

École polytechnique de Louvain

FuxCP : A Constraint Programming Formalization of Fux's Musical Theory of Four-Voice Counterpoint

Authors: **Luc CLEENEWERK, Diego DE PATOUL**

Supervisor: **Peter VAN ROY**

Readers: **Augustin DELECLUSE, Karim HADDAD, Damien SPROCK-
EELS**

Academic year 2023–2024

Master [120] in Computer Science and Engineering

Abstract

This thesis presents a constraint-based formalization of four-voice musical counterpoint according to the classical theory established by Johann Joseph Fux in his 1725 treatise *Gradus ad Parnassum*. It builds upon the theses of Thibault Wafflard and Anton Lamotte, which provided first formalizations of two and three voice counterpoint, respectively. This thesis presents a unified and completed set of mathematical rules for composing four-voice counterpoint, and a C++ constraint programming implementation of this theoretical framework using the Gecode solver. The resulting software is FuxCP, a tool meant to assist composers in writing counterpoint in the style of Fux. An interface to the C++ implementation is provided in OpenMusic, a musical composition environment based on Common Lisp.

The research emphasizes the advantages of constraint programming in the context of music formalization over other approaches such as machine learning. It argues that constraint programming provides transparency and flexibility, and guarantees adherence to Fux's rules, which other methods may lack. This thesis also demonstrates the applicability of Fux's theory to an inheritance-based object-oriented implementation, while acknowledging that this paradigm can only approximate the sophisticated reality of the theory. The implemented inheritance-based structure contributes to a considerably more elegant and robust code.

After presenting examples of the generated compositions, the thesis concludes by discussing the current limitations of FuxCP and proposing directions for future improvements. It emphasizes that while this work represents a crucial step in modelization, an optimization effort is now necessary to effectively navigate the vast search space and guide the solver towards optimal solutions.

Acknowledgements

First and foremost, we would like to sincerely thank Peter Van Roy for supervising this thesis. His genuine enthusiasm for the project and his ability to always help us see the big picture kept our motivation alive during challenging times.

We would also like to thank Damien Sprockeels for sharing his valuable experience in music and constraint programming, and for providing us with a C++ base code to build upon.

Our gratitude also goes to Karim Haddad for supporting this project from the start and for the initial idea to formalize Fux’s theory, and to Augustin Delecluse for taking the time to read and evaluate this thesis.

Special thanks go to Anton Lamotte and Thibault Wafflard for having provided such quality work to carry forward, and for being open to questions and providing helpful suggestions. Their continued interest in the project, even after their direct involvement had concluded, reflect the remarkable dedication and energy they invested in their work.

Finally, a heartfelt **thank you** to our friends, family and loved ones. Their unwavering support and encouragement throughout this journey were invaluable to us, and we would not have achieved this without them.

Contents

1	Introduction and Context of This Work	1
1.1	Music, Mathematics and Computer Science	1
1.2	Fux and Counterpoint	2
1.2.1	The Style of Counterpoint	2
1.2.2	Johann Joseph Fux and his <i>Gradus ad Parnassum</i>	3
1.3	About Constraint Programming and Machine Learning	4
1.4	Previous Works in Computer Counterpoint	5
1.5	Our Contributions	6
1.6	Roadmap for Reading This Thesis	7
2	Tools and Software Used	9
2.1	Constraint Programming	9
2.1.1	Basic overview of Constraint Programming	9
2.1.2	Gecode	10
2.2	OpenMusic	10
2.3	GiL and CFFI	11
3	Methodology for Formalizing Fux’s Work	13
3.1	Theoretical Work	13
3.1.1	Generalizing Existing Rules and Variables	13
3.1.2	Adding New Rules for Four-Voice Composition	14
3.2	Practical Work	14
3.2.1	Migrating from Lisp to C++	14
3.2.2	Incorporating Theoretical Changes and New Rules	15
3.2.3	Testing with Fux’s Examples	15
4	Concepts and Variables Used in the Formalization	16
4.1	Voices, Parts, and Strata	16
4.2	Variables and Arrays	18
4.2.1	Variables	19
4.2.2	Obsolete Variable Array : IsCfB	23
5	Rules for Four Voice Counterpoint	25
5.1	First Species	26
5.1.1	Explicit Rules	26
5.1.2	Implicit Rules	29
5.2	Second Species	31
5.3	Third Species	32
5.3.1	Explicit Rules	33
5.3.2	Implicit Rules	33
5.4	Fourth Species	34
5.5	Fifth Species	36
5.6	Combining Different Species	37

6	Complete Set of Rules and Their Scopes	38
6.1	General and Implicit Rules	38
6.2	Constraints of the First Species	40
6.2.1	Harmonic Rules	40
6.2.2	Melodic Rules	41
6.2.3	Motion Rules	42
6.3	Constraints of the Second Species	43
6.3.1	Harmonic Rules	43
6.3.2	Melodic Rules	44
6.3.3	Motion Rules	44
6.4	Constraints of the Third Species	45
6.4.1	Harmonic Rules	45
6.4.2	Melodic Rules	46
6.4.3	Motion Rules	46
6.5	Constraints of the Fourth Species	46
6.5.1	Harmonic Rules	46
6.5.2	Melodic Rules	47
6.5.3	Motion Rules	47
6.6	Constraints of the Fifth Species	48
6.7	General look back on the set of rules	50
7	Program Architecture	51
7.1	Inheritance-based Structure	51
7.1.1	Centralizing common structural elements	52
7.1.2	Posting the right set of constraints	52
7.1.3	Hierarchy of the main Problem classes	55
7.2	Polymorphism	56
7.3	Functional Abstraction	56
7.4	Flow	56
7.5	Conclusion	57
8	Solution Search, Examples and Evaluation	58
8.1	Verifying that the CSP is correctly defined	58
8.1.1	Completeness : Accepting Correct solutions	58
8.1.2	Soundness : Rejecting Incorrect Solutions	59
8.2	Searching for Good Solutions	60
8.2.1	Type of Minimization	60
8.2.2	Branching	60
8.3	Using Next Solution or Best Solution	60
8.4	Evaluation by species	62
8.4.1	First species evaluation	62
8.4.2	Second species evaluation	63
8.4.3	Third species evaluation	63
8.4.4	Fourth species evaluation	64
8.4.5	Fifth species evaluation	64
8.5	Combining different species	65
8.6	Conclusion	65

