }
2800         delete problem;
2801     }
2802     cout << "End␣test␣1H7_2v_3sp." << endl;
2803 }
2804
2805 void FuxTest::test_1H7_3v_3sp() {
2806     cout << "Start␣test␣1H7_3v_3sp..." << endl;
2807     int cpSize = cfSize*4-3; // size of a counterpoint
2808     spList = {THIRD_SPECIES, THIRD_SPECIES};
2809     v_type = {3, 3};
2810     vector<int> intervals = {1, 2, 4, 5, 6, 8, 10, 11}; // forbidden intervals
2811     for (size_t i = 1; i < 3; i++) {
2812         for (int interval : intervals) {
2813             auto* problem = create_problem(cantusFirmus, spList, v_type,
                   ↪  melodic_params, general_params, specific_params, importance,
                   ↪  borrowMode);
2814             rel(problem->getHome(), problem->getSolutionArray()[cpSize*i-2],
                   ↪  IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the
                   ↪  penultimate solution note as a wrong interval with the cf
2815             if (has_solution(problem)) {
2816                 std::cerr   << "/!\\␣ERROR␣test␣1H7_3v_3sp:␣It␣exists␣solution␣
                       ↪  but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
2817                       << "cantus␣firmus:␣";
2818                 printVector(cantusFirmus);
2819                 std::cerr   << "solution␣array:␣";
2820                 printIntVarArray(problem->getSolutionArray());
2821                 std::cerr   << ">␣The␣harmonic␣interval␣of␣the␣penultimate␣note␣
                       ↪  must␣be␣either␣a␣minor␣third,␣a␣perfect␣fifth,␣a␣major␣
                       ↪  sixth␣or␣an␣octave.␣" << std::endl;
2822             }
2823             delete problem;
2824         }
2825     }
2826     cout << "End␣test␣1H7_3v_3sp." << endl;
2827 }
2828
2829 void FuxTest::test_1H7_2v_4sp() {
2830     cout << "Start␣test␣1H7_2v_4sp..." << endl;
2831     int cpSize = cfSize*2-2; // size of a counterpoint
2832     // Test with cf as the lowest stratum
2833     spList = {FOURTH_SPECIES};
2834     v_type = {3};
2835     for (size_t interval = 0; interval < 12; interval++) {
2836         auto* problem = create_problem(cantusFirmus, spList, v_type,
                   ↪  melodic_params, general_params, specific_params, importance,
                   ↪  borrowMode);
2837         if (interval == 9) { // correct interval
```

```
2838            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ
                  ↪ , cantusFirmus[cfSize-2]+12+interval); // fix the penultimate
                  ↪ solution note as the major sixth with the cf
2839            if (!has_solution(problem)) {
2840                std::cerr   << "/!\\ ERROR test 1H7_2v_4sp: It doesn't exist a
                      ↪ solution but it should with the following configuration:\n
                      ↪ "
2841                            << "Cantus firmus: ";
2842                printVector(cantusFirmus);
2843                std::cerr   << "Solution array: ";
2844                printIntVarArray(problem->getSolutionArray());
2845                std::cerr   << "> The penultimate note should be correct" << std
                      ↪ ::endl;
2846            }
2847        } else {
2848            rel(problem->getHome(), problem->getSolutionArray()[cpSize-2], IRT_EQ
                  ↪ , cantusFirmus[cfSize-2]+12+interval); // fix the penultimate
                  ↪ solution note as a wrong interval with the cf
2849            if (has_solution(problem)) {
2850                std::cerr   << "/!\\ ERROR test 1H7_2v_4sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
2851                            << "cantus firmus: ";
2852                printVector(cantusFirmus);
2853                std::cerr   << "solution array: ";
2854                printIntVarArray(problem->getSolutionArray());
2855                std::cerr   << "> The harmonic interval of the penultimate note
                      ↪ must be a major sixth" << std::endl;
2856            }
2857        }
2858        delete problem;
2859    }
2860    cout << "End test 1H7_2v_4sp." << endl;
2861 }
2862
2863 void FuxTest::test_1H7_3v_4sp() {
2864    cout << "Start test 1H7_3v_4sp..." << endl;
2865    int cpSize = cfSize*2-2; // size of a counterpoint
2866    spList = {THIRD_SPECIES, THIRD_SPECIES};
2867    v_type = {3, 3};
2868    vector<int> intervals = {1, 2, 4, 5, 6, 8, 10, 11}; // forbidden intervals
2869    for (size_t i = 1; i < 3; i++) {
2870        for (int interval : intervals) {
2871            auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪ melodic_params, general_params, specific_params, importance,
                  ↪ borrowMode);
2872            rel(problem->getHome(), problem->getSolutionArray()[cpSize*i-2],
                  ↪ IRT_EQ, cantusFirmus[cfSize-2]+12+interval); // fix the
                  ↪ penultimate solution note as a wrong interval with the cf
2873            if (has_solution(problem)) {
2874                std::cerr   << "/!\\ ERROR test 1H7_3v_4sp: It exists solution
                      ↪ but it shouldn't with the following configuration:\n"
2875                            << "cantus firmus: ";
2876                printVector(cantusFirmus);
2877                std::cerr   << "solution array: ";
2878                printIntVarArray(problem->getSolutionArray());
2879                std::cerr   << "> The harmonic interval of the penultimate note
                      ↪ must be either a minor third, a perfect fifth, a major
                      ↪ sixth or an octave. " << std::endl;
2880            }
2881            delete problem;
2882        }
2883    }
2884    cout << "End test 1H7_3v_4sp." << endl;
2885 }
2886
2887 // In three voice composition, tenths are prohibited in the last chord.
2888 void FuxTest::test_1H10(){
2889    cout << "Start test 1H10..." << endl;
```

```
2890     test_1H10_3v_1sp();
2891     test_1H10_3v_2sp();
2892     test_1H10_3v_3sp();
2893     test_1H10_3v_4sp();
2894     cout << "End test 1H10." << endl;
2895 }
2896
2897 void FuxTest::test_1H10_3v_1sp(){
2898     cout << "Start test_1H10_3v_1sp..." << endl;
2899     spList = {FIRST_SPECIES, FIRST_SPECIES};
2900     v_type = {3, 3};
2901     int intervals[] = {3, 4}; // forbidden intervals
2902     for (size_t i = 1; i < 3; i++) { // iterate over each counter point
2903         for (int interval: intervals) { // interval tested
2904             auto* problem = create_problem(cantusFirmus, spList, v_type,
                    ↪ melodic_params, general_params, specific_params, importance,
                    ↪ borrowMode);
2905             int note = cantusFirmus[cfSize-1] + 12 + interval;
2906             rel(problem->getHome(), problem->getSolutionArray()[(i*cfSize)-1],
                    ↪ IRT_EQ, note); // fix the last note as a thens
2907             if (has_solution(problem)) {
2908                 std::cerr   << "/!\\ ERROR test 1H10_3v_1sp: It exists solution
                        ↪ but it shouldn't with the following configuration:\n"
2909                             << "cantus firmus: ";
2910                 printVector(cantusFirmus);
2911                 std::cerr   << "solution array: ";
2912                 printIntVarArray(problem->getSolutionArray());
2913                 std::cerr   << ">A thenth is played in the last chord" << std::
                        ↪ endl;
2914             }
2915             delete problem;
2916
2917         }
2918     }
2919     cout << "End test 1H10_3v_1sp" << endl;
2920 }
2921
2922 void FuxTest::test_1H10_3v_2sp() {
2923     cout << "Start test_1H10_3v_2sp..." << endl;
2924     int cpSize = cfSize*2-1; // size of a counterpoint
2925     spList = {SECOND_SPECIES, SECOND_SPECIES};
2926     v_type = {3, 3};
2927     int intervals[] = {3, 4}; // forbidden intervals
2928     for (size_t i = 1; i < 3; i++) { // iterate over each counter point
2929         for (int interval: intervals) { // interval tested
2930             auto* problem = create_problem(cantusFirmus, spList, v_type,
                    ↪ melodic_params, general_params, specific_params, importance,
                    ↪ borrowMode);
2931             int note = cantusFirmus[cfSize-1] + 12 + interval;
2932             rel(problem->getHome(), problem->getSolutionArray()[(i*cpSize)-1],
                    ↪ IRT_EQ, note); // fix the last note as a thens
2933             if (has_solution(problem)) {
2934                 std::cerr   << "/!\\ ERROR test 1H10_3v_2sp: It exists solution
                        ↪ but it shouldn't with the following configuration:\n"
2935                             << "cantus firmus: ";
2936                 printVector(cantusFirmus);
2937                 std::cerr   << "solution array: ";
2938                 printIntVarArray(problem->getSolutionArray());
2939                 std::cerr   << ">A thenth is played in the last chord" << std::
                        ↪ endl;
2940             }
2941             delete problem;
2942         }
2943     }
2944     cout << "End test_1H10_3v_2sp" << endl;
2945 }
2946
2947 void FuxTest::test_1H10_3v_3sp() {
```

```
2948     cout << "Start␣test_1H10_3v_3sp..." << endl;
2949     int cpSize = cfSize*4-3; // size of a counterpoint
2950     spList = {THIRD_SPECIES, THIRD_SPECIES};
2951     v_type = {3, 3};
2952     int intervals[] = {3, 4}; // forbidden intervals
2953     for (size_t i = 1; i < 3; i++) { // iterate over each counter point
2954         for (int interval: intervals) { // interval tested
2955             auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪ melodic_params, general_params, specific_params, importance,
                  ↪ borrowMode);
2956             int note = cantusFirmus[cfSize-1] + 12 + interval;
2957             rel(problem->getHome(), problem->getSolutionArray()[(i*cpSize)-1],
                  ↪ IRT_EQ, note); // fix the last note as a thens
2958             if (has_solution(problem)) {
2959                 std::cerr   << "/!\\␣ERROR␣test␣1H10_3v_3sp:␣It␣exists␣solution␣
                      ↪ but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
2960                             << "cantus␣firmus:␣";
2961                 printVector(cantusFirmus);
2962                 std::cerr   << "solution␣array:␣";
2963                 printIntVarArray(problem->getSolutionArray());
2964                 std::cerr   << ">␣A␣thenth␣is␣played␣in␣the␣last␣chord" << std::
                      ↪ endl;
2965             }
2966             delete problem;
2967         }
2968     }
2969     cout << "End␣test_1H10_3v_3sp" << endl;
2970 }
2971
2972 void FuxTest::test_1H10_3v_4sp() {
2973     cout << "Start␣test_1H10_3v_4sp..." << endl;
2974     int cpSize = cfSize*2-2; // size of a counterpoint
2975     spList = {FOURTH_SPECIES, FOURTH_SPECIES};
2976     v_type = {3, 3};
2977     int intervals[] = {3, 4}; // forbidden intervals
2978     for (size_t i = 1; i < 3; i++) { // iterate over each counter point
2979         for (int interval: intervals) { // interval tested
2980             auto* problem = create_problem(cantusFirmus, spList, v_type,
                  ↪ melodic_params, general_params, specific_params, importance,
                  ↪ borrowMode);
2981             int note = cantusFirmus[cfSize-1] + 12 + interval;
2982             rel(problem->getHome(), problem->getSolutionArray()[(i*cpSize)-1],
                  ↪ IRT_EQ, note); // fix the last note as a thens
2983             if (has_solution(problem)) {
2984                 std::cerr   << "/!\\␣ERROR␣test␣1H10_3v_4sp:␣It␣exists␣solution␣
                      ↪ but␣it␣shouldn't␣with␣the␣following␣configuration:\n"
2985                             << "cantus␣firmus:␣";
2986                 printVector(cantusFirmus);
2987                 std::cerr   << "solution␣array:␣";
2988                 printIntVarArray(problem->getSolutionArray());
2989                 std::cerr   << ">␣A␣thenth␣is␣played␣in␣the␣last␣chord" << std::
                      ↪ endl;
2990             }
2991             delete problem;
2992         }
2993     }
2994     cout << "End␣test_1H10_3v_4sp" << endl;
2995 }
2996
2997 // Melodic intervals cannot exceed a minor sixth interval, but octave leaps are
      ↪ allowed in three and four voices.
2998 void FuxTest::test_1M2() {
2999     cout << "Start␣test␣1M2..." << endl;
3000     test_1M2_2v_1sp();
3001     test_1M2_3v_1sp();
3002     cout << "End␣test␣1M2." << endl;
3003 }
3004
```

```
3005 void FuxTest::test_1M2_2v_1sp() {
3006     cout << "Start␣test␣1M2_2v_1sp" << endl;
3007     spList = {FIRST_SPECIES};
3008     v_type = {0};
3009     for (size_t i = 1; i < cfSize; i++) {
3010         auto* problem = create_problem(cantusFirmus, spList, v_type,
                ↪  melodic_params, general_params, specific_params, importance,
                ↪  borrowMode);
3011         rel(problem->getHome(), expr(problem->getHome(), abs(problem->
                ↪  getSolutionArray()[i] - problem->getSolutionArray()[i-1])), IRT_GR
                ↪  , 8); // fix the melodic interval between note i and i-1 greater
                ↪  than 8
3012         if (has_solution(problem)) {
3013             std::cerr   << "/!\\␣ERROR␣test␣1M2_2v_1sp:␣It␣exists␣solution␣but␣it
                    ↪  ␣shouldn't␣with␣the␣following␣configuration:\n"
3014                         << "cantus␣firmus:␣";
3015             printVector(cantusFirmus);
3016             std::cerr   << "solution␣array:␣";
3017             printIntVarArray(problem->getSolutionArray());
3018             std::cerr   << ">␣Same␣note␣played␣at␣the␣mesure:␣" << i << std::endl
                    ↪  ;
3019         }
3020         delete problem;
3021     }
3022     cout << "End␣test␣1M2_2v_1sp" << endl;
3023 }
3024 
3025 void FuxTest::test_1M2_3v_1sp() {
3026     cout << "Start␣test␣1M2_3v_1sp..." << endl;
3027     spList = {FIRST_SPECIES, FIRST_SPECIES};
3028     v_type = {2, 2};
3029     // Add the constraint that the interval between note i and i-1 is greater
          ↪  than 8 and different from 12
3030     for (size_t i = 1; i < cfSize; i++) {
3031         auto* problem = create_problem(cantusFirmus, spList, v_type,
                ↪  melodic_params, general_params, specific_params, importance,
                ↪  borrowMode);
3032         cout << "Add␣constraint" << endl;
3033         IntVar absDiff = expr(problem->getHome(), abs(problem->getSolutionArray()
                ↪  [i] - problem->getSolutionArray()[i-1]));
3034         rel(problem->getHome(), absDiff, IRT_GR, 8);
3035         rel(problem->getHome(), absDiff, IRT_NQ, 12);
3036         cout << "Constraint␣added" << endl;
3037         if (has_solution(problem)) {
3038             std::cerr   << "/!\\␣ERROR␣test␣1M2_2v_1sp:␣It␣exists␣solution␣but␣it
                    ↪  ␣shouldn't␣with␣the␣following␣configuration:\n"
3039                         << "cantus␣firmus:␣";
3040             printVector(cantusFirmus);
3041             std::cerr   << "solution␣array:␣";
3042             printIntVarArray(problem->getSolutionArray());
3043             std::cerr   << ">␣Same␣note␣played␣at␣the␣mesure:␣" << i << std::endl
                    ↪  ;
3044         }
3045         delete problem;
3046     }
3047     cout << "End␣test␣1M2_3v_1sp" << endl;
3048 }
3049 
3050 // void FuxTest::test_1P1_2v_1sp() {
3051 //     cout << "Start test 1P1_2v_1sp..." << endl;
3052 //     spList = {FIRST_SPECIES};
3053 //     v_type = {0};
3054 //     int cons_intervals[] = {0, 3, 4, 7, 8, 9}; // consonant intervals
3055 //     for (size_t i = 1; i < cfSize; i++) {
3056 //         auto* problem = create_problem(cantusFirmus, spList, v_type,
      ↪  melodic_params, general_params, specific_params, importance, borrowMode);
3057 //         // Add the constraint that the note at index i is in cons_intervals
```