9	Obstacles Encountered, Known Issues and Future Improvements	66
9.1	Obstacles encountered during the work on this thesis	66
9.2	Known Issues	67
9.3	Future Improvements	67
9.3.1	Solution Search	67
9.3.2	FuxCP User Experience	68
9.3.3	Beyond <i>Gradus ad Parnassum</i>	68
10	Conclusion	69
	Bibliography	69
A	Constants and Functions used in the mathematical formalization	73
B	User Guide	76
B.1	Installing FuxCP	76
B.1.1	Prerequisites	76
B.1.2	Loading FuxCP in OpenMusic	76
B.2	Using FuxCP in OpenMusic	77
B.3	Running directly from C++	80
C	Complete FuxCP Code	83
C.1	Counterpoint header files	83
C.1.1	CounterpointProblem.hpp	83
C.1.2	FourVoiceCounterpoint.hpp	85
C.1.3	ThreeVoiceCounterpoint.hpp	85
C.1.4	TwoVoiceCounterpoint.hpp	86
C.2	Voice header files	87
C.2.1	CantusFirmus.hpp	87
C.2.2	FifthSpeciesCounterpoint.hpp	88
C.2.3	FirstSpeciesCounterpoint.hpp	89
C.2.4	FourthSpeciesCounterpoint.hpp	91
C.2.5	Part.hpp	93
C.2.6	SecondSpeciesCounterpoint.hpp	96
C.2.7	ThirdSpeciesCounterpoint.hpp	97
C.3	Other header files	98
C.3.1	constraints.hpp	98
C.3.2	CounterpointUtils.hpp	101
C.3.3	fuxTest.hpp	101
C.3.4	problem_wrapper.hpp	103
C.3.5	Stratum.hpp	104
C.3.6	Utilities.hpp	105
C.3.7	Voice.hpp	112
C.4	Counterpoint source files	113
C.4.1	CounterpointProblem.cpp	113
C.4.2	FourVoiceCounterpoint.cpp	121
C.4.3	ThreeVoiceCounterpoint.cpp	125
C.4.4	TwoVoiceCounterpoint.cpp	128
C.5	Voice source files	130
C.5.1	CantusFirmus.cpp	130
C.5.2	FifthSpeciesCounterpoint.cpp	132
C.5.3	FirstSpeciesCounterpoint.cpp	146

C.5.4	FourthSpeciesCounterpoint.cpp	151
C.5.5	Part.cpp	157
C.5.6	SecondSpeciesCounterpoint.cpp	162
C.5.7	ThirdSpeciesCounterpoint.cpp	167
C.6	Other source files	173
C.6.1	constraints.cpp	173
C.6.2	CounterpointUtils.cpp	184
C.6.3	fuxTest.cpp	186
C.6.4	problem_wrapper.cpp	193
C.6.5	Stratum.cpp	195
C.6.6	Utilities.cpp	197
C.6.7	Voice.cpp	202
C.7	Lisp files	203
C.7.1	interface.lisp	203
C.7.2	package.lisp	225
C.7.3	problem-wrapper.lisp	225
C.7.4	FuxCP.lisp	228

Chapter 1

Introduction and Context of This Work

This thesis presents a mathematical, constraint-based formalization of Johann Joseph Fux's classical theory of counterpoint, as described in his foundational treatise *Gradus ad Parnassum*. It builds upon two previous theses by Thibault Wafflard [1] and Anton Lamotte [2], who formalized Fux's counterpoint in two and three voices, respectively. This thesis presents not only the extension to four voices, but completely reimagines the structure in an object-oriented strategy based in C++.

The resulting software is meant as a tool to help composers write counterpoint in the style of Fux. This tool is implemented in OpenMusic (OM) [3], a musical interface application created by IRCAM. The Constraint Satisfaction Problem is implemented in Gecode [4], using C++, and the link between OM and Gecode is made possible by the GiL Framework [5].

After an introductory chapter about formalizing music in general, and Fux's work more specifically, we will move on to describe the tools and software used in this formalization. A third chapter will then detail the methodology used throughout this thesis. Next, we will move on to the core of the thesis by first laying out the structure and concepts of the formalization in Chapter 4, providing a detailed explanation of the new rules for four-voice composition in Chapter 5, and giving the full generalized set of mathematical rules in Chapter 6. The seventh chapter will then detail the workings of the C++ implementation and its object-oriented structure. Finally, we will look at concrete examples of musical compositions generated by the solver, before concluding with a hopeful outlook on the future of this project.

A basic overview of constraint programming will be given in Chapter 2 of this thesis. However, this thesis assumes a basic understanding of music theory concepts. An overview of these concepts can be found in the first chapter of Thibault Wafflard's thesis [1].

1.1 Music, Mathematics and Computer Science

Music and mathematics have long shared a profound connection, rooted in the very fabric of both disciplines. From ancient times, mathematical principles have been essential in the understanding and creation of music. The Pythagoreans, for example, discovered that harmonious musical intervals corresponded to simple numerical ratios, establishing the foundation for Western musical theory.

Throughout history, composers have utilized mathematical structures, formulas, and methodologies in their creative processes. "Bach used the symmetries of retrograde and inversion; Mozart invented the musical dice game; Bartók applied Fibonacci numbers to organize time, Messiaen, Boulez, Pousseur, Eimert, Stockhausen, and others applied group theory to create their serial compositions, extending the thoroughly

mathematical dodecaphonic ideas of Schönberg and Hauer." [6]

This intrinsic relationship between music and mathematics naturally extends to the realm of computer science. Since the early days of computing, music has captivated the interest of computer scientists. This is a logical evolution, since computers revolutionized the way we handle complex mathematical tasks, and as we have seen, music composition often involves many mathematical techniques. As a result, the history of computer music dates back to the 1950s, with notable achievements such as the Illiac Suite, one of the first compositions generated by a computer program, and David Cope's symphony composed in the style of Mozart. [7]

It must be noted that all of these parallels between musical composition and mathematical structures are especially true with tonal music, which is intrinsically more mathematical and formalized, and has thus been of even greater interest to computer scientists. Tonal music represents the vast majority of music in the western world : pop, rock, blues, as well as most classical music all rely heavily on **tonality**: the organization of musical compositions around a central note, the tonic, using major and minor scales and chords. [8]

1.2 Fux and Counterpoint

1.2.1 The Style of Counterpoint

We have seen that tonal music inherently involves structure and organization, which makes it particularly interesting to mathematicians and computer scientists, and anyone interested in building a rigorous formalization of music. To delve deeper, one of the most mathematically rigorous aspects of tonal music is **counterpoint**. Counterpoint is the art of combining different melodic lines in a musical composition.

"Good counterpoint requires two qualities: (1) a meaningful or harmonious relationship between the lines (a "vertical" consideration—i.e., dealing with harmony) and (2) some degree of independence or individuality within the lines themselves (a "horizontal" consideration, dealing with melody)." [9]

In other words, these melodic lines are harmonically interdependent but melodically independent. Counterpoint is built from a given melodic line, the *cantus firmus*, which gives the base against which the various counterpoints are written.

1.2.2 Johann Joseph Fux and his *Gradus ad Parnassum*



Figure 1.1: Johann Joseph Fux (1660-1741) [10]

In order to write good counterpoint, a certain set of rules needs to be observed. For a long time, these rules were always implicit, untold and vague, varying from one composer to another. Before long, many composers tried to tackle the complex task of writing an exhaustive set of rules explicitly, in what is called a harmonic treatise. One of the most influential harmonic treatises, which inspired many great composers like Bach, Mozart and Haydn, is *Gradus ad Parnassum* by Johann Joseph Fux. Published in 1725, this book is written as a dialogue between a master and his pupil, in which the master explains the different rules to follow when writing counterpoint.

Species Counterpoint

Fux defines five different species of counterpoint.

- The **first species** is note against note counterpoint. In other words, for every note in the *cantus firmus*, there is a corresponding note in the counterpoint. Theoretically, the *cantus firmus* could have an arbitrary structure, but to simplify his rules and examples, Fux chose to make all *cantus firmus* have just one whole note per measure. Therefore, a counterpoint of the first species also has one note per measure.