```
3058 //         Gecode::IntSet consSet(cons_intervals, sizeof(cons_intervals) / sizeof
       ↪ (cons_intervals[0]));
3059 //         dom(problem->getHome(), problem->getSolutionArray()[i], consSet);
3060
3061 //         BoolVar bothUp = expr(problem->getHome(), part1->getMelodicIntervals()
       ↪ [i] > 0 && part2->getMelodicIntervals()[i] > 0);
3062 //         BoolVar bothDown = expr(home, part1->getMelodicIntervals()[i] < 0 &&
       ↪ part2->getMelodicIntervals()[i] < 0);
3063 //         BoolVar sameDirection = expr(home, bothUp || bothDown);
3064
3065 //         // Constrain the motion to be direct
3066 //         rel(home, sameDirection == 1);
3067
3068 //         rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ,
       ↪ cantusFirmus[i]); // fix the solution note as the same note as the cf
3069 //         if (has_solution(problem)) {
3070 //             std::cerr   << "/!\\ ERROR test 1P1_2v_1sp: It exists solution but
       ↪  it shouldn't with the following configuration:\n"
3071 //                         << "cantus firmus: ";
3072 //             printVector(cantusFirmus);
3073 //             std::cerr   << "solution array: ";
3074 //             printIntVarArray(problem->getSolutionArray());
3075 //             std::cerr   << "> Same note played at the mesure: " << i << std::
       ↪ endl;
3076 //         }
3077 //         problem->getSolutionArray().
3078 //         delete problem;
3079 //     }
3080 //     cout << "End test 1P1_2v_1sp" << endl;
3081 // }
3082
3083
3084 void FuxTest::test_2H2() {
3085     cout << "Start test 2H2..." << endl;
3086     test_2H2_2v_2sp();
3087     test_2H2_3v_2sp();
3088     test_2H2_4v_2sp();
3089     cout << "End test 2H2." << endl;
3090 }
3091
3092 void FuxTest::test_2H2_2v_2sp() {
3093     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
3094     int cons[] = {3, 4, 7, 8, 9}; // conssonant intervals without 0 because 1H5
3095     spList = {SECOND_SPECIES};
3096     v_type = {3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
3097     // Test that dissonant notes are forbidden
3098     for (int interval : dis) {
3099         //test first note
3100         CounterpointProblem* problem = create_problem(cantusFirmus, spList,
           ↪  v_type, melodic_params, general_params, specific_params,
           ↪  importance, borrowMode);
3101             int note = cantusFirmus[0] + 12 + interval; // note is dissonant
3102             rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, note)
               ↪  ; // fix the first note
3103             if (has_solution(problem)) {
3104                 std::cerr   << "/!\\ ERROR test 2H2_2v_2sp: It exists a solution
                   ↪  but it shouldn't with the following configuration:\n"
3105                             << "Cantus firmus: ";
3106                 printVector(cantusFirmus);
3107                 std::cerr   << "Solution array: ";
3108                 printIntVarArray(problem->getSolutionArray());
3109                 std::cerr   << "> Dissonant harmonic at the mesure " << 0 << std
                   ↪  ::endl;
3110             }
3111             delete problem;
3112         // test next notes
3113         for (size_t i = 1; i < cfSize; i++) {
```

```
3114            CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪ v_type, melodic_params, general_params, specific_params,
                  ↪ importance, borrowMode);
3115            int note = cantusFirmus[i] + 12 + interval; // note is dissonant
3116            rel(problem->getHome(), problem->getSolutionArray()[i*2], IRT_EQ,
                  ↪ note); // fix the note i
3117            if (has_solution(problem)) {
3118                std::cerr   << "/!\\ ERROR test 2H2_2v_2sp: It exists a solution
                      ↪ but it shouldn't with the following configuration:\n"
3119                            << "Cantus firmus: ";
3120                printVector(cantusFirmus);
3121                std::cerr   << "Solution array: ";
3122                printIntVarArray(problem->getSolutionArray());
3123                std::cerr   << ">Dissonant harmonic at the mesure " << i << std
                      ↪ ::endl;
3124            }
3125            delete problem;
3126        }
3127    }
3128 }
3129
3130 void FuxTest::test_2H2_3v_2sp() {
3131     cout << "Start test 2H2_3v_2sp..." << endl;
3132     int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
3133     int cons[] = {3, 4, 7, 8, 9}; // consonant intervals without 0 because 1H5
3134     spList = {SECOND_SPECIES, SECOND_SPECIES};
3135     v_type = {3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
3136     // Test that dissonant notes are forbidden
3137     for (int interval : dis) {
3138         for (size_t j = 0; j < 2; j++) { // iterate over each solution line
3139             //test first note
3140             CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                   ↪ v_type, melodic_params, general_params, specific_params,
                   ↪ importance, borrowMode);
3141             int note = cantusFirmus[0] + 12 + interval; // note is dissonant
3142             rel(problem->getHome(), problem->getSolutionArray()[j*((cfSize*2)-1)
                   ↪ ], IRT_EQ, note); // fix the first note
3143             if (has_solution(problem)) {
3144                 std::cerr   << "/!\\ ERROR test 2H2_3v_2sp: It exists a solution
                       ↪ but it shouldn't with the following configuration:\n"
3145                             << "Cantus firmus: ";
3146                 printVector(cantusFirmus);
3147                 std::cerr   << "Solution array: ";
3148                 printIntVarArray(problem->getSolutionArray());
3149                 std::cerr   << ">Dissonant harmonic at the mesure " << 0 << std
                       ↪ ::endl;
3150             }
3151             delete problem;
3152             // test next notes
3153             for (size_t i = 0; i < cfSize; i++) {
3154                 CounterpointProblem* problem = create_problem(cantusFirmus,
                       ↪ spList, v_type, melodic_params, general_params,
                       ↪ specific_params, importance, borrowMode);
3155                 int note = cantusFirmus[i] + 12 + interval; // note is dissonant
3156                 rel(problem->getHome(), problem->getSolutionArray()[j*cfSize+i],
                       ↪ IRT_EQ, note); // fix the note i
3157                 if (has_solution(problem)) {
3158                     std::cerr   << "/!\\ ERROR test 2H2_3v_2sp: It exists a
                           ↪ solution but it shouldn't with the following
                           ↪ configuration:\n"
3159                                 << "Cantus firmus: ";
3160                     printVector(cantusFirmus);
3161                     std::cerr   << "Solution array: ";
3162                     printIntVarArray(problem->getSolutionArray());
3163                     std::cerr   << ">Dissonant harmonic at the measure " << i <<
                           ↪  std::endl;
3164                 }
3165                 delete problem;
```

```
3166            }
3167        }
3168    }
3169}
3170
3171void FuxTest::test_2H2_4v_2sp() {
3172    cout << "Start test 2H2 4v 2sp..." << endl;
3173    int dis[] = {1, 2, 5, 6, 10, 11}; // dissonant intervals
3174    int cons[] = {3, 4, 7, 8, 9}; // consonant intervals without 0 because 1H5
3175    spList = {SECOND_SPECIES, SECOND_SPECIES, SECOND_SPECIES};
3176    v_type = {3, 3, 3}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
3177
3178    // Test that dissonant notes are forbidden
3179    for (int interval : dis) {
3180        for (size_t j = 0; j < 3; j++) { // iterate over each solution line
3181            //test first note
3182            CounterpointProblem* problem = create_problem(cantusFirmus, spList,
                  ↪ v_type, melodic_params, general_params, specific_params,
                  ↪ importance, borrowMode);
3183            int note = cantusFirmus[0] + 12 + interval; // note is dissonant
3184            rel(problem->getHome(), problem->getSolutionArray()[j*((cfSize*2)-1)
                  ↪ ], IRT_EQ, note); // fix the first note
3185            if (has_solution(problem)) {
3186                std::cerr   << "/!\\ ERROR test 2H2 4v 2sp: It exists a solution
                      ↪ but it shouldn't with the following configuration:\n"
3187                            << "Cantus firmus: ";
3188                printVector(cantusFirmus);
3189                std::cerr   << "Solution array: ";
3190                printIntVarArray(problem->getSolutionArray());
3191                std::cerr   << ">Dissonant harmonic at the mesure " << 0 << std
                      ↪ ::endl;
3192            }
3193            delete problem;
3194            // test next notes
3195            for (size_t i = 0; i < cfSize; i++) {
3196                CounterpointProblem* problem = create_problem(cantusFirmus,
                      ↪ spList, v_type, melodic_params, general_params,
                      ↪ specific_params, importance, borrowMode);
3197                int note = cantusFirmus[i] + 12 + interval; // note is dissonant
3198                rel(problem->getHome(), problem->getSolutionArray()[j*cfSize+i],
                      ↪ IRT_EQ, note); // fix the note i
3199                if (has_solution(problem)) {
3200                    std::cerr   << "/!\\ ERROR test 2H2 4v 2sp: It exists a
                          ↪ solution but it shouldn't with the following
                          ↪ configuration:\n"
3201                                << "Cantus firmus: ";
3202                    printVector(cantusFirmus);
3203                    std::cerr   << "Solution array: ";
3204                    printIntVarArray(problem->getSolutionArray());
3205                    std::cerr   << ">Dissonant harmonic at the measure " << i <<
                          ↪  std::endl;
3206                }
3207                delete problem;
3208            }
3209        }
3210    }
3211}
3212
3213void FuxTest::test_2M2() {
3214    cout << "Start test 2M2..." << endl;
3215    test_2M2_2v_2sp();
3216    cout << "End test 2M2." << endl;
3217}
3218
3219void FuxTest::test_2M2_2v_2sp() {
3220    cout << "Start test 2M2 2v 2sp" << endl;
3221    int cpSize = cfSize*2-1; // size of a counterpoint
3222    spList = {SECOND_SPECIES};
```

```
3223     v_type = {1}; // {(6 * v_type - 6) + cf[0], (6 * v_type + 12) + cf[0]}
3224     for (size_t i = 1; i < cpSize; i++) {
3225         auto* problem = create_problem(cantusFirmus, spList, v_type,
           ↪  melodic_params, general_params, specific_params, importance,
           ↪  borrowMode);
3226         cout << "1" << endl;
3227         rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, problem->
           ↪  getSolutionArray()[i-1]); // fix the note i a the same than i-1
3228         cout << "2" << endl;
3229         if (has_solution(problem)) {
3230             std::cerr   << "/!\\ ERROR test 2M2_2v_2sp: It exists solution but it
               ↪  shouldn't with the following configuration:\n"
3231                         << "cantus firmus: ";
3232             printVector(cantusFirmus);
3233             std::cerr   << "solution array: ";
3234             printIntVarArray(problem->getSolutionArray());
3235             std::cerr   << "Same note is played consecutively" << std::endl;
3236         }
3237         cout << "3" << endl;
3238         delete problem;
3239         cout << "4" << endl;
3240     }
3241     cout << "End test 2M2_2v_2sp" << endl;
3242 }
3243
3244 // ----- Figures tests -----
3245
3246 void FuxTest::test_2v_1sp_fig22() {
3247     cout << "Start test_2v_1sp_fig22" << endl;
3248     spList = {FIRST_SPECIES};
3249     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
3250     cp =           {69,64,67,65,64,72,69,71,71,69,68,69};
3251     v_type = {1};
3252     borrowMode = 1;
3253     test_configuration();
3254     cout << "End test_2v_1sp_fig22." << endl;
3255 }
3256
3257 void FuxTest::test_2v_1sp_fig23(){
3258     cout << "Start test_2v_1sp_fig23" << endl;
3259     spList = {FIRST_SPECIES};
3260     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
3261     cp =           {69,69,67,65,64,64,62,60,67,69,68,69};
3262     v_type = {1};
3263     borrowMode = 1;
3264     test_configuration();
3265     cout << "End test_2v_1sp_fig23." << endl;
3266 }
3267
3268 void FuxTest::test_2v_2sp_fig38(){
3269     cout << "Start test_2v_2sp_fig38" << endl;
3270     spList = {SECOND_SPECIES};
3271     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3272     cp =
       ↪  {-1,65,64,62,60,58,57,55,53,57,60,58,57,69,67,64,65,67,69,65,62,64,65};
       ↪
3273     v_type = {0};
3274     borrowMode = 1;
3275     test_configuration();
3276     cout << "End test_2v_2sp_fig38." << endl;
3277 }
3278
3279 void FuxTest::test_2v_2sp_fig39(){
3280     cout << "Start test_2v_2sp_fig39" << endl;
3281     spList = {SECOND_SPECIES};
3282     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
3283     cp =
       ↪  {-1,53,52,48,53,52,50,48,46,58,55,60,57,53,52,48,53,41,45,50,48,52,53};
```

```
        ↪
    v_type = {-2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_2sp_fig39." << endl;
}

void FuxTest::test_2v_2sp_fig40(){
    cout << "Start test_2v_2sp_fig40" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {-1,
        ↪  67,64,65,67,69,71,69,67,72,71,72,74,72,71,69,67,65,64,72,71,69,67,62,64,66,67};
        ↪
    v_type = {2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_2sp_fig40." << endl;
}

void FuxTest::test_2v_2sp_fig41(){
    cout << "Start test_2v_2sp_fig41" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =
        ↪  {-1,67,64,65,67,65,64,62,60,64,60,72,71,69,67,71,72,71,69,67,66,62,67,59,62,66,67};
        ↪
    v_type = {2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_2sp_fig41." << endl;
}

void FuxTest::test_2v_2sp_fig42(){
    cout << "Start test_2v_2sp_fig42" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {57,60,59,60,64,65,64,62,60,59,57};
    cp =
        ↪  {-1,69,64,65,67,62,64,76,72,71,69,65,67,71,74,69,72,64,66,68,69};
    v_type = {2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_2sp_fig42." << endl;
}

void FuxTest::test_2v_2sp_fig43(){
    cout << "Start test_2v_2sp_fig43" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
    cp =
        ↪  {-1,45,57,52,55,52,50,53,57,59,60,48,50,45,48,52,53,55,57,45,52,56,57};
        ↪
    v_type = {-1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_2sp_fig43." << endl;
}

void FuxTest::test_2v_2sp_fig44(){
    cout << "Start test_2v_2sp_fig44" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {60,64,65,67,64,69,67,64,65,64,62,60};
    cp =
        ↪  {-1,67,72,71,69,74,71,69,67,71,72,74,76,74,72,71,69,71,72,67,69,71,72};
        ↪
    v_type = {1};
    borrowMode = 1;
    test_configuration();
```

```
    cout << "End test_2v_2sp_fig44." << endl;
}

void FuxTest::test_2v_2sp_fig45(){
    cout << "Start test_2v_2sp_fig45" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {60,64,65,67,64,69,67,64,65,64,62,60};
    cp =
      ↪ {-1,60,72,71,69,74,71,67,72,71,69,72,76,74,72,69,74,69,72,60,67,71,72};
      ↪
    v_type = {1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_2sp_fig45." << endl;
}

void FuxTest::test_2v_3sp_fig55() {
    cout << "Start test_2v_3sp_fig55" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =
      ↪ {62,64,65,67,69,71,72,74,76,74,71,72,74,72,70,69,70,72,74,76,77,65,69,71,72,69,70,72,70,69,67,71,69,62,6
      ↪
    v_type = {2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_3sp_fig55." << endl;
}

void FuxTest::test_2v_3sp_fig56(){
    cout << "Start test_2v_3sp_fig56" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =
      ↪ {62,64,65,67,69,62,69,71,72,71,67,69,71,69,67,65,64,76,71,72,74,69,62,64,65,67,69,71,72,74,76,72,74,69,6
      ↪
    v_type = {1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_3sp_fig56." << endl;
}

void FuxTest::test_2v_3sp_fig57(){
    cout << "Start test_2v_3sp_fig57" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =
      ↪ {71,67,69,71,72,71,69,67,65,67,69,71,72,64,65,67,69,72,76,74,72,71,69,72,71,74,71,69,67,71,72,71,69,71,7
      ↪
    v_type = {2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_3sp_fig57." << endl;
}

void FuxTest::test_2v_3sp_fig58(){
    cout << "Start test_2v_3sp_fig58" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =
      ↪ {64,65,67,64,69,67,65,64,62,64,65,67,69,64,69,67,65,64,62,64,65,67,69,71,72,74,76,74,72,60,72,71,69,74,6
      ↪
    v_type = {1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_3sp_fig58." << endl;
}

```

```
void FuxTest::test_2v_3sp_fig59() {
    cout << "Start test_2v_3sp_fig59" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =
      ↪ {65,64,62,60,59,62,67,65,64,62,60,58,57,60,62,64,65,62,64,65,67,64,65,67,69,67,65,69,67,65,64,62,60,64,6
      ↪
    v_type = {-1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_3sp_fig59." << endl;
}


void FuxTest::test_2v_3sp_fig60() {
    cout << "Start test_2v_3sp_fig60" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =
      ↪ {62,74,72,71,72,69,71,72,74,72,71,69,71,69,66,65,66,74,78,77,76,69,81,78,77,76,74,71,72,69,71,72,74,72,7
      ↪
    v_type = {1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_3sp_fig60." << endl;
}

void FuxTest::test_2v_4sp_fig74() {
    cout << "Start test_2v_4sp_fig74" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {62,62,74,74,72,72,71,71,76,76,74,74,77,77,76,76,74,74,73,74};
    v_type = {1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_4sp_fig74." << endl;
}

void FuxTest::test_2v_4sp_fig75() {
    cout << "Start test_2v_4sp_fig75" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {76,76,72,72,71,72,64,64,65,65,72,72,71,71,76,76,74,76};
    v_type = {2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_4sp_fig75." << endl;
}

void FuxTest::test_2v_4sp_fig76() {
    cout << "Start test_2v_4sp_fig76" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {64,64,69,69,67,67,65,65,62,62,74,74,72,72,76,76,74,76};
    v_type = {0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_4sp_fig76." << endl;
}

void FuxTest::test_2v_4sp_fig77() {
    cout << "Start test_2v_4sp_fig77" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =
      ↪ {65,65,64,64,60,60,65,65,69,69,67,67,65,65,64,64,69,69,65,65,64,65};
    v_type = {0};
    borrowMode = 1;
```

```
    test_configuration();
    cout << "End test_2v_4sp_fig77." << endl;
}

void FuxTest::test_2v_4sp_fig78() {
    cout << "Start test_2v_4sp_fig78" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =
      ↪ {74,74,72,72,71,71,66,66,64,64,76,76,74,74,72,72,71,71,74,74,72,74};
    v_type = {0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_2v_4sp_fig78." << endl;
}

void FuxTest::test_3v_1sp_fig108(){
    cout << "Start test_3v_1sp_fig108" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {67,64,65,69,72,72,76,72,69,68,
                    64,69,62,65,65,65,72,72,74,76};
    v_type = {1, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig108." << endl;
}

void FuxTest::test_3v_1sp_fig109(){
    cout << "Start test_3v_1sp_fig109" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {52,48,50,48,45,57,55,52,53,52};
    cp =           {67,64,65,64,64,64,67,67,62,64,
                    71,72,69,69,72,72,71,71,69,71};
    v_type = {2, 3};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig109." << endl;
}

void FuxTest::test_3v_1sp_fig110(){
    cout << "Start test_3v_1sp_fig110" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {60,60,60,62,58,59,57,69,65,62,64,65,
                    53,52,53,50,55,55,53,45,50,50,48,41};
    v_type = {-1, -3};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig110." << endl;
}

void FuxTest::test_3v_1sp_fig111(){
    cout << "Start test_3v_1sp_fig111" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {69,72,72,74,74,70,69,67,65,77,76,77,
                    77,76,77,74,70,67,65,64,65,62,60,65};
    v_type = {1, 2};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig111." << endl;
}

void FuxTest::test_3v_1sp_fig112(){
    cout << "Start test_3v_1sp_fig112" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
```

```
    cp =           {69,70,72,69,70,72,69,67,72,69,70,69,
                    65,62,60,60,65,67,60,64,64,65,64,65};
    v_type = {1, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig112." << endl;
}

void FuxTest::test_3v_1sp_fig113(){
    cout << "Start test_3v_1sp_fig113" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {71,67,67,71,76,72,71,67,67,69,66,67,66,67,
                    55,52,52,52,48,48,55,52,48,45,47,43,50,43};
    v_type = {2, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig113." << endl;
}

void FuxTest::test_3v_1sp_fig114(){
    cout << "Start test_3v_1sp_fig114" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {71,67,67,71,67,69,71,71,72,64,66,67,66,67,
                    55,52,52,52,52,48,55,52,48,48,47,43,50,43};
    v_type = {3, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig114." << endl;
}

void FuxTest::test_3v_1sp_fig115(){
    cout << "Start test_3v_1sp_fig115" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {69,72,71,74,72,76,77,76,74,72,71,69};
    cp =           {60,64,67,65,64,67,69,67,65,69,68,69,
                    69,69,76,74,69,67,65,72,74,69,76,69};
    v_type = {-1, 0};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig115." << endl;
}

void FuxTest::test_3v_1sp_fig116(){
    cout << "Start test_3v_1sp_fig116" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
    cp =           {64,64,67,65,64,72,69,72,71,69,68,69,
                    45,45,52,50,57,57,50,48,55,57,52,45};
    v_type = {1, -1};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig116." << endl;
}

void FuxTest::test_3v_1sp_fig117(){
    cout << "Start test_3v_1sp_fig117" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {45,48,47,50,48,52,53,52,50,48,47,45};
    cp =           {60,64,62,65,64,60,60,60,62,57,56,57,
                    69,69,71,71,72,67,69,67,65,64,62,64};
    v_type = {2, 3};
    borrowMode = 1;
    test_configuration();
    cout << "End test_3v_1sp_fig117." << endl;
}

```

```
3598 void FuxTest::test_3v_1sp_fig118(){
3599     cout << "Start test test_3v_1sp_fig118" << endl;
3600     spList = {FIRST_SPECIES, FIRST_SPECIES};
3601     cantusFirmus = {60, 64, 65, 67, 64, 69, 67, 64, 65, 64, 62, 60};
3602     cp =           {67, 72, 69, 67, 72, 72, 76, 72, 71, 72, 71, 72,
3603                     48, 48, 50, 52, 48, 53, 52, 57, 50, 48, 55, 48};
3604     v_type = {1, -1};
3605     borrowMode = 1;
3606    test_configuration();
3607     cout << "End test_3v_1sp_fig118." << endl;
3608 }
3609
3610 void FuxTest::test_3v_1sp_fig119(){
3611     cout << "Start test_3v_1sp_fig119" << endl;
3612     spList = {FIRST_SPECIES, FIRST_SPECIES};
3613     cantusFirmus = {60, 64, 65, 67, 64, 69, 67, 64, 65, 64, 62, 60};
3614     cp =           {64, 67, 69, 71, 72, 72, 76, 72, 69, 72, 71, 72,
3615                     72, 72, 69, 67, 69, 65, 72, 72, 74, 72, 67, 72};
3616     v_type = {1, 1};
3617     borrowMode = 1;
3618     test_configuration();
3619     cout << "End test_3v_1sp_fig119." << endl;
3620 }
3621
3622
3623 void FuxTest::test_3v_2sp_fig125(){
3624     cout << "Start test_3v_2sp_fig125" << endl;
3625     spList = {FIRST_SPECIES, SECOND_SPECIES};
3626     cantusFirmus = {52,   48,   50,   48,   45,   57,   55,   52,   53,   52};
3627     cp =           {64,   64,   65,   64,   65,   64,   67,   67,   69,   68,
3628                     -1,71,72,71,69,65,67,64,69,71,72,69,71,74,76,74,72,69,71};
3629     v_type = {2 ,3};
3630     borrowMode = 1;
3631     test_configuration();
3632     cout << "End test_3v_2sp_fig125." << endl;
3633 }
3634
3635 void FuxTest::test_3v_2sp_fig128(){
3636     cout << "Start test_3v_2sp_fig128" << endl;
3637     spList = {SECOND_SPECIES, FIRST_SPECIES};
3638     cantusFirmus = {65,   67,   69,   65,   62,   64,   65,   72,   69,   65,
          ↪  67,   65};
3639     cp =
          ↪  {-1,77,76,74,72,71,69,72,70,69,67,72,69,81,79,76,72,69,74,77,77,76,77,
3640                   65,   60,   65,   65,   67,   72,   74,   76,   77,   74,
                        ↪  72,   65};
3641     v_type = {1 ,0};
3642     borrowMode = 1;
3643     test_configuration();
3644     cout << "End test_3v_2sp_fig128." << endl;
3645 }
3646
3647 void FuxTest::test_3v_2sp_fig129(){
3648     cout << "Start test_3v_2sp_fig129" << endl;
3649     spList = {FIRST_SPECIES, SECOND_SPECIES};
3650     cantusFirmus =  {65,   67,   69,   65,   62,   64,   65,   72,   69,   65,
          ↪  67,   65}; //1sp 2v cf
3651     cp =            {60,   60,   60,   62,   65,   67,   69,   67,   60,   62,
          ↪  64,   65,
3652                      -1,53,52,48,53,52,50,48,46,45,43,48,41,53,52,48,53,52,50,46,43,48,41};
                          ↪
3653     v_type = {-1, -3};
3654     borrowMode = 1;
3655     test_configuration();
3656     cout << "End test_3v_2sp_fig129." << endl;
3657 }
3658
3659 void FuxTest::test_3v_3sp_fig132(){
```

```
3660     cout << "Start test_3v_3sp_fig132" << endl;
3661     spList = {FIRST_SPECIES, THIRD_SPECIES};
3662     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62}; //1sp 2v cf
3663     cp =            {69,69,72,71,71,74,72,76,74,73,74,
3664                      -1,62,65,64,62,64,65,67,69,67,64,65,67,62,67,65,64,67,65,64,62,74,62,64,65,67,69,71,72,76,74,
3665     v_type = {1, 0};
3666     borrowMode = 1;
3667     test_configuration();
3668     cout << "End test_3v_3sp_fig132." << endl;
3669 }
3670
3671 void FuxTest::test_3v_3sp_fig133(){
3672     cout << "Start test_3v_3sp_fig133" << endl;
3673     spList = {THIRD_SPECIES, FIRST_SPECIES};
3674     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62}; //1sp 2v cf
3675     cp =
3676                      50,50,48,55,52,50,53,48,50,57,50};
3677     v_type = {0, -3};
3678     borrowMode = 1;
3679     test_configuration();
3680     cout << "End test_3v_3sp_fig133." << endl;
3681 }
3682
3683 void FuxTest::test_3v_4sp_fig150(){
3684     cout << "Start test_3v_4sp_fig150" << endl;
3685     spList = {FOURTH_SPECIES, FIRST_SPECIES};
3686     cantusFirmus =  {65,67,69,65,62,64,65,72,69,65,67,65};
3687     cp =            {69,71,72,69,65,67,69,67,72,69,70,69,
3688                      65,65,64,64,60,60,57,57,62,62,60,60,65,65,64,64,60,60,65,65,64,65};
3689     v_type = {0, -3};
3690     borrowMode = 1;
3691     test_configuration();
3692     cout << "End test_3v_4sp_fig150." << endl;
3693 }
3694
3695 void FuxTest::test_3v_4sp_fig151(){
3696     cout << "Start test_3v_4sp_fig151" << endl;
3697     spList = {FIRST_SPECIES, FOURTH_SPECIES};
3698     cantusFirmus =  {65,67,69,65,62,64,65,72,69,65,67,65}; //1sp 2v cf
3699     cp =            {53,60,65,62,58,55,62,64,65,62,58,57,
3700                      53,53,52,52,50,50,46,46,43,-1,48,48,46,46,45,45,50,50,53,53,52,53};
3701     v_type = {0, -3};
3702     borrowMode = 1;
3703     test_configuration();
3704     cout << "End test_3v_4sp_fig151." << endl;
3705 }
3706
3707 void FuxTest::test_4v_1sp_fig171(){
3708     cout << "Start test_4v_1sp_fig171" << endl;
3709     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
3710     cantusFirmus =  {65,67,69,65,62,64,65,72,69,65,67,65};
3711     cp =            {69,67,65,65,65,67,65,64,65,65,64,65,
3712                      60,60,60,62,62,58,57,57,53,57,60,57,
3713                      53,52,53,50,46,43,50,45,50,50,48,41};
3714     v_type = {3, 2, 3};
3715     borrowMode = 1;
3716     test_configuration();
3717     cout << "End test_test_4v_1sp_fig171." << endl;
3718 }
3719
3720 void FuxTest::test_4v_1sp_fig172(){
3721     cout << "Start test_4v_1sp_fig172" << endl;
3722     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
```

```
    cantusFirmus =  {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =            {72,71,72,69,69,72,72,72,72,69,70,65,
                     69,67,64,65,65,67,69,67,65,65,64,65,
                     65,62,60,60,62,60,60,64,60,62,58,60};
    v_type = {3, 2, 3};
    borrowMode = 1;
    test_configuration();
    cout << "End test test_4v_1sp_fig172." << endl;
}

void FuxTest::test_4v_2sp_fig175(){
    cout << "Start test_4v_2sp_fig175" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, SECOND_SPECIES};
    cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
    cp =            {74,74,72,74,76,77,77,76,74,73,74,
                     69,69,69,71,71,74,72,72,69,69,69,
                     -1,74,62,65,69,72,67,65,64,67,62,64,65,69,72,60,62,65,69,57,62};
    v_type = {3, 2, 3};
    borrowMode = 1;
    test_configuration();
    cout << "End test test_4v_2sp_fig175." << endl;
}


void FuxTest::test_4v_2sp_fig176(){
    cout << "Start test_4v_2sp_fig176" << endl;
    spList = {FIRST_SPECIES, SECOND_SPECIES, FIRST_SPECIES};
    cantusFirmus =  {50,   53,   52,   50,   55,   53,   57,   55,   53,   52,
      ↪  50};
    cp =            {65,   69,   72,   74,   70,   69,   69,   71,   74,   73,
      ↪  74,
                     -1,62,60,62,64,67,65,64,62,64,65,69,64,65,67,62,65,69,67,64,66,
                     69,   69,   67,   69,   70,   72,   72,   74,   69,   76,
                     ↪  69};
    v_type = {3, 2, 3};
    borrowMode = 1;
    test_configuration();
    cout << "End test test_4v_2sp_fig176." << endl;
}

void FuxTest::test_4v_3sp_fig184(){
    cout << "Start test_4v_3sp_fig184" << endl;
    spList = {FIRST_SPECIES, THIRD_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus =  {76,72,74,72,69,81,79,76,77,76};
    cp =
      ↪  {68,71,64,67,69,72,69,67,65,62,64,65,67,65,64,62,60,72,69,67,65,67,69,71,72,74,76,71,72,69,71,72,69,74,6
      ↪
                     59,57,57,55,57,60,64,60,62,59,
                     64,65,62,64,65,65,64,69,62,64};
    v_type = {3, 2, 3};
    borrowMode = 1;
    test_configuration();
    cout << "End test test_4v_3sp_fig184." << endl;
}

void FuxTest::test_4v_3sp_fig186(){
    cout << "Start test_4v_3sp_fig186" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES, THIRD_SPECIES};
    cantusFirmus =  {64,60,62,60,57,69,67,64,65,64};
    cp =            {71,76,77,76,77,77,76,76,69,68,
                     68,69,69,72,72,72,72,72,74,71,
                     76,74,72,71,69,67,65,64,62,64,65,62,69,72,69,67,65,67,69,67,65,67,69,71,72,64,65,67,69,67,65,
                     ↪
    v_type = {3, 2, 3};
    borrowMode = 1;
    test_configuration();
```

```
3783     cout << "End test test_4v_3sp_fig186." << endl;
3784 }
3785
3786 void FuxTest::test_4v_4sp_fig193(){
3787     cout << "Start test_4v_4sp_fig193" << endl;
3788     spList = {FOURTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
3789     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
3790     cp =            {69,69,74,74,72,72,71,71,74,74,69,69,77,77,76,76,74,74,73,74,
3791                      69,69,69,62,74,74,72,72,69,69,69,
3792                      50,50,45,47,43,50,53,48,50,45,50};
3793     v_type = {3, 2, 3};
3794     borrowMode = 1;
3795     test_configuration();
3796     cout << "End test test_4v_4sp_fig193." << endl;
3797 }
3798
3799 void FuxTest::test_4v_4sp_fig196(){
3800     cout << "Start test_4v_4sp_fig196" << endl;
3801     spList = {FOURTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
3802     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
3803     cp =            {57,57,62,62,60,60,59,59,62,62,57,57,65,65,64,64,62,62,61,62,
3804                      65,69,69,62,74,74,72,72,69,69,69,
3805                      50,50,45,47,43,50,53,48,50,45,50};
3806     v_type = {3, 2, 3};
3807     borrowMode = 1;
3808     test_configuration();
3809     cout << "End test test_4v_4sp_fig196." << endl;
3810 }
3811
3812
3813
3814 // ----- Util functions -----
3815
3816 void printVector(const std::vector<int>& array) {
3817     for (size_t i = 0; i < array.size(); ++i) {
3818         std::cout << array[i] << " ";
3819     }
3820     std::cout << std::endl;
3821 }
3822
3823 void printIntVarArray(const IntVarArray& array) {
3824     for (int i = 0; i < array.size(); ++i) {
3825         std::cout << array[i] << " ";
3826     }
3827     std::cout << std::endl;
3828 }
3829
3830 std::vector<CounterpointProblem*> get_all_solutions(CounterpointProblem* problem)
      ↪   {
3831     std::vector<CounterpointProblem*> solutions;
3832     BAB<CounterpointProblem> e(problem);
3833     while (CounterpointProblem* pb = e.next()) {
3834         solutions.push_back(pb);
3835     }
3836     return solutions;
3837 }
3838
3839 void FuxTest::test_configuration() {
3840     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
        ↪  general_params, specific_params, importance, borrowMode);
3841     for (size_t i = 0; i < cp.size(); i++) {
3842         int note = cp[i];
3843         if (note >  0) {
3844             rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, note)
          ↪  ;
3845         }
3846     }
3847     if (!has_solution(problem)) {
```