Figure 1.2: Example of a first species counterpoint composition. Mann [11, p.29]

- The **second species** has two half notes for every whole note of the *cantus firmus*. It thus introduces the concepts of *thesis*, the downbeat, and *arsis*, the upbeat.
- The **third species** continues this pattern, now having four quarter notes for every whole note in the *cantus firmus*, allowing for even more richness and freedom in composition, which also implies more (computational) complexity.

- The **fourth species** is called the *ligature* or *syncopation*. It consists of half notes, but ligated in such a way that they sound like whole notes occurring on the upbeat.
- The **fifth species** is *florid counterpoint*. It is the richest form of counterpoint, combining all of the previous species.

The influence of Fux's treatise, as well as its clear separation into different species and number of voices, justify the choice to use it as a base to mathematically formalize classic counterpoint.

1.3 About Constraint Programming and Machine Learning

Now that Fux's treatise has been selected as the basis for formalizing the style of counterpoint, it is time to choose a paradigm to implement this formalization.

Artificial Intelligence (AI) is pervasive in today's world, influencing various fields, including music. Among the most hyped and popular AI techniques is machine learning (ML), particularly deep learning. ML has been used to generate music, analyze compositions, and emulate the styles of famous composers. However, despite its widespread use, ML has significant limitations, particularly in terms of transparency and control.

Another modeling technique that can be qualified as AI – some would rightly say that it is a truer form of Artificial Intelligence than ML – is constraint programming (CP). CP is a method of problem-solving where relationships between variables are expressed as constraints, and solutions are found by satisfying these constraints. CP's transparency and control make it especially suitable for applications where adherence to specific rules is crucial. More detail on constraint programming will be given in the next chapter.

Choosing CP over ML for modeling the art of counterpoint is motivated by two main reasons:

Transparency and Flexibility: ML operates as a black box, with complex models that often lack transparency and provide limited control over the generated output. In contrast, CP allows for full visibility into how solutions are generated and provides complete control over the rules that apply to the composition process. As Thibault Wafflard noted, "Unlike machine learning, constraint programming allows full transparency on how solutions are generated and full control over the musical rules that apply. In a way, this paradigm allows the model to 'understand' the music it generates. This is practical to leave to the composer the possibility of making the music he truly wants. " [1]

Few have studied and summarized the challenges of using deep learning for music generation better than Briot and Pachet :

"A direct application of deep learning to generate content rapidly reaches limits as the generated content tends to mimic the training set without exhibiting true creativity. [...] Deep learning architectures alone are autistic automata which generate music autonomously without human user interaction, far from the objective of interactively assisting musicians to compose and refine music. " [12]

Direct Encoding of Rules: ML aims to learn an implicit set of rules (which are not explicitly formalized and remain hidden inside the model) to best model a given problem. Typically, ML tries to estimate a function that describes some real-world phenomenon, such as mapping faces to genders. In most applications, the underlying "function" is extremely complex and cannot be completely formalized in mathematical terms. Therefore, ML uses complex mathematical models to approximate this function, but the result almost never fully describes the underlying reality. As George Box put it, "all models are wrong, but some are useful".

However, in the context of counterpoint, we have the immense advantage of having a well-defined set of rules, thanks to Johann Joseph Fux's detailed work in *Gradus ad Parnassum*. This means that instead of approximating an unknown function, we can directly encode these established rules into a constraint model. This direct encoding is feasible and highly effective because the rules of counterpoint are already clearly articulated. Using ML for counterpoint would be counterproductive, as it would involve trying to use complex mathematical models to estimate a function that we already know precisely. By formalizing Fux's rules into a CP model, we leverage this existing knowledge directly, ensuring that the generated music adheres strictly to the principles of counterpoint as intended by its greatest practitioners.

For these two reasons, CP is a much more intelligent choice to formalize counterpoint. It allows FuxCP to be a transparent, flexible, and musically "correct" tool for composers. There is no guesswork involved as there would be with ML: the counterpoints generated are created using the exact same rules employed by the greatest counterpoint composers.

1.4 Previous Works in Computer Counterpoint

We have now chosen a foundational treatise to formalize, and a paradigm to formalize it. This section will give a summary of previous efforts in creating a software tool to generate counterpoint, showing that none of them have reached a satisfactory level for composers (nor a complete mathematical formalization of counterpoint), thus justifying the need for FuxCP. It should be noted that these previous efforts have not necessarily chosen Fux's work as a basis nor CP as a formalization framework, as these are the particular choices of FuxCP. This section is a summary of Lamotte's section about previous works, which gives sufficient detail and remains up to date.

1987 : Early Attempts

Schottstaedt, in 1987, [13] developed an expert system based on Fux's rules using over 300 if-else clauses. This method had limitations compared to modern constraints, as if-else clauses are unidirectional and less flexible than constraint systems, which can represent sets of potential solutions and specify intricate search heuristics.

1997-2004 : Genetic Programming

In 1997, Polito et al. [14] used a genetic programming approach to optimize fifth-species counterpoint. This approach was promising but insufficient for complete compositions. A similar method was applied to generating fugues in 2004 [15], yielding interesting results but still falling short of giving a full counterpoint composition.

2010-2012 : Probabilistic and Search Algorithms

Aguilera et al. (2010) [16] developed an automated method for generating first-species counterpoint using probabilistic logic, specifically for compositions in C major. This approach evaluated harmonic attributes but neglected melodic aspects.

Herremans and Sörensen (2012) [17] used a variable neighborhood search algorithm to generate high-quality first-species counterpoint. Their research presented a user-friendly interface and implemented preferences, but only formalized first species, two-voice counterpoint.

2015 : Dominance Relations for Fux’s Counterpoint

In 2015, Maciej Komosinski and Piotr Szachewicz [18] conducted research in which they used dominance relations for generating first species counterpoint. This strategy yielded good results, but the research was (again) limited to the first species of counterpoint.

2019-2021 : User-Ready Applications

Two notable applications, both named ‘Counterpointer’, offer tools for composing counterpoint. The first, released in 2019 [19], checks the validity of user-submitted counterpoints according to traditional rules and provides feedback but does not generate counterpoints automatically. It is closed source and its underlying paradigm is unknown.

The second Counterpointer project [20], created in 2021 by students at Brown University, generates two-voice counterpoints for the first, second, and third species. Although open source, the project is less accomplished and has been discontinued.

Conclusion

This summary highlights the advancements and limitations of previous works in counterpoint generation. Despite promising developments, there remains no satisfactory tool for composing counterpoint in a user-friendly way, with good quality, quickly, and for multiple voices. This gap is the focus of our current research, building on the foundations laid by T. Wafflard’s thesis and A. Lamotte’s continuation.

1.5 Our Contributions

As this introduction shows, we are standing on the shoulders of giants. The quality and sheer quantity of the work that precedes this thesis, from Fux’s life work of understanding and putting into words the inner workings of Beauty in counterpoint, to the various attempts at making machines create this Beauty themselves, have laid a robust foundation for our research. We feel grateful to write our thesis within today’s context, where advancements in computational power and algorithms, coupled with the rich literature that precedes us, allow us to push the boundaries of what can be achieved in the intersection of music theory and artificial intelligence.

Building upon these achievements, the main contribution of this work is the design, implementation, and testing of FuxCP, a constraint formalization of the Fux theory of four voice counterpoint, written as a constraint program in C++ using the Gecode solver and with a user interface written in Common Lisp for the OpenMusic platform. We distinguish four parts in this contribution:

1. Migration to C++ and definition of object-oriented structure

In the previous works by Thibault Wafflard and Anton Lamotte, the constraints were defined in Common Lisp and passed to Gecode through the GiL interface. This is a cumbersome solution, so the present work defines all constraints in C++ so they are directly used by Gecode, including all the constraints defined in the previous works as well as the new four-voice constraints of the present work.

The C++ translation uses an object-oriented structure in order to simplify the code. The Fux theory defines counterpoint for two, three and four voices, and each voice can be defined in five species. Many constraints are applicable, either unchanged or with small changes, depending on the number of voices and which species is used for each voice. The object-oriented structure uses both inheritance and functional abstraction to simplify the C++ code.

2. Extension to four voices, for all species

This contribution encompasses both theoretical and practical advancements. Theoretically, we translate the third book of *Gradus ad Parnassum*, which focuses on four-part composition, from Fux’s natural language dialogues into a set of rigorous mathematical constraints. Practically, we extend the new C++ implementation of FuxCP to incorporate these newly formalized rules, ensuring the system can effectively handle four-voice counterpoint across all species. We also extend the Lisp-based OpenMusic interface, enabling composers to generate four-part counterpoint compositions.

3. Unified theoretical framework of formal rules

A comprehensive theoretical contribution of this thesis is the creation of a unified list of formal rules that merges the separate contributions of Wafflard, Lamotte, and our own work. This effort involved revisiting and consolidating the various sets of rules, removing redundancies, and ensuring consistency across different contexts. The result is a generalized constraint-based mathematical formalization of Fux’s theory of counterpoint.

4. Testing framework to validate the implementation

Testing is difficult because FuxCP intentionally allows **many** solutions; it enforces counterpoint but does not determine the solution. All possible counterpoints that satisfy the Fux theory can potentially be generated by FuxCP. This means that it is impossible to test all possible solutions, but we have implemented tests to verify that the solutions given by Fux in his treatise are also given by FuxCP.

In summary, this thesis contributes to the ongoing effort to merge the rich theoretical framework of counterpoint with modern computational techniques, providing composers with a powerful, transparent, and flexible tool. By doing so, we continue the journey of exploring and formalizing the intricate beauty of counterpoint, ensuring that Fux’s legacy remains vibrant in the age of artificial intelligence and computational creativity.

1.6 Roadmap for Reading This Thesis

This thesis is structured to guide the reader through the theoretical foundations, practical implementations, and evaluation of our work on formalizing Fux’s four-voice

counterpoint using constraint programming. Below is an overview of the thesis structure :

Chapters 1 to 4 lay the essential context and theoretical foundations. Chapter 2 introduces key tools and software, including the new method for linking Lisp and C++ in section 2.3, which is crucial even for readers familiar with earlier works. Chapter 3 covers the methodology, with both theoretical and practical sections. Chapter 4 presents the fundamental concepts and variables used in the formalization, including efforts to generalize and unify theoretical frameworks. The variables and concepts are widely similar to those introduced by Wafflard and extended by Lamotte, but readers familiar with those works are still encouraged to read the opening paragraphs of section 4.2 as well as subsection 4.2.2 to understand the generalization efforts made.

Chapters 5 to 7 focus on the main contributions of this thesis. Chapter 5 details the formalization of four-voice counterpoint rules, and Chapter 6 then provides a unified set of rules, and, importantly, details the specific contexts in which each rule should apply. Chapter 7 then describes the program architecture, demonstrating how theoretical concepts are implemented in the software.

Chapters 8 to 10 evaluate the implementation and reflect on the work. Chapter 8 starts by describing the testing framework, then assesses the effectiveness of FuxCP through generated examples. Chapter 9 then discusses challenges, limitations, and future improvements. Finally, Chapter 10, the conclusion, summarizes the contributions and suggests directions for future research.

The User Guide for FuxCP is included in Appendix B. The full code of the project can be found on GitHub at <https://github.com/diegodepatoul/FuxCP4>, and is included in this manuscript in Appendix C. In order to understand the current functionalities and limitations of FuxCP, please read this thesis before experimenting with the solver.

In summary, after three more introductory chapters, the theoretical part of the work is detailed in Chapters 4, 5 and 6; the implementation is presented in Chapter 7 and evaluated in Chapter 8; and the last two chapters conclude the thesis. Readers familiar with Wafflard's and Lamotte's works can skim Chapters 2 and 4 but should review the sections that introduce new or significantly modified concepts. These sections are mentioned above (2.3, the opening paragraphs of 4.2, and 4.2.2). When reading mathematical constraints, Appendix A is useful as it contains the definitions of constants used.

Chapter 2

Tools and Software Used

2.1 Constraint Programming

2.1.1 Basic overview of Constraint Programming

Constraint Programming (CP) is a powerful paradigm for solving combinatorial search problems [21], where the goal is to find values for problem variables within a given set of constraints.

Formally, a Constraint Satisfaction Problem (CSP) is defined by a triplet consisting of :

- a set of **decision variables**,
- the **domains** of these variables (the domain of a variable is the set of values that it could take), and
- a set of **constraints**, logical relations restricting the combinations of values that the variables can take.

Here, we can already see how prone this paradigm is to formalizing Fux's counterpoint. The notes of the counterpoints are formalized as variables, whose domains are the MIDI values that they can take, while Fux's rules are formalized as constraints acting on these variables (notes). More detail on this aspect is given in Chapter 4.

Once the problem is defined, the solver can start searching for solutions. This is done using **branching**, which selects a variable to constrain in order to delve deeper into the search tree. A solution is found when all of the variables are set to a value of their domain, and all constraints are satisfied by this set of values.

Costs and Preferences

Some of Fux's rules are defined as **preferences**, not strict constraints. In fact, in general, the more counterpoint voices we are composing, the more strict rules will be relaxed as preferences, else there would be many cases where no solution would be feasible.

These preferences are implemented as **costs**. Once costs are introduced, the CSP is not merely a satisfaction problem anymore: it becomes an optimization problem. In such a problem, when multiple solutions exist, the best solution is the one that has the lowest cost.

2.1.2 Gecode



Figure 2.1: Gecode logo [4]

Fortunately, there are many libraries available for developing constraint-based systems, so there is no need to implement a CSP solver from scratch. The toolkit that was chosen for this thesis is Gecode, an open source C++ environment distributed under the MIT license [4]. Gecode's performance, accessibility, and extensive documentation made it an excellent choice for supporting the CSP solving aspect of this project.

2.2 OpenMusic



Figure 2.2: OpenMusic Logo [3]

ircam
Centre
Pompidou



Figure 2.3: IRCAM Logo [22]

As said previously, FuxCP, beyond being theoretically interesting as it mathematically formalizes *Gradus ad Parnassum*, is ultimately meant as a tool for composers. The goal is to make it accessible to all composers, regardless if they have a programming background or not. This is why FuxCP is implemented as an external library to OpenMusic [3], a composition environment made by IRCAM [22], the *Institut de Recherche et Coordination Acoustique/Musique* in Paris. In their own words,

"OpenMusic (OM) is a visual programming language designed for music composition. Visual programs (called "patches") can be created by connecting graphical modules together, corresponding to musical objects or data processing functions.