```
3848            std::cerr    << "/!\\ ERROR: It doesn't exist a solution but it should
                    ↪  with the following configuration:" << endl;
3849        } else {
3850            cout << "\t Test passed with the following configuration:" << endl;
3851        }
3852        cout << "\t Cantus firmus: ";
3853        printVector(cantusFirmus);
3854        cout << "\t Solution array: ";
3855        printIntVarArray(problem->getSolutionArray());
3856        delete problem;
3857    }
3858
3859    int FuxTest::get_motions(CounterpointProblem* problem, int i) {
3860        if (cp[i] > cp[i-1]) {
3861            if (problem->getSolutionArray()[i].val() > problem->getSolutionArray()[i
                    ↪  -1].val()) {
3862                return 2;
3863            } else if (problem->getSolutionArray()[i].val() < problem->
                    ↪  getSolutionArray()[i-1].val()) {
3864                return 0;
3865            } else {
3866                return 1;
3867            }
3868        }
3869        if (cp[i] < cp[i-1]) {
3870            if (problem->getSolutionArray()[i].val() > problem->getSolutionArray()[i
                    ↪  -1].val()) {
3871                return 0;
3872            } else if (problem->getSolutionArray()[i].val() < problem->
                    ↪  getSolutionArray()[i-1].val()) {
3873                return 2;
3874            } else {
3875                return 1;
3876            }
3877        }
3878        if (cp[i] == cp[i-1]) {
3879            if (problem->getSolutionArray()[i].val() == problem->getSolutionArray()[i
                    ↪  -1].val()) {
3880                return 2;
3881            } else if (problem->getSolutionArray()[i].val() < problem->
                    ↪  getSolutionArray()[i-1].val()) {
3882                return 0;
3883            } else {
3884                return 1;
3885            }
3886        }
3887        return -1;
3888    }
3889
3890    // ================================================
3891
3892    CounterpointProblem* FuxTest::dispatcher(char* test){
3893        if(strcmp(test, "1H1")==0){
3894            return test_1sp_H1();
3895        } else if(strcmp(test, "1H2")==0){
3896            return test_1sp_H2();
3897        } else if(strcmp(test, "1H32")==0){
3898            return test_1sp_H3();
3899        } else if(strcmp(test, "1H33")==0){
3900            return test_1sp_H3_2();
3901        } else if(strcmp(test, "1H41")==0){
3902            return test_1sp_H4_1();
3903        } else if(strcmp(test, "1H42")==0){
3904            return test_1sp_H4_2();
3905        } else if(strcmp(test, "1H5")==0){
3906            return test_1sp_H5();
3907        } else if(strcmp(test, "1H7")==0){
3908            return test_1sp_H7();
```

```
    } else if(strcmp(test, "1H72")==0){
        return test_1sp_H7_2();
    } else if(strcmp(test, "2H2")==0){
        return test_2sp_H2();
    } else if(strcmp(test, "3H1")==0){
        return test_3sp_H1();
    } else if(strcmp(test, "4H2")==0){
        return test_4sp_H2();
    } else {
        std::invalid_argument("Test for constraint not found!");
        return nullptr;
    }
}

FuxTest::FuxTest(int testNumber): FuxTest(testNumber, 0){
    idx = cp.size();
}

FuxTest::FuxTest(int testNumber, int i){
    if(testNumber==22){
        test_2v_1sp_fig22_setter(i);
    } else if(testNumber==23){
        test_2v_1sp_fig23_setter(i);
    } else if(testNumber==38){
        test_2v_2sp_fig38_setter(i);
    } else if(testNumber==39){
        test_2v_2sp_fig39_setter(i);
    } else if(testNumber==40){
        test_2v_2sp_fig40_setter(i);
    } else if(testNumber==55){
        test_2v_3sp_fig55_setter(i);
    } else if(testNumber==74){
        test_2v_4sp_fig74_setter(i);
    }else if(testNumber==82){
        test_2v_5sp_fig82_setter(i);
    }else if(testNumber==108){
        test_3v_1sp_fig108_setter(i);
    } else if(testNumber==109){
        test_3v_1sp_fig109_setter(i);
    } else if(testNumber==110){
        test_3v_1sp_fig110_setter(i);
    } else if(testNumber==111){
        test_3v_1sp_fig111_setter(i);
    } else if(testNumber==125){
        test_3v_2sp_fig125_setter(i);
    } else if(testNumber==146){
        test_3v_4sp_fig146_setter(i);
    } else if(testNumber==166){
        test_4v_1sp_fig166_setter(i);
    } else if(testNumber==167){
        test_4v_1sp_fig167_setter(i);
    } else if(testNumber==176){
        test_4v_2sp_fig176_setter(i);
    }
    else {
        throw std::invalid_argument("This test number has not been implemented (
          ↪ yet)");
    }
}

CounterpointProblem* FuxTest::test_1sp_H1(){
    spList = {FIRST_SPECIES};
    v_type = {2};
    auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
      ↪ general_params, specific_params,
            importance, borrowMode);
    rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 72); //does
      ↪ not work since it is not a consonant
```

```
3974     return problem;
3975 }
3976
3977 CounterpointProblem* FuxTest::test_1sp_H2(){
3978     spList = {FIRST_SPECIES};
3979     v_type = {2};
3980     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
       ↪  general_params, specific_params,
3981           importance, borrowMode);
3982     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 64);
3983     return problem;
3984 }
3985
3986 CounterpointProblem* FuxTest::test_1sp_H3(){
3987     spList = {FIRST_SPECIES};
3988     v_type = {2};
3989     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
       ↪  general_params, specific_params,
3990           importance, borrowMode);
3991     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-1],
       ↪  IRT_EQ, 64);
3992     return problem;
3993 }
3994
3995 CounterpointProblem* FuxTest::test_1sp_H3_2(){
3996     spList = {FIRST_SPECIES, FIRST_SPECIES};
3997     v_type = {2, 1};
3998     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
       ↪  general_params, specific_params,
3999           importance, borrowMode);
4000     rel(problem->getHome(), problem->getSolutionArray()[(problem->getSize()/2)
       ↪  -1], IRT_EQ, 67);
4001     rel(problem->getHome(), problem->getSolutionArray()[(problem->getSize())-1],
       ↪  IRT_EQ, 65);
4002     return problem;
4003 }
4004
4005 CounterpointProblem* FuxTest::test_1sp_H4_1(){
4006     spList = {FIRST_SPECIES};
4007     v_type = {1};
4008     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
       ↪  general_params, specific_params,
4009           importance, borrowMode);
4010     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 67);
4011     return problem;
4012 }
4013
4014 CounterpointProblem* FuxTest::test_1sp_H4_2(){
4015     spList = {FIRST_SPECIES};
4016     v_type = {1};
4017     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
       ↪  general_params, specific_params,
4018           importance, borrowMode);
4019     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-1],
       ↪  IRT_EQ, 67);
4020     return problem;
4021 }
4022
4023 CounterpointProblem* FuxTest::test_1sp_H5(){
4024     spList = {FIRST_SPECIES};
4025     v_type = {0};
4026     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
       ↪  general_params, specific_params,
4027           importance, borrowMode);
4028     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 62);
4029     return problem;
4030 }
4031
```

```
CounterpointProblem* FuxTest::test_1sp_H7(){
    spList = {FIRST_SPECIES};
    v_type = {0};
    auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
      ↪ general_params, specific_params,
            importance, borrowMode);
    rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-2],
      ↪ IRT_EQ, 69);
    return problem;
}

CounterpointProblem* FuxTest::test_1sp_H7_2(){
    spList = {FIRST_SPECIES};
    v_type = {-1};
    auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
      ↪ general_params, specific_params,
            importance, borrowMode);
    rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-2],
      ↪ IRT_EQ, 55);
    return problem;
}

CounterpointProblem* FuxTest::test_2sp_H2(){
    spList = {SECOND_SPECIES};
    v_type = {2};
    auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
      ↪ general_params, specific_params,
            importance, borrowMode);
    //these three lines are an example of the constraint allowing a dissonance
    //rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 78);
    //rel(problem->getHome(), problem->getSolutionArray()[3], IRT_EQ, 79);
    //rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 81);
    //these three lines are an example of the constraint blocking a dissonance
    rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 74);
    rel(problem->getHome(), problem->getSolutionArray()[3], IRT_EQ, 79);
    rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 81);
    return problem;
}

CounterpointProblem* FuxTest::test_3sp_H1(){
    spList = {THIRD_SPECIES};
    v_type = {2};
    auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
      ↪ general_params, specific_params,
            importance, borrowMode);
    rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 66);
    rel(problem->getHome(), problem->getSolutionArray()[5], IRT_EQ, 66);
    rel(problem->getHome(), problem->getSolutionArray()[6], IRT_EQ, 67);
    rel(problem->getHome(), problem->getSolutionArray()[7], IRT_EQ, 69);
    rel(problem->getHome(), problem->getSolutionArray()[8], IRT_EQ, 69);
    return problem;
}

CounterpointProblem* FuxTest::test_4sp_H2(){
    spList = {FOURTH_SPECIES};
    v_type = {-1};
    auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
      ↪ general_params, specific_params,
            importance, borrowMode);
    rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 52);
    rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 52);
    return problem;
}

vector<Species> FuxTest::getSpList(){
    return spList;
}

```

```
vector<int> FuxTest::getCf(){
    return cantusFirmus;
}

vector<int> FuxTest::getVType(){
    return v_type;
}

int FuxTest::getBMode(){
    return borrowMode;
}

vector<int> FuxTest::getCp(){
    return cp;
}

int FuxTest::getIdx(){
    return idx;
}

void FuxTest::test_2v_1sp_fig22_setter(int i){
    cout << "Fig. 22" << endl;
    spList = {FIRST_SPECIES};
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57}; //1sp 2v cf
    cp =           {69,64,67,65,64,72,69,71,71,69,68,69};
    v_type = {1};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_2v_1sp_fig23_setter(int i){
    cout << "Fig. 23" << endl;
    spList = {FIRST_SPECIES};
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57}; //1sp 2v cf
    cp =           {57,57,55,53,52,52,50,48,55,57,56,57};
    v_type = {-1};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_2v_2sp_fig38_setter(int i){
    cout << "Fig. 38" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53}; //1sp 2v cf
    cp =           {0,65,
                    64,62,
                    60,58,
                    57,55,
                    53,57,
                    60,58,
                    57,69,
                    67,64,
                    65,67,
                    69,65,
                    62,64,
                    65};
    v_type = {1};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_2v_2sp_fig39_setter(int i){
    cout << "Fig. 39 (the voice types as defined previously don't work here)" <<
      ↪ endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53}; //1sp 2v cf
    cp =
      ↪ {0,53,52,48,53,52,50,48,46,58,55,60,57,53,52,48,53,41,45,50,48,52,53};
```

```
4159     v_type = {-1};
4160     idx = i;
4161     borrowMode = 1;
4162 }
4163 
4164 void FuxTest::test_2v_2sp_fig40_setter(int i){
4165     cout << "Fig. 40" << endl;
4166     spList = {SECOND_SPECIES};
4167     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55}; //1sp 2v cf
4168     cp =
           ↪  {0,67,64,65,67,69,71,69,67,72,71,72,74,72,71,69,67,65,64,72,71,69,67,62,64,66,67};
           ↪
4169     v_type = {2};
4170     idx = i;
4171     borrowMode = 1;
4172 }
4173 
4174 void FuxTest::test_2v_3sp_fig55_setter(int i){
4175     cout << "Fig. 55" << endl;
4176     spList = {THIRD_SPECIES};
4177     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62}; //1sp 2v cf
4178     cp =           {62,64,65,67,//ok
4179                     69,71,72,74,//ok
4180                     76,74,71,72,//ok
4181                     74,72,70,69,//ok
4182                     70,72,74,76,//ok
4183                     77,65,69,71,//ok, but check for maybe bemol on last note
4184                     72,69,70,72,//ok
4185                     70,69,67,70,//HERE FIRST NOTE ERROR
4186                     69,62,64,65,
4187                     67,69,71,73,
4188                     74};
4189     cout << "CP size: " << endl;
4190     cout << cp.size() << endl;
4191     v_type = {1};
4192     idx = i;
4193     borrowMode = 1;
4194 }
4195 
4196 void FuxTest::test_2v_4sp_fig74_setter(int i){
4197     cout << "Fig. 74" << endl;
4198     spList = {FOURTH_SPECIES};
4199     cantusFirmus = {62,   65,   64,   62,   67,   65,   69,   67,   65,   64,
           ↪  62}; //1sp 2v cf
4200     cp =           {
           ↪  50,50,62,62,60,60,59,59,64,64,62,62,65,65,64,64,62,62,61,62};
4201     cout << cp.size() << endl;
4202     v_type = {-1};
4203     idx = i;
4204     borrowMode = 1;
4205 }
4206 
4207 void FuxTest::test_2v_5sp_fig82_setter(int i){
4208     cout << "Fig. 82" << endl;
4209     spList = {FIFTH_SPECIES};
4210     cantusFirmus = {62,   65,   64,   62,   67,   65,   69,   67,   65,   64,
           ↪  62}; //1sp 2v cf
4211     cp =           {0,0,69,69,69,62,64,65,67,65,64,67,65,62,
           ↪  74,74,74,72,70,67,69,71,72,72,72,72, 77,77,77,76,76,76,76,69,
           ↪  74,74,74,74,73,73,74};
4212     cout << cp.size() << endl;
4213     v_type = {1};
4214     idx = i;
4215     borrowMode = 1;
4216 }
4217 
4218 void FuxTest::test_3v_1sp_fig108_setter(int i){
4219     cout << "Fig. 108" << endl;
```

```
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {67,64,65,69,72,72,76,72,69,68,
                    52,57,50,53,53,53,60,60,62,64};
    v_type = {1,-2};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_3v_1sp_fig109_setter(int i){
    cout << "Fig. 109" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {52,48,50,48,45,57,55,52,53,52};
    cp =           {67,64,65,64,64,64,67,67,62,64,
                    59,60,57,57,60,60,59,59,57,59};
    v_type = {2,0};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_3v_1sp_fig110_setter(int i){
    cout << "Fig. 110" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {60,60,60,62,58,59,57,69,65,62,64,65,
                    53,52,53,50,55,55,53,45,50,50,48,41};
    v_type = {-1 ,-3};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_3v_1sp_fig111_setter(int i){
    cout << "Fig. 111" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {69,72,72,74,74,70,69,67,65,77,76,77,
                    65,64,65,62,58,55,53,52,53,50,48,53};
    v_type = {1 ,-2};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_3v_2sp_fig125_setter(int i){
    cout << "Fig. 125" << endl;
    spList = {FIRST_SPECIES, SECOND_SPECIES};
    cantusFirmus = {52,   48,   50,   48,   45,   57,   55,   52,   53,   52};
    cp =           {64,   64,   65,   64,   65,   64,   67,   67,   69,   68,
                    00,59,60,59,57,53,55,52,57,59,60,57,59,62,64,62,60,57,59};
    v_type = {2 ,1};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_3v_4sp_fig146_setter(int i){
    cout << "Fig. 146" << endl;
    spList = {FOURTH_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,   60,   62,   60,   57,   69,   67,   64,   65,   64};
    cp =           {   76,76,72,72,71,71,69,69,72,72,74,74,72,72,69,69,71,68,
                    52,   57,   55,   57,   53,   53,   52,   48,   50,   52};
    v_type = {1 ,-2};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_4v_1sp_fig166_setter(int i){
    cout << "Fig. 166" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
```

```
    cp =           {59,57,57,57,60,60,64,60,62,59,
                    56,57,53,52,52,53,60,60,57,56,
                    52,53,50,45,45,41,40,45,50,52};
    v_type = {-1 ,-1, -3};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_4v_1sp_fig167_setter(int i){
    cout << "Fig. 167" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {68,69,65,64,64,65,72,72,69,68,
                    59,57,57,57,60,60,64,60,62,59,
                    52,53,50,45,45,53,52,57,50,52};
    v_type = {0 ,-1, -3};
    idx = i;
    borrowMode = 1;
}

void FuxTest::test_4v_2sp_fig176_setter(int i){
    cout << "Fig. 173" << endl;
    spList = {SECOND_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {50,   53,   52,   50,   55,   53,   57,   55,   53,   52,
      ↪  50};
    cp =
      ↪  {00,62,60,62,64,67,65,64,62,64,65,69,64,65,67,62,65,69,67,64,66,
                    65,   69,   72,   74,   70,   69,   69,   71,   74,   73,
                      ↪  74,
                    57,   57,   55,   57,   58,   60,   60,   62,   57,   64,
                      ↪  57};
    v_type = {2, 3, 1};
    idx = i;
    borrowMode = 1;
}
```

figureTests.cpp

```
    #include "../headers/figureTests.hpp"
#include <iostream>
#include <queue>

using namespace std;

FigureTests::FigureTests() {
    borrowMode = 1;
    melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
    general_params = {4, 1, 1, 2, 2, 2, 8, 1};
    specific_params = {8, 4, 0, 2, 1, 8, 50};
    importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
    notesSpeciesFor5sp = {};
}