OM offers you a complete freedom in programming, including text programming via the Common Lisp language. A large numbers of specialized projects and libraries are available for specific applications (mathematical tools, sound processing and synthesis, etc.)" [23]

FuxCP is thus one of these specialized libraries. This "complete freedom in programming" made OM an ideal choice for this project. For an explanation of how to install OpenMusic and use FuxCP, as well as an explanation of the FuxCP interface, please refer to the User Guide in Appendix B of this thesis.

2.3 GiL and CFFI

With the selection of the two primary software components — Gecode, a state-of-the-art constraint programming library, and OpenMusic, an innovative visual composition environment — a key challenge arises: OM is an extension of Common Lisp, whereas Gecode is implemented in C++. A connection between the two must be established to allow the definition and execution of a constraint solver from the Common Lisp environment. The solution to this challenge lies in GiL.

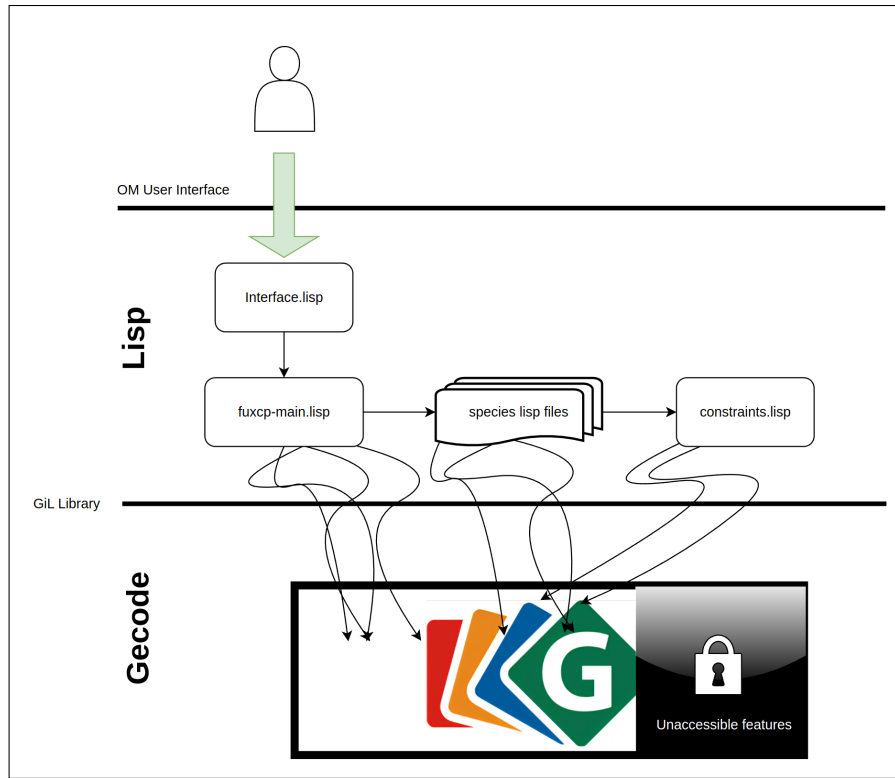
GiL (Gecode interface Lisp) [5] is a tool developed by Baptiste Lapière and extended by Damien Sprockeels which enables Lisp programs to call Gecode functions via the Common Foreign Function Interface (CFFI) [24]. GiL is available in two distinct versions: GiL Framework and GiL Library. The Library version wraps Gecode functions in Lisp, enabling direct definition and execution of the CSP from within Lisp. In contrast, the Framework version offers a lighter interface, where the problem creation is initiated from Lisp, but all subsequent creation processes are handled directly in C++.

This marks the key distinction between the previous versions of FuxCP and the new approach implemented in this thesis. Wafflard and Lamotte used GiL Library, which means that the project was completely based in Lisp, making use of the wrapped functions provided by GiL to create variables, post constraints, and launch the search for solutions directly from Lisp. Initially, this approach appeared promising, given that the GiL Library had already proven effective for defining CSPs from Lisp. However, it soon became evident that this strategy was inadequate for a project as ambitious as FuxCP.

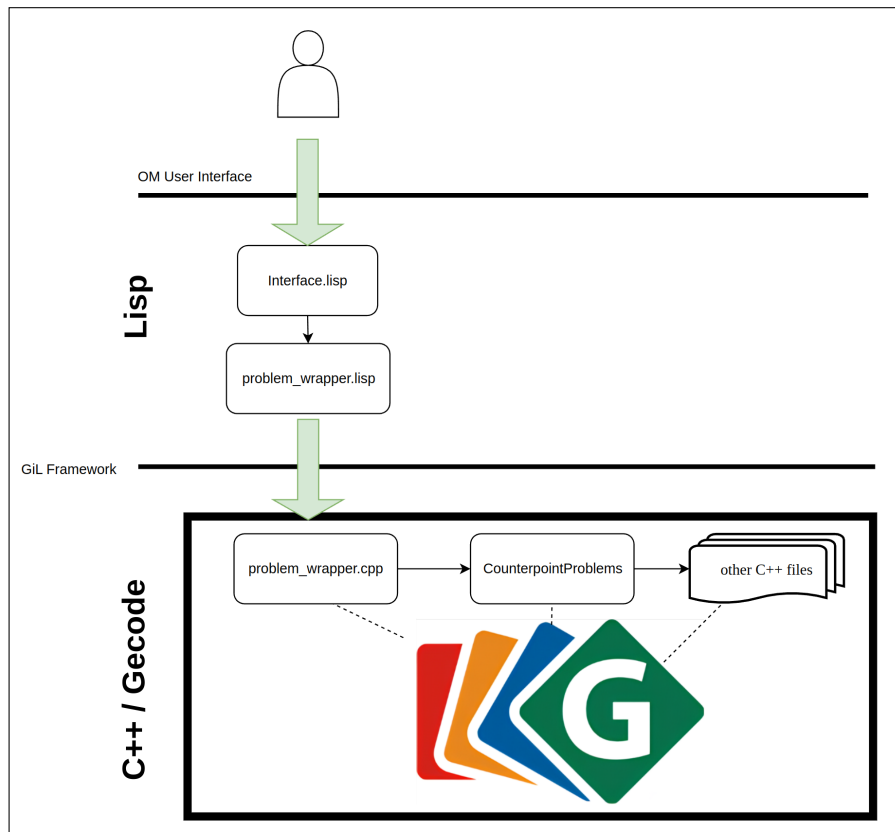
The limitations inherent in a Lisp-based FuxCP include:

- **Difficulty managing objects:** Gecode, being rooted in C++, an object-oriented language, extensively utilizes objects in its design — spaces, CSPs, solvers, and more are implemented as classes. However, Lisp’s lack of inherent object-oriented support makes working with Gecode exceedingly cumbersome. Furthermore, as will be demonstrated later in this thesis, Fux’s organization of species and rules lends itself naturally to an inheritance-based architecture, a paradigm not supported in Lisp.
- **Verbosity and reduced code readability** are natural consequences of the aforementioned limitations, and an intrinsic disadvantage of the Lisp language, especially when used in a mathematical context.
- **Lack of access to some of Gecode’s features:** The GiL Library supports a large number of Gecode functions, but the Gecode libraries are extremely rich and are simply not meant to be coded in Lisp. Therefore, some of the more complex features are inaccessible from the foreign interface.

These limitations result in cumbersome programming and suboptimal utilization of Gecode, ultimately leading to inefficient software. As a result, the choice was made at the start of the work on this thesis to migrate FuxCP to C++, which also implies a migration from the GiL Library to the Framework. The following diagrams visualize this migration, emphasizing the more compact and centralized interface that the Framework provides.



(a) **Interfaces when using GiL Library.** A total of 107 different CFFI functions are called 1,282 times across 10 different files. Some features remain inaccessible.



(b) **Interfaces when using GiL Framework.** The new design reduces the number of CFFI functions to 11, each called only once, from a single file. This streamlines the process and increases flexibility, allowing easier integration with other software. The codebase has been moved to C++, where Gecode is natively supported, providing full access to Gecode's capabilities and significantly enhancing ease of programming.

Figure 2.4: Comparison of GiL Library and GiL Framework interfaces.

Chapter 3

Methodology for Formalizing Fux's Work

Formalizing Fux's work using constraint programming is a meticulous task. In such a complex CSP (Constraint Satisfaction Problem), even small errors in constraints or variable declarations can lead to the solver either missing valid solutions or finding solutions that do not conform to Fux's rules. Therefore, a rigorous methodology is essential throughout the formalization, coding, and testing processes.

This chapter outlines the methodology used for the two main objectives of this thesis: generalizing and extending Fux's rules and variables to accommodate four-voice counterpoint, and migrating from Lisp to C++ while incorporating these theoretical advancements. Testing using Fux's examples is also discussed.

3.1 Theoretical Work

3.1.1 Generalizing Existing Rules and Variables

The first step in the theoretical work was to systematically review the existing rules and variables introduced by Wafflard and Lamotte. The goal was to determine whether each rule is applicable in contexts beyond the one in which it was originally introduced, and if so, whether the current mathematical formalization remains valid across all these contexts. It is important to note that when Fux gives a rule, he rarely explicitly states its exact scope, so determining in which configurations the rule should apply sometimes comes down to judgement and musical logic.

A good example of a rule that was introduced in a two-voice context but should apply in all contexts, and whose mathematical formalization had to be adapted, is given in section 4.2.2.

In summary, the methodology used to transfer the rules of two and three voice composition into our complete mathematical formalization had to be done by rigorously following these steps. For each rule, the following procedure was followed :

1. Understanding the true meaning of the rule as intended by Fux. These are often well explained in the thesis in which they had been introduced, but sometimes going back to the original source, *Gradus ad Parnassum*, was necessary.
2. Checking whether the rule is already covered by another rule. Fux is sometimes redundant in his work, restating some of the important rules in other chapters, although the general rule had already been given. In some cases, one general rule is formalized multiple times across the different previous thesis. In these cases, these multiple formalizations were merged into one generalized rule.
3. Determining the scope of the rule. Some rules introduced in two voices are only meant to be applied when composing one counterpoint, and others are general

rules to be adopted in any configuration ¹. The same logic applies for rules introduced by Lamotte for three-voice counterpoint. To determine the scope, two scenarios were possible.

- (a) If Fux was clear about its scope from the beginning, or explicitly stated that the rule still applied for broader configurations, then determining the scope was easy and we could be confident that it accurately captured Fux’s intentions. However, this was rarely the case.
 - (b) Unfortunately, most of the time, the scope is made clear neither by Fux nor by the thesis that first formalized it. In that case, determining the scope of the rule came down to judgement and common sense. Fortunately, Fux provides many examples in every configuration, with which we can confirm whether or not the rule applies.
4. Determining whether the existing mathematical formalization accurately captures the intended meaning of the rule in all of the configurations in which it applies. For many rules, the configuration had to be adapted for it to apply in all relevant contexts, not only the context in which it was introduced.
 5. Verifying that this formalization of the rule does not clash with any of the other rules.

3.1.2 Adding New Rules for Four-Voice Composition

The process for adding new rules for four voice counterpoint is similar to the one described above, except for the fact that there exists no previous formalization of Fux’s theory in these configurations. The individual rules had to be extracted from the dialogue written by Fux in *Gradus ad Parnassum*, put into a simplified and concise English formulation detailing both the rule and its scope, then formalized mathematically.

This process is described in Chapter 5, and the complete mathematical formalization of all rules is given in Chapter 6.

3.2 Practical Work

3.2.1 Migrating from Lisp to C++

The migration from a Lisp-based to a C++-based implementation was a significant undertaking and was approached in a phased manner.

The initial phase involved translating the Lisp code for the first two species into C++. This translation preserved the original structure as much as possible to ensure functional equivalence. Once a basic version for the first two species was operational and producing accurate results, the next phase focused on extending this code to support four-voice compositions.

As the extension progressed, it became evident that the Lisp-like structure was insufficient for handling the complexities of four-voice counterpoint. Therefore, an object-oriented structure was adopted, leveraging C++’s capabilities. This required exploring various inheritance structures to best represent Fux’s rules. After testing several options, a balanced structure was chosen that made effective use of object-oriented programming without introducing unnecessary complexity. The final structure is detailed in Chapter 7.

¹A **configuration** is determined by the number of voices, the species of each counterpoint, and its relative pitch range.

It is important to note that while the CSP building part of the code was migrated to C++, the OpenMusic interface remains in Lisp. Therefore, a new link between Lisp and C++ had to be designed, building upon the GiL Framework.

3.2.2 Incorporating Theoretical Changes and New Rules

With the migration complete, the next step was to integrate the theoretical advancements into the C++ implementation. This involved incorporating the generalized rules and the new rules for four-voice composition into the code.

The integration process required careful attention to ensure that the changes did not disrupt the existing functionality while accurately reflecting the theoretical improvements. Each new rule was coded and then tested to verify that it worked correctly in all applicable contexts.

3.2.3 Testing with Fux's Examples

Additionally to testing the individual constraints during the coding process, testing the implementation against Fux's examples from *Gradus ad Parnassum* was an essential part of the validation process. Fux provides numerous examples that cover a wide range of cases across different species, voice numbers, and relative pitches of the *cantus firmus*.

To verify that the CSP correctly represents Fux's rules and can generate the same music, each example was encoded as a set of constraints, with each note variable set to match the corresponding MIDI value in the example. The solver was then run, and two outcomes were possible:

- If the search returned one solution, of course matching Fux's example exactly, it confirmed that the example was a valid solution to the CSP, indicating that no constraints were violated.
- If the search returned no solutions, this indicated that one or more constraints or variable declarations were preventing the example from being accepted. In such cases, constraints were systematically deactivated until the example was feasible, and then reactivated one by one to identify the source of the problem.

This iterative testing process was crucial for ensuring that the final implementation adhered to Fux's intentions.