CounterpointProblem* FigureTests::set_configuration() {
    // create a new problem
    auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
      ↪  general_params, specific_params, importance, borrowMode);
    // set the solution array
    for (size_t i = 0; i < cp.size(); i++) {
        int note = cp[i];
        if (note >  0) {
            rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, note)
              ↪  ;
        }
    }
```

```
    // set the species array for the 5th sepcies counterpoint
    if (spList[0] == FIFTH_SPECIES) {
        auto* part = problem->getCounterpoint_1();
        for (int i = 0; i < notesSpeciesFor5sp.size(); i++) {
            int sp = notesSpeciesFor5sp[i];
            if (sp > -1) {
                rel(problem->getHome(), part->getSpeciesArray()[i], IRT_EQ, sp);
            }

        }
    }
    if (spList.size() > 1 && spList[1] == FIFTH_SPECIES) {
        auto* part = problem->getCounterpoint_2();
        for (int i = 0; i < notesSpeciesFor5sp.size(); i++) {
            int sp = notesSpeciesFor5sp[i];
            if (sp > -1) {
                rel(problem->getHome(), part->getSpeciesArray()[i], IRT_EQ, sp);
            }
        }
    }
    if (spList.size() > 2 && spList[2] == FIFTH_SPECIES) {
        auto* part = problem->getCounterpoint_3();
        for (int i = 0; i < notesSpeciesFor5sp.size(); i++) {
            int sp = notesSpeciesFor5sp[i];
            if (sp > -1) {
                rel(problem->getHome(), part->getSpeciesArray()[i], IRT_EQ, sp);
            }
        }
    }
    return problem;
}

std::vector<CounterpointProblem*> FigureTests::get_all_solutions() {
    CounterpointProblem* problem = set_configuration();
    std::vector<CounterpointProblem*> solutions;
    BAB<CounterpointProblem> e(problem);
    while (CounterpointProblem* pb = e.next()) {
        cout << "Solution found" << endl;
        solutions.push_back(pb);
        cout << pb->getSolutionArray() << endl;
        cout << pb->getCounterpoint_1()-> getSpeciesArray() << endl;
        cout << "Solution added" << endl;
    }
    return solutions;
}

/*
* Check if the problem is unsatisfiable
* @return true if the problem is unsatisfiable, false otherwise
*/
bool FigureTests::is_unsat(const set<int>& cons_set) {
    // set active cons_set
    activeConstraints = std::vector<bool>(activeConstraints.size(), false);
    for (int cons : cons_set) {
        activeConstraints[cons] = true;
    }
    // create a new problem and set the solution array
    CounterpointProblem* problem = set_configuration();
    // check if the problem is unsatisfiable
    bool unsat = !has_solution(problem);
    delete problem;
    return unsat;
}

/**
 * Minimize the set of constraints by removing one constraint at a time
 * until the problem becomes satisfiable.
 * @param cons_set the set of constraints to minimize
```

```
 * @return the minimized set of constraints
 */
std::set<int> FigureTests::minimize(const std::set<int>& cons_set) {
    std::set<int> mus = cons_set;
    for (auto it = mus.begin(); it != mus.end(); ) {
        set<int> trial = mus;
        trial.erase(*it);
        if (is_unsat(trial)) {
            mus = trial;
            it = mus.begin(); // restart
        } else {
            it++;
        }
    }
    return mus;
}

/**
 * Find all minimal unsatisfiable subsets (MUSes) of the active constraints
 * using a breadth-first search algorithm.
 */
void FigureTests::findAllMUSes() {
    // Initialization
    set<int> cons_set; // all constraint indices
    for (int i = 0; i < activeConstraints.size(); i++) {
        cons_set.insert(i);
    }

    set<set<int>> mus_found;
    set<set<int>> visited;
    queue<set<int>> queue;

    queue.push(cons_set);
    visited.insert(cons_set);

    while (!queue.empty()) {
        std::set<int> current = queue.front();
        queue.pop();
        // Check if the current set is a minimal unsatisfiable subset
        if (!is_unsat(current))  {
            continue;
        }
        std::set<int> mus = minimize(current);
        if (!mus_found.count(mus)) {
            mus_found.insert(mus);
            // Remove one constraint at a time and add the new set to the queue
            ↪ to explore further
            for (int c : mus) {
                set<int> next = current;
                next.erase(c);
                if (!visited.count(next)) {
                    visited.insert(next);
                    queue.push(next);
                }
            }
        }
    }
    // Print the MUSes found
    for (set<int> mus : mus_found) {
        cout << "\t MUS found: ";
        for (int cons : mus) {
            cout << get_constraint_name(cons) << " ";
        }
        cout << endl;
    }
}

void FigureTests::find_unsat_constraints() {
```

```
    // activeConstraints = std::vector<bool>(activeConstraints.size(), false);
    // cout << has_solution(set_configuration()) << endl;
    for (int i = 0; i < consSize; i++) {
        set<int> cons_set; // all constraint indices
        cons_set.insert(i);
        if (is_unsat(cons_set))  {
            cout << "\t" << get_constraint_name(i) << "␣is␣unsatisfiable." <<
              ↪  endl;
        }
    }
}

/* ============================================================================
                            FIGURES
============================================================================ */

void FigureTests::test_2v_1sp_fig22() {
    cout << "Start␣test_2v_1sp_fig22" << endl;
    spList = {FIRST_SPECIES};
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
    cp =           {69,64,67,65,64,72,69,71,71,69,68,69};
    v_type = {1};
    findAllMUSes();
}

void FigureTests::test_2v_1sp_fig23() {
    cout << "Start␣test_2v_1sp_fig23" << endl;
    spList = {FIRST_SPECIES};
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
    cp =           {57,57,55,53,52,52,50,48,55,57,56,57};
    v_type = {-1};
    findAllMUSes();
}

void FigureTests::test_2v_2sp_fig38() {
    cout << "Start␣test_2v_2sp_fig38" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =
      ↪  {-1,65,64,62,60,58,57,55,53,57,60,58,57,69,67,64,65,67,69,65,62,64,65};
      ↪
    v_type = {2};
    findAllMUSes();
}

void FigureTests::test_2v_2sp_fig39() {
    cout << "Start␣test_2v_2sp_fig39" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =
      ↪  {-1,53,52,48,53,52,50,48,46,58,55,60,57,53,52,48,53,41,45,50,48,52,53};
      ↪
    v_type = {0};
    findAllMUSes();
}

void FigureTests::test_2v_2sp_fig40() {
    cout << "Start␣test_2v_2sp_fig40" << endl;
    spList = {SECOND_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {-1,
      ↪  67,64,65,67,69,71,69,67,72,71,72,74,72,71,69,67,65,64,72,71,69,67,62,64,66,67};
      ↪
    v_type = {2};
    findAllMUSes();
}

void FigureTests::test_2v_2sp_fig41() {
```

```
222     cout << "Start test_2v_2sp_fig41" << endl;
223     spList = {SECOND_SPECIES};
224     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
225     cp =
          ↪ {-1,55,52,53,55,53,52,50,48,52,48,60,59,57,55,59,60,59,57,55,54,50,55,47,50,54,55};
          ↪
226     v_type = {0};
227     findAllMUSes();
228 }
229
230 void FigureTests::test_2v_2sp_fig42() {
231     cout << "Start test_2v_2sp_fig42" << endl;
232     spList = {SECOND_SPECIES};
233     cantusFirmus = {57,60,59,60,64,65,64,62,60,59,57};
234     cp =
          ↪ {-1,69,64,65,67,62,64,76,72,71,69,65,67,71,74,69,72,64,66,68,69};
235     v_type = {2};
236     findAllMUSes();
237 }
238
239 void FigureTests::test_2v_2sp_fig43() {
240     cout << "Start test_2v_2sp_fig43" << endl;
241     spList = {SECOND_SPECIES};
242     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
243     cp =
          ↪ {-1,45,57,52,55,52,50,53,57,59,60,48,50,45,48,52,53,55,57,45,52,56,57};
          ↪
244     v_type = {-1};
245     findAllMUSes();
246 }
247
248 void FigureTests::test_2v_2sp_fig44() {
249     cout << "Start test_2v_2sp_fig44" << endl;
250     spList = {SECOND_SPECIES};
251     cantusFirmus = {60,64,65,67,64,69,67,64,65,64,62,60};
252     cp =
          ↪ {-1,67,72,71,69,74,71,69,67,71,72,74,76,74,72,71,69,71,72,67,69,71,72};
          ↪
253     v_type = {1};
254     findAllMUSes();
255 }
256
257 void FigureTests::test_2v_2sp_fig45() {
258     cout << "Start test_2v_2sp_fig45" << endl;
259     spList = {SECOND_SPECIES};
260     cantusFirmus = {60,64,65,67,64,69,67,64,65,64,62,60};
261     cp =
          ↪ {-1,48,60,59,57,62,59,55,60,59,57,60,64,62,60,57,62,57,60,48,55,59,60};
          ↪
262     v_type = {-1};
263     findAllMUSes();
264 }
265
266 void FigureTests::test_2v_3sp_fig55() {
267     cout << "Start test_2v_3sp_fig55" << endl;
268     spList = {THIRD_SPECIES};
269     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
270     cp =
          ↪ {62,64,65,67,69,71,72,74,76,74,71,72,74,72,70,69,70,72,74,76,77,65,69,71,72,69,70,72,70,69,67,71,69,62,6
          ↪
271     v_type = {2};
272     findAllMUSes();
273 }
274
275 void FigureTests::test_2v_3sp_fig56() {
276     cout << "Start test_2v_3sp_fig56" << endl;
277     spList = {THIRD_SPECIES};
278     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
```

```
    cp =
      ↪ {50,52,53,55,57,50,57,59,60,59,55,57,59,57,55,53,52,64,59,60,62,57,50,52,53,55,57,59,60,62,64,60,62,57,5
      ↪
    v_type = {-1};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig57() {
    cout << "Start test_2v_3sp_fig57" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =
      ↪ {71,67,69,71,72,71,69,67,65,67,69,71,72,64,65,67,69,72,76,74,72,71,69,72,71,74,71,69,67,71,72,71,69,71,7
      ↪
    v_type = {2};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig58() {
    cout << "Start test_2v_3sp_fig58" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =
      ↪ {52,53,55,52,57,55,53,52,50,52,53,55,57,52,57,55,53,52,50,52,53,55,57,59,60,62,64,62,60,48,60,59,57,62,5
      ↪
    v_type = {-1};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig59() {
    cout << "Start test_2v_3sp_fig59" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =
      ↪ {65,64,62,60,59,62,67,65,64,62,60,58,57,60,62,64,65,62,64,65,67,64,65,67,69,67,65,69,67,65,64,62,60,64,6
      ↪
    v_type = {1};
    findAllMUSes();
}

void FigureTests::test_2v_3sp_fig60() {
    cout << "Start test_2v_3sp_fig60" << endl;
    spList = {THIRD_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =
      ↪ {41,53,52,50,52,48,50,52,53,52,50,48,50,48,46,45,46,53,58,57,55,48,60,58,57,55,53,50,52,48,50,52,53,52,5
      ↪
    v_type = {0};
    findAllMUSes();
}

void FigureTests::test_2v_4sp_fig74() {
    cout << "Start test_2v_4sp_fig74" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {50,50,62,62,60,60,59,59,64,64,62,62,65,65,64,64,62,62,61,62};
    v_type = {-1};
    findAllMUSes();
}

void FigureTests::test_2v_4sp_fig75() {
    cout << "Start test_2v_4sp_fig75" << endl;
    spList = {FOURTH_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {76,76,72,72,71,72,64,64,65,65,72,72,71,71,76,76,74,76};
    v_type = {2};
    findAllMUSes();
}
```

```
337
338 void FigureTests::test_2v_4sp_fig76() {
339     cout << "Start test_2v_4sp_fig76" << endl;
340     spList = {FOURTH_SPECIES};
341     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
342     cp =           {52,52,57,57,55,55,53,53,50,50,62,62,60,60,64,64,62,64};
343     v_type = {-2};
344     findAllMUSes();
345 }
346
347 void FigureTests::test_2v_4sp_fig77() {
348     cout << "Start test_2v_4sp_fig77" << endl;
349     spList = {FOURTH_SPECIES};
350     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
351     cp =
        ↪  {65,65,64,64,60,60,65,65,69,69,67,67,65,65,64,64,69,69,65,65,64,65};
352     v_type = {2};
353     findAllMUSes();
354 }
355
356 void FigureTests::test_2v_4sp_fig78() {
357     cout << "Start test_2v_4sp_fig78" << endl;
358     spList = {FOURTH_SPECIES};
359     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
360     cp =
        ↪  {53,53,52,52,50,50,46,46,43,43,55,55,53,53,52,52,50,50,53,53,52,53};
361     v_type = {-1};
362     findAllMUSes();
363 }
364
365 void FigureTests::test_2v_5sp_fig82() {
366     cout << "Start test_2v_5sp_fig82" << endl;
367     spList = {FIFTH_SPECIES};
368     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
369     cp =           {-1,-1,69,-1,
370                     69,-1, 64,65,
371                     67,65,64,67,
372                     65,62,74,-1,
373                     74,-1,70,67,
374                     69,71,72,-1,
375                     72,-1,77,-1,
376                     77,-1,76,-1,
377                     76,-1,74,-1,
378                     74,-1,73,-1,
379                     74};
380     notesSpeciesFor5sp = {
381                     -1, -1, FOURTH_SPECIES, -1,
382                     FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
383                     THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES,
384                     THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
385                     FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
386                     THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
387                     FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
388                     FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
389                     FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
390                     FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
391                     FOURTH_SPECIES
392                 };
393     v_type = {2};
394     findAllMUSes();
395 }
396
397 void FigureTests::test_2v_5sp_fig83() {
398     cout << "Start test_2v_5sp_fig83" << endl;
399     spList = {FIFTH_SPECIES};
400     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
401     cp =           {-1,-1,62,-1,
402                     62,-1,57,59,
```

```
                    60,55,60,-1,
                    60,-1,59,57,
                    55,57,59,60,
                    62,57,62,-1,
                    62,-1,65,-1,
                    65,-1,64,-1,
                    64,-1,62,-1,
                    62,-1,61,-1,
                    62};
    notesSpeciesFor5sp = {
                    -1, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES
    };
    v_type = {0};
    findAllMUSes();
}

void FigureTests::test_2v_5sp_fig87_1() {
    cout << "Start test_2v_5sp_fig87_1" << endl;
    spList = {FIFTH_SPECIES};
    cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
    cp =           {-1,-1,69,-1,
                    69,-1,67,69,
                    71,67,71,-1,
                    71,-1,69,71,
                    72,67,72,-1,
                    72,-1,71,72,
                    74,69,74,-1,
                    74,-1,72,-1,
                    72,-1,71,-1,
                    71,-1,69,-1,
                    69,-1,68,-1,
                    69};
    notesSpeciesFor5sp = {
                    -1, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                    THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                    FOURTH_SPECIES
                };
    v_type = {2};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig108(){
    cout << "Start test_3v_1sp_fig108" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {67,64,65,69,72,72,76,72,69,68,
                    52,57,50,53,53,53,60,60,62,64};
    v_type = {1, -2};
    findAllMUSes();
```

```
}

void FigureTests::test_3v_1sp_fig109(){
    cout << "Start test_3v_1sp_fig109" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {52,48,50,48,45,57,55,52,53,52};
    cp =           {67,64,65,64,64,64,67,67,62,64,
                    59,60,57,57,60,60,59,59,57,59};
    v_type = {2, 1};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig110(){
    cout << "Start test_3v_1sp_fig110" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {60,60,60,62,58,59,57,69,65,62,64,65,
                    53,52,53,50,55,55,53,45,50,50,48,41};
    v_type = {-1, -3};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig111(){
    cout << "Start test_3v_1sp_fig111" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {69,72,72,74,74,70,69,67,65,77,76,77,
                    65,64,65,62,58,55,53,52,53,50,48,53};
    v_type = {1, -2};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig112(){
    cout << "Start test_3v_1sp_fig112" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =           {69,70,72,69,70,72,69,67,72,69,70,69,
                    65,62,60,60,65,67,60,64,64,65,64,65};
    v_type = {3, 2};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig113(){
    cout << "Start test_3v_1sp_fig113" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {59,55,55,59,64,60,59,55,55,57,54,55,54,55,
                    55,52,52,52,48,48,55,52,48,45,47,43,50,43};
    v_type = {0, -1};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig114(){
    cout << "Start test_3v_1sp_fig114" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55};
    cp =           {71,67,67,71,67,69,71,71,72,64,66,67,66,67,
                    55,52,52,52,52,48,55,52,48,48,47,43,50,43};
    v_type = {2, -1};
    findAllMUSes();
}

void FigureTests::test_3v_1sp_fig115(){
    cout << "Start test_3v_1sp_fig115" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {69,72,71,74,72,76,77,76,74,72,71,69};
    cp =           {60,64,67,65,64,67,69,67,65,69,68,69,
                    57, 57, 64, 62, 57, 55, 53, 60, 62, 57, 64, 57};
```