Chapter 4

Concepts and Variables Used in the Formalization

It is now time to delve into the core of this work : the formalization of Fux's rules for counterpoint. It is important to note that we present not only an **extension** to four voices of the previous formalizations of two and three voice counterpoint, but also a **generalization** and **harmonization** of the complete formalization of Fux's work. The aim is to have *one* set of concepts and variables used throughout the formalization, regardless of the number of voices or the species configuration of the specific composition. Therefore, there has been a revisitation of some rules that used concepts specific to a certain case. The formalization of these rules were modified so that all rules use the same generalized variables and concepts.

4.1 Voices, Parts, and Strata

Before defining the specific variables used in the formalization, we must first lay an important foundation : defining the concepts of voices, parts, and strata. These concepts were defined by A. Lamotte to accommodate the increasing complexity of three-part counterpoint compared to T. Wafflard's previous formalization of two-part counterpoint. The definitions and interactions between these concepts remain the same in this thesis, so this section provides a sufficient overview without delving into as much detail and justification as Lamotte.

The simplest concept to define is the concept of **parts**, because they represent the concrete reality of the music. A part corresponds to what a person sings or what an instrument plays. Each musical staff in the composition corresponds to a part. The parts in a counterpoint composition are the *cantus firmus* and the counterpoint(s).

Where parts represent a concrete reality, **strata**, on the other hand, are an abstract concept that is useful only in the mathematical formalization of Fux's rules. Strata are used to capture, at any given measure, which voice is sounding the lowest, highest, or any "ranking" in between. This implies that there are as many strata as there are parts. If the parts all stay in their voice range and never cross each other, the strata are identical to the parts. However, the crossing of parts is allowed, which is why the concept of strata is useful.

In Lamotte's own words,

"[The concept of strata] provides a way of analyzing and understanding the distribution of pitches across different parts, allowing more complex rules to be established. For example, it would now be possible to establish a rule between the notes of the *cantus firmus* and the highest sounding notes (no matter which part they come from)." Lamotte [2, p.12]

This possibility of allowing more complex rules is necessary because Fux defines many rules as a relation between parts and the "lowest voice", and clearly states that this

could be any part at any given moment. The lowest voice could come from the bass part at one measure and the tenor part at the next; in both situations, the lowest sounding note is the one that should be considered.

The last concept, **voices**, are just a general concept that includes parts and strata. When we talk about voices, we refer to parts and strata alike.

Defining the strata : a bijection with the parts at every measure

To give a more precise definition of the strata, it is necessary to consider what happens when two parts play the same notes. This is important because, as will be explained later on, we cannot have two different parts both be the lowest stratum at a given measure. Therefore, there must be a bijection between parts and strata at every measure. In other words, for any given measure, each stratum corresponds to a single part, and vice versa. As a convention, when multiple parts play the same note, the lowest of the strata is mapped in the bijection with the first of the parts in this order : $[cf, cp_1, cp_2, cp_3]$. For example, if the *cantus firmus* and the first counterpoint play the same note and it is the lowest note of the composition, the lowest stratum will be the *cantus firmus* at that measure, and the second stratum (also called first upper stratum, or *upper_1*) will be the *cp_1*.

Let us now give the mathematical definition of the notes of the α -th stratum (when $\alpha = 0$, it is the lowest stratum). See the following section for mathematical notations.

$$\forall \alpha \in [0, 3] \quad \forall j \in [0, m - 1] : N(l[\alpha])[0, j] = X[\alpha]$$

where $X = \text{sorted}([N(cf)[0, j], N(cp_1)[0, j], N(cp_2)[0, j], N(cp_3)[0, j]])$ (4.1)
and $l = [a, b, c, d]$ to keep the right stratum notation.

The other notes of the measure are the notes of the part that was linked to the stratum using the first note. That way, strata do not include notes of multiple different parts in a single measure. The *sorted* function sorts notes from lowest to highest, breaking ties by prioritizing the *cantus firmus* then the counterpoints in their order, as was explained above.

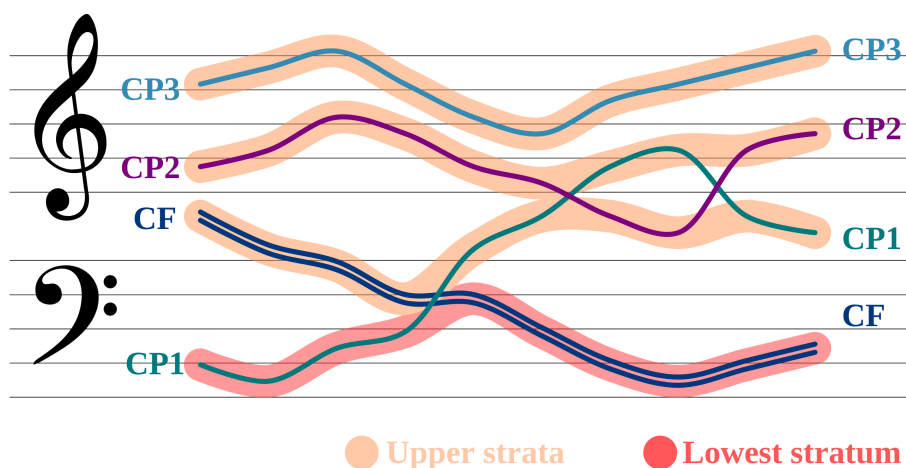


Figure 4.1: Visualisation of parts and strata

4.2 Variables and Arrays

This section, building upon the essential concepts presented in the previous section, presents the variables used in the formalization. We use the same notations and logic as the previous formalizations by T. Wafflard and A. Lamotte, while trying to generalize and simplify the final set of variables as much as possible, giving us a more unified and straightforward final formalization.

It is important to note that there is a distinction between the mathematical formalization we present here and the computer implementation we will present later in this thesis. The goal of this complete mathematical formalization is to provide a full constraint-based mathematical formalization of *Gradus ad Parnassum*, which can then serve as a base for a computer implementation. As will be detailed in another chapter, the computer formalization we implemented is based in C++, and widely uses the concept of inheritance. This induces the creation of new helper arrays, which are useless to define in the general mathematical formalization, and conversely, some variables defined here are not needed in our particular computer implementation. However, all of the important arrays (N, H, M, P...) are of course present in the code, and the code itself is extremely close to the math.

Since a CSP is theoretically defined by the sets of variables, domains and constraints, the CSP's defined by the mathematical formalization and the C++ implementation are slightly different because the set of variables is not perfectly identical. However, the set of solutions (a MIDI value attributed to each counterpoint note) is the same, because the variables that differ are only different formalizations of the same mathematical reality.

The decision to give a mathematical formalization that doesn't perfectly represent the code was taken because, as will be explained in the chapter about the C++ implementation, tradeoffs in the code design had to be found in multiple contexts. One example of this is the tradeoff between an over-hierarchicalisation of the structure and a structure without hierarchy. The final implementation is what is currently believed to be the right tradeoff in multiple dilemmas, but as the software evolves with the next theses, the optimal tradeoff might evolve too, which implies that the final set of variables evolves as well. The goal is that the mathematical formalization can withstand the test of time, continuing to give a correct CSP that accurately represents Fux's work, with a set of solutions that completely represent the style of counterpoint according to Fux.