```
539     v_type = {-1, -2};
540     findAllMUSes();
541 }
542 
543 void FigureTests::test_3v_1sp_fig116(){
544     cout << "Start test_3v_1sp_fig116" << endl;
545     spList = {FIRST_SPECIES, FIRST_SPECIES};
546     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57};
547     cp =           {64,64,67,65,64,72,69,72,71,69,68,69,
548                     45,45,52,50,57,57,50,48,55,57,52,45};
549     v_type = {1, -1};
550     findAllMUSes();
551 }
552 
553 void FigureTests::test_3v_1sp_fig117(){
554     cout << "Start test_3v_1sp_fig117" << endl;
555     spList = {FIRST_SPECIES, FIRST_SPECIES};
556     cantusFirmus = {45,48,47,50,48,52,53,52,50,48,47,45};
557     cp =           {60,64,62,65,64,60,60,60,62,57,56,57,
558                     57,57,59,59,60,55,57,55,53,52,50,52};
559     v_type = {2, 1};
560     findAllMUSes();
561 }
562 
563 void FigureTests::test_3v_1sp_fig118(){
564     cout << "Start test test_3v_1sp_fig118" << endl;
565     spList = {FIRST_SPECIES, FIRST_SPECIES};
566     cantusFirmus = {60, 64, 65, 67, 64, 69, 67, 64, 65, 64, 62, 60};
567     cp =           {55, 60, 57, 55, 60, 60, 64, 60, 59, 60, 59, 60,
568                     48, 48, 50, 52, 48, 53, 52, 57, 50, 48, 55, 48};
569     v_type = {-1, -1};
570    findAllMUSes();
571 }
572 
573 void FigureTests::test_3v_1sp_fig119(){
574     cout << "Start test_3v_1sp_fig119" << endl;
575     spList = {FIRST_SPECIES, FIRST_SPECIES};
576     cantusFirmus = {60, 64, 65, 67, 64, 69, 67, 64, 65, 64, 62, 60};
577     cp =           {64, 67, 69, 71, 72, 72, 76, 72, 69, 72, 71, 72,
578                     60, 60, 57, 55, 57, 53, 60, 60, 62, 60, 55, 60};
579     v_type = {1, -1};
580     findAllMUSes();
581 }
582 
583 void FigureTests::test_3v_2sp_fig125(){
584     cout << "Start test_3v_2sp_fig125" << endl;
585     spList = {FIRST_SPECIES, SECOND_SPECIES};
586     cantusFirmus = {52,   48,   50,   48,   45,   57,   55,   52,   53,   52};
587     cp =           {64,   64,   65,   64,   65,   64,   67,   67,   69,   68,
588                     -1,59,60,59,57,53,55,52,57,59,60,57,59,62,64,62,60,57,59};
589     v_type = {2 ,1};
590     findAllMUSes();
591 }
592 
593 void FigureTests::test_3v_2sp_fig126(){
594     cout << "Start test_3v_2sp_fig126" << endl;
595     spList = {FIRST_SPECIES, SECOND_SPECIES};
596     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
597     cp =           {52,57,53,57,53,65,64,60,57,56,
598                     -1,52,53,52,50,52,53,48,50,52,53,57,60,55,57,52,52,50,52};
599     v_type = {-1, -2};
600     findAllMUSes();
601 }
602 
603 void FigureTests::test_3v_2sp_fig127(){
604     cout << "Start test_3v_2sp_fig127" << endl;
605     spList = {SECOND_SPECIES, FIRST_SPECIES};
606     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
```

```
    cp =
      ↪ {-1,65,64,62,60,57,62,57,58,57,55,60,57,69,67,64,60,57,62,65,65,64,65,
                    53,48,53,50,46,48,50,52,53,50,48,41};
    v_type = {-1, -3};
    findAllMUSes();
}



void FigureTests::test_3v_2sp_fig128(){
    cout << "Start test_3v_2sp_fig128" << endl;
    spList = {SECOND_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,   67,   69,   65,   62,   64,   65,   72,   69,   65,
      ↪ 67,   65};
    cp =
      ↪ {-1,77,76,74,72,71,69,72,70,69,67,72,69,81,79,76,72,69,74,77,77,76,77,
                    53,   48,   53,   53,   55,   60,   62,   64,   65,   62,
                      ↪ 60,   53};
    v_type = {1 ,-2};
    findAllMUSes();
}

void FigureTests::test_3v_2sp_fig129(){
    cout << "Start test_3v_2sp_fig129" << endl;
    spList = {FIRST_SPECIES, SECOND_SPECIES};
    cantusFirmus =  {53, 55, 57, 53, 50, 52, 53, 60, 57, 53, 55, 53};
    cp =            {60,   60,   60,   62,   65,   67,   69,   67,   60,   62,
      ↪ 64,   65,
                     -1,53,52,48,53,52,50,48,46,45,43,48,41,53,52,48,53,52,50,46,43,48,41};
                       ↪
    v_type = {1, -1};
    findAllMUSes();
}

void FigureTests::test_3v_3sp_fig130(){
    cout << "Start test_3v_3sp_fig130" << endl;
    spList = {THIRD_SPECIES, FIRST_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =
      ↪ {69,62,65,67,69,65,69,71,72,71,67,69,71,72,74,71,76,74,71,73,74,77,76,74,72,69,72,74,76,74,72,71,69,62,6
      ↪
                    50,50,48,55,52,50,53,48,50,57,50};
    v_type = {1, -2};
    findAllMUSes();
}

void FigureTests::test_3v_3sp_fig131(){
    cout << "Start test_3v_3sp_fig131" << endl;
    spList = {THIRD_SPECIES, FIRST_SPECIES};
    cantusFirmus = {50,53,52,50,55,53,57,55,53,52,50};
    cp =
      ↪ {62,64,65,67,69,57,60,62,64,65,67,64,65,69,65,64,62,58,62,64,65,69,67,65,64,60,64,65,67,64,65,67,69,57,6
      ↪
                    57,57,61,62,58,57,60,58,57,55,54};
    v_type = {2, 1};
    findAllMUSes();
}

void FigureTests::test_3v_3sp_fig132(){
    cout << "Start test_3v_3sp_fig132" << endl;
    spList = {FIRST_SPECIES, THIRD_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {69,69,72,71,71,74,72,76,74,73,74,
                    -1,50,53,52,50,52,53,55,57,55,52,53,55,50,55,53,52,55,53,52,50,62,50,52,53,55,57,59,60,64,62,6
                      ↪
    v_type = {1, -1};
    findAllMUSes();
}
```

```
664
665 void FigureTests::test_3v_3sp_fig133(){
666     cout << "Start test_3v_3sp_fig133" << endl;
667     spList = {THIRD_SPECIES, FIRST_SPECIES};
668     cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
669     cp =
        ↪  {53,50,53,55,57,53,57,59,60,52,55,57,59,62,59,57,55,57,59,61,62,64,65,62,60,57,60,62,64,62,60,59,57,62,5
        ↪
670                     50,50,48,55,52,50,53,48,50,57,50};
671     v_type = {-1, -2};
672     findAllMUSes();
673 }
674
675 void FigureTests::test_3v_4sp_fig146() {
676     cout << "Start test_3v_4sp_fig146" << endl;
677     spList = {FOURTH_SPECIES, FIRST_SPECIES};
678     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
679     cp =           {76,76,72,72,71,71,69,69,72,72,74,74,72,72,69,69,71,68,
680                     52,57,55,57,53,53,52,48,50,52};
681     v_type = {2,-2};
682     findAllMUSes();
683 }
684
685 void FigureTests::test_3v_4sp_fig147() {
686     cout << "Start test_3v_4sp_fig147" << endl;
687     spList = {FOURTH_SPECIES, FIRST_SPECIES};
688     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
689     cp =           {64,64,60,60,59,59,57,57,62,62,60,60,59,59,57,57,59,56,
690                     52,57,55,57,53,53,52,48,50,52};
691     v_type = {0,-2};
692     findAllMUSes();
693 }
694
695 void FigureTests::test_3v_4sp_fig148() {
696     cout << "Start test_3v_4sp_fig148" << endl;
697     spList = {FIRST_SPECIES, FOURTH_SPECIES};
698     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
699     cp =           {67,67,67,64,72,74,76,72,69,68,
700                     52,52,48,48,47,47,45,45,53,53,50,50,48,48,52,52,50,52};
701     v_type = {2,-2};
702     findAllMUSes();
703 }
704
705 void FigureTests::test_3v_4sp_fig149() {
706     cout << "Start test_3v_4sp_fig149" << endl;
707     spList = {FOURTH_SPECIES, FIRST_SPECIES};
708     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
709     cp =
        ↪  {65,65,64,64,62,62,65,65,67,67,69,69,65,65,64,64,62,62,65,65,64,65,
710                     53,48,53,50,46,48,50,57,53,50,48,41};
711     v_type = {2,0};
712     findAllMUSes();
713 }
714
715 void FigureTests::test_3v_4sp_fig150(){
716     cout << "Start test_3v_4sp_fig150" << endl;
717     spList = {FIRST_SPECIES, FOURTH_SPECIES};
718     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
719     cp =           {69,71,72,69,65,67,69,67,72,69,70,69,
720                     65,65,64,64,60,60,57,57,62,62,60,60,65,65,64,64,60,60,65,65,64,65};
                        ↪
721     v_type = {3, 2};
722     findAllMUSes();
723 }
724
725 void FigureTests::test_3v_4sp_fig151(){
726     cout << "Start test_3v_4sp_fig151" << endl;
727     spList = {FIRST_SPECIES, FOURTH_SPECIES};
```

```
728     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
729     cp =           {53,60,65,62,58,55,62,64,65,62,58,57,
730                     53,53,52,52,50,50,46,46,43,-1,48,48,46,46,45,45,50,50,53,53,52,53};
                          ↪
731     v_type = {-2, -2};
732     findAllMUSes();
733 }
734
735 void FigureTests::test_3v_5sp_fig154() {
736     cout << "Start_test_3v_5sp_fig154" << endl;
737     spList = {FIFTH_SPECIES, FIRST_SPECIES};
738     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
739     cp =           {-1,-1,69,-1,
740                     69,-1,69,71,
741                     72,67,72,-1,
742                     72,-1,71,-1,
743                     71,-1,71,73,
744                     74,77,76,74,
745                     72,65,77,-1,
746                     77,-1,76,-1,
747                     76,-1,74,-1,
748                     74,-1,73,-1,
749                     74,
750
751                     50,62,60,55,52,50,53,60,62,57,50};
752     notesSpeciesFor5sp = {
753                          -1, -1, FOURTH_SPECIES, -1,
754                          FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
755                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
756                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
757                          FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
758                          THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES,
                               ↪  THIRD_SPECIES,
759                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
760                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
761                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
762                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
763                          FOURTH_SPECIES,
764     };
765     v_type = {2, -2};
766     findAllMUSes();
767 }
768
769 void FigureTests::test_3v_5sp_fig155() {
770     cout << "Start_test_3v_5sp_fig155" << endl;
771     spList = {FIFTH_SPECIES, FIRST_SPECIES};
772     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
773     cp =           {-1,-1,57,-1,
774                     57,-1,57,59,
775                     60,55,60,-1,
776                     60,-1,59,-1,
777                     59,-1,59,61,
778                     62,57,62,-1,
779                     62,-1,65,-1,
780                     65,-1,64,-1,
781                     64,-1,62,-1,
782                     62,-1,61,-1,
783                     62,
784
785                     50,50,48,55,52,50,53,48,50,57,50};
786     notesSpeciesFor5sp = {
787                          -1, -1, FOURTH_SPECIES, -1,
788                          FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
789                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
790                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
791                          FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
792                          THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
793                          FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
```

```
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,
    };
    v_type = {0, -2};
    findAllMUSes();
}

void FigureTests::test_3v_5sp_fig156() {
    cout << "Start test_3v_5sp_fig156" << endl;
    spList = {FIRST_SPECIES, FIFTH_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {69,69,72,71,71,74,72,72,69,67,66,

                    -1,-1,50,-1,
                    50,-1,53,55,
                    57,52,57,-1,
                    57,-1,55,53,
                    52,55,53,52,
                    50,52,53,-1,
                    53,-1,53,-1,
                    53,-1,52,-1,
                    52,-1,50,-1,
                    50,-1,49,-1,
                    50};
    notesSpeciesFor5sp = {
                        -1, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, THIRD_SPECIES,
                          ↪ THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES
    };
    v_type = {2,-2};
    findAllMUSes();
}

void FigureTests::test_3v_5sp_fig157() {
    cout << "Start test_3v_5sp_fig157" << endl;
    spList = {FIFTH_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {-1,-1,76,-1,
                    76,-1,72,71,
                    -1,-1,67,65,
                    -1,-1,69,-1,
                    69,-1,72,-1,
                    72,-1,77,-1,
                    77,-1,76,74,
                    72,67,72,-1,
                    72,-1,71,69,
                    68,

                    52,57,53,57,53,53,60,60,62,64};
    notesSpeciesFor5sp = {
                        -1, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
                        -1,  -1, THIRD_SPECIES, THIRD_SPECIES,
                        -1, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
```

```
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                        FOURTH_SPECIES
    };
    v_type = {2,0};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig166() {
    cout << "Start test_4v_1sp_fig166" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {59,57,57,57,60,60,64,60,62,59,
                    56,57,53,52,52,53,60,60,57,56,
                    52,53,50,45,45,41,40,45,50,52};
    v_type = {-1,-1,-3};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig167() {
    cout << "Start test_4v_1sp_fig167" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {68,69,65,64,64,65,72,72,69,68,
                    59,57,57,57,60,60,64,60,62,59,
                    52,53,50,45,45,53,52,57,50,52};
    v_type = {1,-1,-3};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig168() {
    cout << "Start test_4v_1sp_fig168" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {71,69,69,72,72,72,76,76,74,71,
                    68,69,65,67,69,65,72,72,69,68,
                    52,53,50,52,53,53,52,57,50,52};
    v_type = {1,1,-2};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig169() {
    cout << "Start test_4v_1sp_fig169" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {69,67,65,69,70,71,69,69,65,65,64,65,
                    60,60,60,62,65,67,65,64,62,57,60,57,
                    53,52,53,50,46,43,50,45,50,50,48,41};
    v_type = {0,-1,-3};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig170() {
    cout << "Start test_4v_1sp_fig170" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
    cp =           {60,60,60,60,58,59,57,69,65,65,64,65,
                    57,55,53,57,53,55,53,52,53,57,60,57,
                    53,52,53,41,46,43,50,45,50,50,48,41};
    v_type = {-1,-2,-3};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig171() {
    cout << "Start test_4v_1sp_fig171" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =           {69,67,65,65,65,67,65,64,65,65,64,65,
```

```
                    60,60,60,62,62,58,57,57,53,57,60,57,
                    53,52,53,50,46,43,50,45,50,50,48,41};
    v_type = {2,1,-1};
    findAllMUSes();
}

void FigureTests::test_4v_1sp_fig172() {
    cout << "Start test_4v_1sp_fig172" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53};
    cp =           {72,71,72,69,69,72,72,72,72,69,70,65,
                    69,67,64,65,65,67,69,67,65,65,64,65,
                    65,62,60,60,62,60,60,64,60,62,58,60};
    v_type = {3,2,1};
    findAllMUSes();
}

void FigureTests::test_4v_2sp_fig173() {
    cout << "Start test_4v_2sp_fig173" << endl;
    spList = {FIRST_SPECIES, SECOND_SPECIES, FIRST_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {57,62,55,55,59,62,60,64,62,61,62,
                    -1,53,57,59,60,55,59,57,55,52,57,53,52,53,52,60,57,53,57,57,57,
                      ↪
                    50,50,48,55,52,50,45,48,50,45,50};
    v_type = {-1,-1,-2};
    findAllMUSes();
}

void FigureTests::test_4v_2sp_fig174() {
    cout << "Start test_4v_2sp_fig174" << endl;
    spList = {SECOND_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =
      ↪  {-1,65,69,71,72,67,71,74,76,71,74,77,76,72,76,72,69,74,73,73,74,
                    57,53,55,55,55,57,60,60,53,57,57,
                    50,50,48,55,52,50,45,48,50,45,50};
    v_type = {1,-1,-2};
    findAllMUSes();
}

void FigureTests::test_4v_2sp_fig175() {
    cout << "Start test_4v_2sp_fig175" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, SECOND_SPECIES};
    cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62};
    cp =           {74,74,72,74,76,77,77,76,74,73,74,
                    69,69,69,71,71,74,72,72,69,69,69,
                    -1,62,50,53,57,60,55,53,52,55,50,52,53,57,60,48,50,53,57,45,50};
                      ↪
    v_type = {2,1,-2};
    findAllMUSes();
}

void FigureTests::test_4v_2sp_fig176() {
    cout << "Start test_4v_2sp_fig176" << endl;
    spList = {FIRST_SPECIES, SECOND_SPECIES, FIRST_SPECIES};
    cantusFirmus = {50,53,52,50,55,53,57,55,53,52,50};
    cp =           {65,   69,   72,   74,   70,   69,   69,   71,   74,   73,
      ↪  74,
                     -1,62,60,62,64,67,65,64,62,64,65,69,64,65,67,62,65,69,67,64,66,
                       ↪
                     57,   57,   55,   57,   58,   60,   60,   62,   57,   64,
                       ↪  57};
    v_type = {3, 2, 1};
    findAllMUSes();
}

void FigureTests::test_4v_3sp_fig183() {
```

```
    cout << "Start test_4v_3sp_fig183" << endl;
    spList = {THIRD_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =
      ↪ {71,64,71,72,69,72,69,67,65,62,64,65,67,64,65,67,69,65,69,71,72,69,72,74,76,71,76,74,72,76,72,71,69,74,6
      ↪
                    56,57,57,55,60,57,59,60,62,59,
                    52,53,50,52,53,53,52,57,50,52};
    v_type = {0, -1, -2};
    findAllMUSes();
}

void FigureTests::test_4v_3sp_fig184(){
    cout << "Start test_4v_3sp_fig184" << endl;
    spList = {THIRD_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus = {76,72,74,72,69,81,79,76,77,76};
    cp =
      ↪ {68,71,64,67,69,72,69,67,65,62,64,65,67,65,64,62,60,72,69,67,65,67,69,71,72,74,76,71,72,69,71,72,69,74,6
      ↪
                    59,57,57,55,57,60,64,60,62,59,
                    52,53,50,52,53,53,52,57,50,52};
    v_type = {-2, -4, -5};
    findAllMUSes();
}

void FigureTests::test_4v_3sp_fig185(){
    cout << "Start test_4v_3sp_fig185" << endl;
    spList = {FIRST_SPECIES, THIRD_SPECIES, FIRST_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {71,69,69,72,72,72,76,69,69,68,
                    56,59,52,55,57,60,57,55,53,57,55,53,55,52,64,62,60,59,57,55,53,55,57,59,60,62,64,62,60,57,60,5
                      ↪
                    52,53,50,52,53,53,48,48,50,52};
    v_type = {1, -2, -2};
    findAllMUSes();
}

void FigureTests::test_4v_3sp_fig186(){
    cout << "Start test_4v_3sp_fig186" << endl;
    spList = {FIRST_SPECIES, FIRST_SPECIES, THIRD_SPECIES};
    cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
    cp =           {71,76,77,76,77,77,76,76,69,68,
                    68,69,69,72,72,72,72,72,74,71,
                    64, 62, 60, 59, 57, 55, 53, 52, 50, 52, 53, 50, 57, 60, 57,
                      ↪ 55, 53, 55, 57, 55, 53, 55, 57, 59, 60, 52, 53, 55,
                      ↪ 57, 55, 53, 52, 50, 53, 52, 50, 52};
    v_type = {1, 1, -2};
    findAllMUSes();
}

void FigureTests::test_4v_4sp_fig196(){
    cout << "Start test_4v_4sp_fig196" << endl;
    spList = {FOURTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
    cp =            {57,57,62,62,60,60,59,59,62,62,57,57,65,65,64,64,62,62,61,62,
                    53, 57, 57, 50, 62, 62, 60, 60, 57, 57, 57,
                     50,50,45,47,43,50,53,48,50,45,50};
    v_type = {-1, -1, -3};
    findAllMUSes();
}

void FigureTests::test_4v_5sp_fig200(){
    cout << "Start test_4v_5sp_fig200" << endl;
    spList = {FIFTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
    cp =            {
                    -1,-1,69,-1,
                    69,-1,69,71,
```

```
                    72,67,72,-1,
                    72,-1,71,-1,
                    71,-1,71,73,
                    74,69,74,-1,
                    74,-1,77,-1,
                    77,-1,76,-1,
                    76,-1,74,-1,
                    74,-1,73,-1,
                    74,

                    57, 53, 55, 55, 55, 57, 57, 60, 57, 57, 57,
                    50,50,48,55,52,50,53,48,50,45,50};
    notesSpeciesFor5sp = {
                        -1, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,
    };
    v_type = {1, -1, -2};
    findAllMUSes();
}

void FigureTests::test_4v_5sp_fig201() {
    cout << "Start test_4v_5sp_fig201" << endl;
    spList = {FIFTH_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
    cantusFirmus =  {62,65,64,62,67,65,69,67,65,64,62};
    cp =            {
                    -1, -1, 57, -1,
                    57, -1, 57, 59,
                    60, 55, 60, -1,
                    60, -1, 59, -1,
                    59, -1, 59, 61,
                    62, 50, 62, -1,
                    62, -1, 65, -1,
                    65, -1, 64, -1,
                    64, -1, 62, -1,
                    62, -1, 61, -1,
                    62,

                    57, 53, 55, 55, 55, 57, 57, 60, 57, 57, 57,
                    50,50,48,55,52,50,53,48,50,45,50};
    notesSpeciesFor5sp = {
                        -1, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,-1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, THIRD_SPECIES, THIRD_SPECIES,
                        THIRD_SPECIES, THIRD_SPECIES, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES, -1, FOURTH_SPECIES, -1,
                        FOURTH_SPECIES,
    };
    v_type = {-1, -1, -2};
    findAllMUSes();
}

void FigureTests::test_4v_Xsp_fig204() {
    cout << "Start test_4v_Xsp_fig204" << endl;
    spList = {SECOND_SPECIES, THIRD_SPECIES, FOURTH_SPECIES};
```

```
1120        cantusFirmus =  {50,53,52,50,55,53,57,55,53,52,50};
1121        cp = {  69,74,72,69,71,72,74,62,70,67,69,65,72,69,70,67,69,65,67,64,66,
1122                62,64,65,67,69,67,65,69,67,65,64,67,65,62,65,64,62,60,58,62,65,67,69,65,64,62,60,57,62,60,58,55,60,59,
                      ↪
1123                57, 57, 62, 62, 60, 60, 58, 58, 62, 62, 60, 60, 65, 65, 64, 64, 62,
                      ↪  62, 61, 62,
1124            };
1125        v_type = {2,2,1};
1126        findAllMUSes();
1127    }
1128
1129    void FigureTests::run_twoVoice_tests() {
1130        cout << "Running two voice tests..." << endl;
1131        // Two voice first species figures
1132        test_2v_1sp_fig22();
1133        test_2v_1sp_fig23();
1134        // Two voice second species figures
1135        test_2v_2sp_fig38();
1136        test_2v_2sp_fig39();
1137        test_2v_2sp_fig40();
1138        test_2v_2sp_fig41();
1139        test_2v_2sp_fig42();
1140        test_2v_2sp_fig43();
1141        test_2v_2sp_fig44();
1142        test_2v_2sp_fig45();
1143        // Two voice third species figures
1144        test_2v_3sp_fig55();
1145        test_2v_3sp_fig56();
1146        test_2v_3sp_fig57();
1147        test_2v_3sp_fig58();
1148        test_2v_3sp_fig59();
1149        test_2v_3sp_fig60();
1150        // Two voice fourth species figures
1151        test_2v_4sp_fig74();
1152        test_2v_4sp_fig75();
1153        test_2v_4sp_fig76();
1154        test_2v_4sp_fig77();
1155        test_2v_4sp_fig78();
1156        // Two voice fifth species figures
1157        test_2v_5sp_fig82();
1158        test_2v_5sp_fig83();
1159        test_2v_5sp_fig87_1();
1160        test_2v_5sp_fig82();
1161        test_2v_5sp_fig83();
1162        test_2v_5sp_fig87_1();
1163    }
1164
1165    void FigureTests::run_threeVoice_tests() {
1166        cout << "Running three voice tests..." << endl;
1167        // Three voice first species figures
1168        test_3v_1sp_fig108();
1169        test_3v_1sp_fig109();
1170        test_3v_1sp_fig110();
1171        test_3v_1sp_fig111();
1172        test_3v_1sp_fig112();
1173        test_3v_1sp_fig113();
1174        test_3v_1sp_fig114();
1175        test_3v_1sp_fig115();
1176        test_3v_1sp_fig116();
1177        test_3v_1sp_fig117();
1178        test_3v_1sp_fig118();
1179        test_3v_1sp_fig119();
1180        // Three voice second species figures
1181        test_3v_2sp_fig125();
1182        test_3v_2sp_fig126();
1183        test_3v_2sp_fig127();
1184        test_3v_2sp_fig128();
1185        test_3v_2sp_fig129();
```

```
1186        // Three voice third species figures
1187        test_3v_3sp_fig130();
1188        test_3v_3sp_fig131();
1189        test_3v_3sp_fig132();
1190        test_3v_3sp_fig133();
1191        // Three voice fourth species figures
1192        test_3v_4sp_fig146();
1193        test_3v_4sp_fig147();
1194        test_3v_4sp_fig148();
1195        test_3v_4sp_fig149();
1196        test_3v_4sp_fig150();
1197        test_3v_4sp_fig151();
1198        // Three voice fifth species figures
1199        test_3v_5sp_fig154();
1200        test_3v_5sp_fig155();
1201        test_3v_5sp_fig156();
1202    }
1203
1204    void FigureTests::run_fourVoice_tests() {
1205        cout << "Running four voice tests..." << endl;
1206        // Four voice first species figures
1207        test_4v_1sp_fig166();
1208        test_4v_1sp_fig167();
1209        test_4v_1sp_fig168();
1210        test_4v_1sp_fig169();
1211        test_4v_1sp_fig170();
1212        test_4v_1sp_fig171();
1213        test_4v_1sp_fig172();
1214        // Four voice second species figures
1215        test_4v_2sp_fig173();
1216        test_4v_2sp_fig174();
1217        test_4v_2sp_fig175();
1218        test_4v_2sp_fig176();
1219        // Four voice third species figures
1220        test_4v_3sp_fig183();
1221        test_4v_3sp_fig184();
1222        test_4v_3sp_fig185();
1223        test_4v_3sp_fig186();
1224        // Four voice fourth species figures
1225        test_4v_4sp_fig196();
1226        // Four voice fifth species figures
1227        test_4v_5sp_fig201();
1228    }
1229
1230    void FigureTests:: run_thirdSpecies_tests() {
1231        cout << "Running third species tests..." << endl;
1232        // Two voice third species figures
1233        test_2v_3sp_fig55();
1234        test_2v_3sp_fig56();
1235        test_2v_3sp_fig57();
1236        test_2v_3sp_fig58();
1237        test_2v_3sp_fig59();
1238        test_2v_3sp_fig60();
1239        // Three voice third species figures
1240        test_3v_3sp_fig130();
1241        test_3v_3sp_fig131();
1242        test_3v_3sp_fig132();
1243        test_3v_3sp_fig133();
1244        // Four voice third species figures
1245        test_4v_3sp_fig183();
1246        test_4v_3sp_fig184();
1247        test_4v_3sp_fig185();
1248        test_4v_3sp_fig186();
1249        // Four voice fifth species figures
1250        test_4v_5sp_fig201();
1251    }
1252
1253    void FigureTests::run_fourthSpecies_tests() {
```

```
    cout << "Running four species tests..." << endl;
    // Two voice fourth species figures
    test_2v_4sp_fig74();
    test_2v_4sp_fig75();
    test_2v_4sp_fig76();
    test_2v_4sp_fig77();
    test_2v_4sp_fig78();
    // Three voice fourth species figures
    test_3v_4sp_fig146();
    test_3v_4sp_fig147();
    test_3v_4sp_fig148();
    test_3v_4sp_fig149();
    test_3v_4sp_fig150();
    test_3v_4sp_fig151();
    // Four voice fourth species figures
    test_4v_4sp_fig196();
}

void FigureTests::run_fifthSpecies_tests() {
    cout << "Running fifth species tests..." << endl;
    test_2v_5sp_fig82();
    test_2v_5sp_fig83();
    test_2v_5sp_fig87_1();
    test_3v_5sp_fig154();
    test_3v_5sp_fig155();
    test_3v_5sp_fig156();
    test_4v_5sp_fig201();
}

void FigureTests::run_all_tests() {
    cout << "Running all figure tests..." << endl;
    run_twoVoice_tests();
    run_threeVoice_tests();
    run_fourVoice_tests();
    test_4v_Xsp_fig204();
    cout << "All figure tests completed." << endl;
}

// Add to the MUSTest method to include our new test
void FigureTests::MUSTest() {
    cout << "Running MUSTest..." << endl;
    run_all_tests();

    cout << "MUSTest completed." << endl;
}

void FigureTests::quickTest() {
    // test_2v_1sp_fig22();
    // test_2v_2sp_fig39();
    // test_2v_2sp_fig40();
    // test_2v_2sp_fig41();
    // test_2v_2sp_fig42();
    // test_2v_2sp_fig43();
    // test_2v_2sp_fig44();
    // test_2v_2sp_fig45();
    // test_2v_3sp_fig55();
    // test_2v_3sp_fig56();
    // test_2v_3sp_fig57();
    // test_2v_3sp_fig58();
    // test_2v_4sp_fig74();
    // test_2v_4sp_fig76();
    // test_2v_4sp_fig77();
    // test_2v_4sp_fig78();
    // test_2v_5sp_fig82();
    // test_2v_5sp_fig83();
    // test_3v_1sp_fig108();
    // test_3v_1sp_fig109();
    // test_3v_1sp_fig110();
```

```
    test_3v_2sp_fig126();
    // test_3v_2sp_fig127();
    // test_3v_2sp_fig128();
    // test_3v_3sp_fig130();
    // test_3v_3sp_fig133();
    // test_3v_4sp_fig146();
    // test_3v_4sp_fig147();
    // test_3v_4sp_fig148();
    // test_3v_4sp_fig151();
    // test_3v_5sp_fig154();
    // test_3v_5sp_fig156();
    // test_4v_1sp_fig169();
    // test_4v_1sp_fig171();
    // test_4v_2sp_fig173();
    // test_4v_3sp_fig183();
    // test_4v_3sp_fig185();
    // test_4v_3sp_fig186();
    // test_4v_3sp_fig184();
    // test_4v_3sp_fig186();
    // test_4v_4sp_fig196();
    // test_4v_5sp_fig201();
    // test_4v_Xsp_fig204();
}
```

problem_wrapper.cpp

```
    //
// This file contains the implementations of the C++ interface functions.
//

#include "../headers/problem_wrapper.hpp"
#include "../headers/CounterpointProblems/CounterpointProblem.hpp"

#include "../headers/Utilities.hpp"
#include "../headers/CounterpointUtils.hpp"

std::vector<Species> convertToSpeciesVector(const std::vector<int>& intVec);


/**
 * Wraps the Problem constructor.
 * @return A pointer to a Problem object casted as a void*
 */
void* create_new_problem(int* cantusFirmus, int size, int n_cp, int* splist, int*
  ↪   v_types, int b_mode, int min_skips, int* general_params,
       int* motion_params, int* melodic, int* specific, int* importance, int
         ↪  t_off, int* scle, int scale_size,
       int* chromatic, int chrom_size, int* borrow, int borrow_size){

    writeToLogFile("Entered problem_wrapper.cpp . ");

    vector<int> cf(int_pointer_to_vector(cantusFirmus, size));
    vector<int> sp(int_pointer_to_vector(splist, n_cp));
    vector<int> mot(int_pointer_to_vector(motion_params, 3));
    vector<int> vt(int_pointer_to_vector(v_types, n_cp));
    vector<int> sc(int_pointer_to_vector(scle, scale_size));
    vector<int> chr(int_pointer_to_vector(chromatic, chrom_size));
    vector<int> brw(int_pointer_to_vector(borrow, borrow_size));
    vector<int> mel(int_pointer_to_vector(melodic, 8));
    vector<int> gen(int_pointer_to_vector(general_params, 8));
    vector<int> spec(int_pointer_to_vector(specific, 7));
    vector<int> imp(int_pointer_to_vector(importance, 14));

    vector<Species> speciesList = convertToSpeciesVector(sp);
    return create_problem(cf, speciesList, vt, mel, gen, spec, imp, b_mode);
```

```
}

/**
 * returns the size of the problem
 * @param sp a void* pointer to a CounterpointProblem object
 * @return an integer representing the size of the problem
 */
int get_size(void* sp){
    return static_cast<CounterpointProblem*>(sp)->getSize();
}


void delete_pointer(void* p){
    writeToLogFile("deleting counterpointproblem pointer...");
    delete static_cast<CounterpointProblem*>(p);
    writeToLogFile("deleted CounterpointProblem pointer . ");
}

void delete_solver_pointer(void* p){
    delete static_cast<DFS<CounterpointProblem>*>(p);
    writeToLogFile("deleted solver pointer");
}


/**
 * returns the values of the variables for a solution
 * @param sp a void* pointer to a Problem object
 * @return an int* pointer representing the values of the variables
 */
int* return_solution(void* sp){
    return static_cast<CounterpointProblem*>(sp)->return_solution();
}


// =========== NEEDED BY 5SP PARSER =================

int* return_species_array_5sp(void* sp, int ctp_index){
    return static_cast<CounterpointProblem*>(sp)->get_species_array_5sp(ctp_index
      ↪ );
}

int* return_extended_cp_domain(void* sp, int ctp_index){
    return static_cast<CounterpointProblem*>(sp)->get_extended_cp_domain(
      ↪ ctp_index);
}

int get_extended_cp_domain_size(void* sp, int ctp_index){
    return static_cast<CounterpointProblem*>(sp)->get_ext_cp_domain_size(
      ↪ ctp_index);
}

// ====================================================


/**
 * creates a search engine for Problem objects
 * @param sp a void* pointer to a Problem object
 * @return a void* cast of a Base<Problem>* pointer
 */
void* create_solver(void* sp, int type){
    return (void*) make_solver(static_cast<CounterpointProblem*>(sp), type);  //
      ↪ TODO
}

/**
 * returns the next solution space, it should be bound. If not, it will return
   ↪ NULL.
```

```
 * @param solver a void* pointer to a Base<CounterpointProblem>* pointer for the search engine of the problem
 * @return a void* cast of a CounterpointProblem* pointer
 */
void* return_next_solution_space(void* solver){
    writeToLogFile("return_next_solution_space_function_called");
    return (void*) get_next_solution_space(static_cast<BAB<CounterpointProblem>*>(solver));
}



int search_stopped(void* solver){
    return static_cast<int>(static_cast<Base<CounterpointProblem>*>(solver)->stopped());
}




// UTILITY FUNCTION JUST FOR PROBLEMWRAPPER
// Conversion function from std::vector<int> to std::vector<species>, DOES NOT HANDLE CANTUS FIRMUS because it is not supposed to be in spList
std::vector<Species> convertToSpeciesVector(const std::vector<int>& intVec) {
    std::vector<Species> speciesVec;
    speciesVec.reserve(intVec.size()); // Reserve space to avoid multiple allocations

    for (int value : intVec) {
        switch (value) {
            case 1:
                speciesVec.push_back(FIRST_SPECIES);
                break;
            case 2:
                speciesVec.push_back(SECOND_SPECIES);
                break;
            case 3:
                speciesVec.push_back(THIRD_SPECIES);
                break;
            case 4:
                speciesVec.push_back(FOURTH_SPECIES);
                break;
            case 5:
                speciesVec.push_back(FIFTH_SPECIES);
                break;
            default:
                std::cerr << "Warning: Value " << value << " is out of range for species enum\n";
                // Handle out of range values if needed
                // throw std::out_of_range("Value out of range for species enum");
                // speciesVec.push_back(static_cast<species>(0)); // Or handle it appropriately
                break;
        }
    }
    return speciesVec;
}
```

main.cpp

```
    //
// Created by Luc Cleenewerk and Diego de Patoul.
//
```

```
4
5   #include "headers/Utilities.hpp"
6   #include "headers/CounterpointUtils.hpp"
7   #include "headers/Parts/Part.hpp"
8   #include "headers/Parts/FirstSpeciesCounterpoint.hpp"
9   #include "headers/Parts/SecondSpeciesCounterpoint.hpp"
10  #include "headers/Parts/CantusFirmus.hpp"
11  #include "headers/CounterpointProblems/TwoVoiceCounterpoint.hpp"
12  #include "headers/CounterpointProblems/CounterpointProblem.hpp"
13  #include "headers/fuxTest.hpp"
14  #include "headers/figureTests.hpp"
15
16  using namespace Gecode;
17  using namespace std;
18
19  int main(int argc, char* argv[]) {
20      if (argc < 2) {
21          std::cout << "Please␣provide␣a␣test␣number␣or␣'figs'␣as␣argument" << std
                ↪  ::endl;
22          return 1;
23      }
24      if (consSize != constraintNames.size()) {
25          cout << "Error:␣constraintNames␣size␣does␣not␣match␣the␣number␣of␣
                ↪  constraints." << endl;
26          cout << "consSize␣=␣" << consSize << endl;
27          cout << "constraintNames.size()␣=␣" << constraintNames.size() << endl;
28          return 1;
29      }
30
31      string arg1 = argv[1];
32      if (arg1 == "figs") {
33          FigureTests figureTests;
34          if (argc > 2) {
35              string arg2 = argv[2];
36              if (arg2 == "2v") {
37                  figureTests.run_twoVoice_tests();
38              } else if (arg2 == "3v") {
39                  figureTests.run_threeVoice_tests();
40              } else if (arg2 == "4v") {
41                  figureTests.run_fourVoice_tests();
42              } else if (arg2 == "4sp") {
43                  figureTests.run_fourthSpecies_tests();
44              } else if (arg2 == "5sp") {
45                  figureTests.run_fifthSpecies_tests();
46              } else if (arg2 == "MUS") {
47                  figureTests.MUSTest();
48              } else if (arg2 == "quick") {
49                  figureTests.quickTest();
50              }
51              else {
52                  std::cout << "Invalid␣argument:␣" << arg2 << std::endl;
53                  return 1;
54              }
55          } else {
56              figureTests.run_all_tests();
57          }
58          return 0;
59      }
60
61      if(argc==1){
62          cout << argv[0] << endl;
63          cout << "-------------" << endl;
64          vector<Species> species = {THIRD_SPECIES};
65          //la do si re do mi fa mi re do si la
66          //57 60 59 62 60 64 65 64 62 60 59 57
67          // vector<int> cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57}; //1
                ↪  sp 2v cf
```

```
68          vector<int> cantusFirmus = {60,   62,   65,   64,   67,   65,   64,   62,
              ↪     60};
69          //cantusFirmus = {62,   65,   64,   62,   67,   65,   69,   67,   65, 64,
              ↪   62};
70
71          int size = cantusFirmus.size();
72          vector<int> v_type = {2 , 1, -1};
73
74          vector<int> melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
75          //borrow, h-5th, h-octave, succ, variety, triad, direct move, penult rule
              ↪   check
76          vector<int> general_params = {4, 1, 1, 2, 2, 2, 8, 1};
77
78          //penult sixth, non-ciambata, con m after skip, h triad 3rd species, m2
              ↪  eq zero, no syncopation, pref species slider
79          vector<int> specific_params = {8 , 4 , 0 , 2 , 1 , 8 , 50};
80
81
82          vector<int> importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
83
84          int borrowMode = 1;
85
86          // create a new problem
87          // auto* problem = new TwoVoiceCounterpoint(cantusFirmus, species[0], C,
              ↪  lower_bound_domain, upper_bound_domain);
88          auto* problem = create_problem(cantusFirmus, species, v_type,
              ↪  melodic_params, general_params, specific_params,
89            importance, borrowMode);
90
91          //solution from illustration 8.4
92          //vector<int> sol =
              ↪  {76,77,79,77,74,76,83,72,74,72,74,72,79,72,74,76,81,77,79,83,81,76,77,79,81,74,76,83,81,79,77,79,76
              ↪
93          //for(int i = 0; i < problem->getSize(); i++){
94          //    rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, sol
              ↪  [i]);
95          //}
96          //solution from illustration 8.2
97          //costs : ...,...,octave,...,fifth,...,...,melodic,motion
98          //vector<int> sol =
              ↪  {88,81,81,83,83,84,72,74,79,67,65,60,67,62,69,69,69,64,48,50,53,52,55,53,57,65,60};
              ↪
99          //for(int i = 0; i < problem->getSize(); i++){
100         //    rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, sol
              ↪  [i]);
101         //}
102
103         //solution from illustration 8.3
104         //costs : ...,...,octave,...,fifth,...,...,melodic,motion
105         //vector<int> sol =
              ↪  {76,81,74,86,83,86,79,88,81,88,86,89,81,79,89,86,79,67,65,62,60,64,69,69,69,64,48,50,50,48,48,50,48
              ↪
106         //for(int i = 0; i < problem->getSize(); i++){
107         //    rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, sol
              ↪  [i]);
108         //}
109
110         //solution from illustration 8.5
111         //costs : ...,...,octave,...,fifth,...,...,melodic,motion
112         //vector<int> sol =
              ↪  {72,72,76,76,81,81,74,74,76,76,83,83,79,79,77,72,60,65,62,60,64,62,69,69,64,48,50,50,48,48,50,48,50
              ↪
113         //for(int i = 0; i < problem->getSize(); i++){
114         //    rel(problem->getHome(), problem->getSolutionArray()[i], IRT_EQ, sol
              ↪  [i]);
115         //}
116
117         /*
```

```
        rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 57);
        rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 57);
        rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 60);
        rel(problem->getHome(), problem->getSolutionArray()[3], IRT_EQ, 59);
        rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 59);

        rel(problem->getHome(), problem->getSolutionArray()[11], IRT_EQ, 53);
        rel(problem->getHome(), problem->getSolutionArray()[12], IRT_EQ, 53);
        rel(problem->getHome(), problem->getSolutionArray()[13], IRT_EQ, 55);
        rel(problem->getHome(), problem->getSolutionArray()[14], IRT_EQ, 55);
        rel(problem->getHome(), problem->getSolutionArray()[15], IRT_EQ, 55);

        rel(problem->getHome(), problem->getSolutionArray()[22], IRT_EQ, 50);
        rel(problem->getHome(), problem->getSolutionArray()[23], IRT_EQ, 50);
        rel(problem->getHome(), problem->getSolutionArray()[24], IRT_EQ, 48);
        rel(problem->getHome(), problem->getSolutionArray()[25], IRT_EQ, 55);
        rel(problem->getHome(), problem->getSolutionArray()[16], IRT_EQ, 52);*/
        Search::Base<CounterpointProblem>* e = make_solver(problem, bab_solver);
        int nb_sol = 0;
        while(CounterpointProblem* pb = get_next_solution_space(e)){

            nb_sol++;
            cout << "Solution " << nb_sol << ": " << endl;
            cout << pb->to_string() << endl;
            cout << pb->getSize() << endl;
            // cout << int_vector_to_string(cantusFirmus) << endl;

            delete pb;
            if (nb_sol >= 1)
                break;
        }
        cout << "No (more) solutions." << endl;
    } else if(argc==3){ // Run test targetting a figure example by fixing the
      ↪  fisrt n measures
        FuxTest* test = new FuxTest(atoi(argv[1]), atoi(argv[2]));
        vector<Species> species = test->getSpList();
        //la do si re do mi fa mi re do si la
        //57 60 59 62 60 64 65 64 62 60 59 57
        vector<int> cantusFirmus = test->getCf();

        int size = cantusFirmus.size();
        vector<int> v_type = test->getVType();

        vector<int> melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
        //borrow, h-5th, h-octave, succ, variety, triad, direct move, penult rule
          ↪   check
        vector<int> general_params = {4, 1, 1, 2, 2, 2, 8, 1};

        //penult sixth, non-ciambata, con m after skip, h triad 3rd species, m2
          ↪  eq zero, no syncopation, pref species slider
        vector<int> specific_params = {8 , 4 , 0 , 2 , 1 , 8 , 50};

        vector<int> importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};

        int borrowMode = test->getBMode();

        // create a new problem
        // auto* problem = new TwoVoiceCounterpoint(cantusFirmus, species[0], C,
          ↪  lower_bound_domain, upper_bound_domain);
        auto* problem = create_problem(cantusFirmus, species, v_type,
          ↪  melodic_params, general_params, specific_params,
            importance, borrowMode);
        //cout << problem->to_string() << endl;
        // create a new search engine

        if(atoi(argv[1])==125){
            for(int j = 0; j < problem->getSize(); j++){
                if(j!=10){
```

```
                    rel(problem->getHome(), problem->getSolutionArray()[j],
                      ↪ IRT_EQ, test->getCp()[j]);
                }
            }
        }
        else if(atoi(argv[1])==74){
            for(int j = 0; j < test->getIdx(); j++){
                if(j!=0){
                    rel(problem->getHome(), problem->getSolutionArray()[j],
                      ↪ IRT_EQ, test->getCp()[j]);
                }
            }
        }
        else if(atoi(argv[1])==82){
            vector<int> speciesArray =
              ↪ {-1,-1,3,-1,3,2,2,2,2,2,2,2,2,3,-1,3,2,2,2,2,2,3,-1,3,-1,3,-1,3,2,3,-1,3,2,3,-1,3,-1,3,-1,3};
              ↪
            for(int j = 0; j < test->getIdx(); j++){
                if(j>1){
                    rel(problem->getHome(), problem->getSolutionArray()[j],
                      ↪ IRT_EQ, test->getCp()[j]);
                    //rel(problem->getHome(), problem->get_species_array_5sp(0)[j
                      ↪ ], IRT_EQ, test->getCp()[j]);
                }
            }
        }
        else{
            if(species[0]==FIRST_SPECIES || species[0]==THIRD_SPECIES || species
              ↪ [0]==FOURTH_SPECIES){
                for(int j = 0; j < test->getIdx(); j++){
                    rel(problem->getHome(), problem->getSolutionArray()[j],
                      ↪ IRT_EQ, test->getCp()[j]);
                }
            } else if(species[0]==SECOND_SPECIES){
                for(int j = 1; j < test->getIdx(); j++){
                    rel(problem->getHome(), problem->getSolutionArray()[j],
                      ↪ IRT_EQ, test->getCp()[j]);
                }
            }
        }

        BAB<CounterpointProblem> e(problem);

        int nb_sol = 0;
        while(CounterpointProblem* pb = e.next()){
            nb_sol++;
            cout << "Solution␣" << nb_sol << ":␣" << endl;
            cout << pb->to_string() << endl;
            // cout << int_vector_to_string(cantusFirmus) << endl;

            delete pb;
            if (nb_sol >= 1)
                break;
        }
        cout << "No␣(more)␣solutions." << endl;
    } else if(argc==2){ // TESTS
        char* str = argv[1];
        if(!notInt(str)){ // Run test targetting a figure example
            FuxTest* test = new FuxTest(atoi(argv[1]));
            vector<Species> species = test->getSpList();
            //la do si re do mi fa mi re do si la
            //57 60 59 62 60 64 65 64 62 60 59 57
            vector<int> cantusFirmus = test->getCf();

            int size = cantusFirmus.size();
            vector<int> v_type = test->getVType();

            vector<int> melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
```

```
            //borrow, h-5th, h-octave, succ, variety, triad, direct move, penult
              ↪ rule check
            vector<int> general_params = {4, 1, 1, 2, 2, 2, 8, 1};

            //penult sixth, non-ciambata, con m after skip, h triad 3rd species,
              ↪ m2 eq zero, no syncopation, pref species slider
            vector<int> specific_params = {8 , 4 , 0 , 2 , 1 , 8 , 50};

            vector<int> importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};

            int borrowMode = test->getBMode();

            // create a new problem
            // auto* problem = new TwoVoiceCounterpoint(cantusFirmus, species[0],
              ↪  C, lower_bound_domain, upper_bound_domain);
            auto* problem = create_problem(cantusFirmus, species, v_type,
              ↪ melodic_params, general_params, specific_params,
                importance, borrowMode);
            //cout << problem->to_string() << endl;
            // create a new search engine
            if(atoi(argv[1])==125){
                cout << "HERE" << endl;
                for(int j = 0; j < problem->getSize(); j++){
                    if(j!=10){
                        rel(problem->getHome(), problem->getSolutionArray()[j],
                          ↪ IRT_EQ, test->getCp()[j]);
                    }
                }
            }
            else{
                if(species[0]==FIRST_SPECIES || species[0]==THIRD_SPECIES){
                    for(int j = 0; j < problem->getSize(); j++){
                        rel(problem->getHome(), problem->getSolutionArray()[j],
                          ↪ IRT_EQ, test->getCp()[j]);
                    }
                } else if(species[0]==SECOND_SPECIES || species[0]==
                  ↪ FOURTH_SPECIES){
                    for(int j = 1; j < problem->getSize(); j++){
                        rel(problem->getHome(), problem->getSolutionArray()[j],
                          ↪ IRT_EQ, test->getCp()[j]);
                    }
                }
            }

            BAB<CounterpointProblem> e(problem);

            int nb_sol = 0;
            while(CounterpointProblem* pb = e.next()){
                nb_sol++;

                // cout << int_vector_to_string(cantusFirmus) << endl;
                cout << "Solution " << nb_sol << ": " << endl;
                cout << pb->to_string() << endl;
                delete pb;
                if (nb_sol >= 1)
                    break;
            }
            if(nb_sol!=1){
                cout << "This test did not pass. An error was found!" << endl;
            } else {
                cout << "This test passed successfully!" << endl;
            }
        } else { // Run test targetting a specific constraint
            FuxTest* test = new FuxTest(argv[1]);
        }
    } else {
        throw std::invalid_argument("Wrong amount of arguments!");
    }
```

```
300
301     return 0;
302 }
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**