With all of that said, let us move on to the definition of the variables and constants. The first thing to note is that the CSP we are defining is an integer problem, because note pitches are defined by MIDI values (which are integers ranging between 0 and 127, which each semitone having a value of 1). As said before, a solution to our problem consists of a pitch attribution to every note of every generated counterpoint part. Therefore, the vast majority of variables are integer variables, representing either the MIDI values of the note pitches themselves or intervals between different note pitches. Since notes take potentially different values for every beat of the composition, the variables are mainly integer arrays.

Note name	A0#	C1#	D1#	F1#	A1#	G1#	C2#	D2#	A2#	F2#	C3#	D3#	F3#	A3#	C4#	D4#	F4#	A4#	G4#	F4#	C5#	D5#	F5#	A5#	G5#	F5#	C6#	D6#	F6#	C7#	D7#	F7#	A7#	C8																																																					
Midi number	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108
Note name	A0	B0	C1	D1	E1	F1	G1	A1	B1	C2	D2	E2	F2	G2	A2	B2	C3	D3	E3	F3	G3	A3	B3	C4	D4	E4	F4	G4	A4	B4	C5	D5	E5	F5	G5	A5	B5	C6	D6	E6	F6	G6	A6	B6	C7	D7	E7	F7	G7	A7	B7	C8																																			

Figure 4.2: 88 notes classical keyboard note names and MIDI values [25]

4.2.1 Variables

The single most important set of variables is N , which holds the MIDI values of the notes of the composition. To know which beat we are talking about, we address the variable in a computer notation. For example, $N[i, j]$ means "the note on the i -th beat of the j -th measure". This implies that i ranges from 0 to 3 and that j ranges from 0 to the number of measures (which is written as m). We can now distinguish which particular beat we are looking at, but in three and four part composition, there are multiple different counterpoints generated, which means that there are multiple variable notes at any given beat. Therefore, we define theoretical functions to determine which voice's variable we are talking about : $N(v)$ represents the notes of the voice v .

- cf - for linking the variable to the *cantus firmus* ($N(cf)$ refers to the notes of the *cantus firmus*).
- cp_1 , cp_2 , or cp_3 - for linking the variable to the first, second, or third counterpoint, respectively.
- a , b , c , or d - for linking the variable to the lowest, second, third, or uppermost stratum, respectively. For example, $N(a)[2, 5]$ refers to the second note of the fifth measure of the lowest stratum.

When a variable is not explicitly linked to a voice, it is implied that the relation expressed is true for all parts :

$$N \equiv \forall p \in \{cf, cp_1, cp_2, cp_3\} : N(p) \quad (4.2)$$

Theoretically, all of the rules could be written mathematically using just the N arrays. All of the constraints are some way of limiting the simultaneous values that notes can take. However, the rules often make use of redundant concepts, and it would be foolish not to use them to simplify the formalization. The majority of the rules can be classified into three categories, each using one of the main concepts : Harmony, Melody, and Motion.

- **Harmonic rules** deal with the "vertical" part of composition. At any given time, the notes sounding together are constraint by rules, that mostly regulate the intervals between these notes.
- **Melodic rules** deal with the "horizontal" part of composition. These rules apply themselves to individual parts, governing which succession of notes is allowed or not.
- **Motion rules** deal with the relative movement of the parts. There are three types of motion: **direct**, when two parts move in the same direction, **contrary** when two parts move in opposite directions, and **oblique** when one part moves and the other remains stationary.

To make use of these concepts, we define for each one a set of variables (see Appendix A for the definitions of the constants). Note that these arrays are **derived arrays**, constructed from the main arrays $\mathbf{N}$ containing the note pitches.

- $H(v_1, v_2)$ - the harmonic intervals between voice v_1 and voice v_2 . See table 4.1 for the definitions of the different types of intervals.

$$\begin{aligned} \forall v_1, v_2 \in \{cf, cp_1, cp_2, cp_3, a, b, c, d\}, \quad \forall i \in \mathcal{B}(v_1), \quad \forall j \in [0, m) : \\ H(v_1, v_2)[i, j] = |N(v_1)[i, j] - N(v_2)[0, j]| \bmod 12 \end{aligned} \quad (4.3)$$

Notice that $H(v_1, v_2)[i, j]$ represents the intervals between the i th beat of the j th measure of voice v_1 and the **first** beat of the j th measure of voice v_2 . This is to simplify notations, by just defining that when focusing on a part, the only notes that matter from the other parts are the first notes of each measure. This is a general idea that applies across the formalization.

Also notice that there is no more H_{abs} (the harmonic interval without the modulo). Since it was only used in one rule (because for the vast majority of rules we just need to know the nature of the interval, not the precise number of semitones separating two notes), it was decided that defining a whole array was superfluous, counterproductive and inefficient. Where the precise number of semitones of a harmonic interval is needed, we will just calculate the difference manually.

If v_2 is not specified, it is equal to a by default. In other words, $H(v_1)$ represents the intervals between the voice v_1 and the lowest stratum: $H(v_1) \equiv H(v_1, a)$. This goes to show how important the harmonic interval with respect to the lowest stratum is in musical composition. In fact, it is so important that a boolean variable array, $\mathbf{IsCons}(v)$ is defined that represents whether a voice is consonant with the lowest stratum at a given index :

$$\begin{aligned} \forall i \in \mathcal{B}, \forall j \in [0, m) \\ IsCons(v)[i, j]_{(all, p, imp)} = \begin{cases} \top & \text{if } H(v)[i, j] \in Cons_{(all, p, imp)} \\ \perp & \text{otherwise} \end{cases} \end{aligned} \quad (4.4)$$

- $\mathbf{M}_{brut}(v)$ - the melodic intervals of the voice v .

– $\mathbf{M}_{brut}^x(p)$ (i.e. when related to a part) is defined as :

$$\begin{aligned} \forall x \in \{1, 2\}, \forall i \in \mathcal{B}, \forall j \in [0, m - x) : \\ M_{brut}^x[i, j] = \begin{cases} N[i + xd, j] - N[i, j] & \text{if } i + xd < 4 \\ N[(i + xd) \bmod 4, j + 1] - N[i, j] & \text{if } i + xd \geq 4 \end{cases} \quad (4.5) \\ M^x[i, j] = |M_{brut}^x[i, j]| \\ \text{where } M_{brut}^x[i, j] \in [-12, 12], \quad M^x[i, j] \in [0, 11] \end{aligned}$$

In practice, M^2 is very rarely used, so it is not defined as a variable array but manually calculated whenever needed. It was decided to keep it in the formalization because of the elegance and insight on the general concept of melodic intervals that this general formalization gives. By default, $M \equiv M^1$, thus, $M \equiv \forall p \in \{cf, cp_1, cp_2, cp_3\} : M^1(p)$.

– $M_{brut}(s)$ (i.e. when related to a stratum) :

$\forall j \in [0, m - 1) :$

$$M_{brut}(a)[j] = \begin{cases} M_{brut}(cf)[0][j] & \text{if } A(cf)[j + 1] \\ M_{brut}(cp_1)[\max(\mathcal{B}(cp_1))][j] & \text{if } A(cp_1)[j + 1] \\ M_{brut}(cp_2)[\max(\mathcal{B}(cp_2))][j] & \text{if } A(cp_2)[j + 1] \\ M_{brut}(cp_3)[\max(\mathcal{B}(cp_3))][j] & \text{if } A(cp_3)[j + 1] \end{cases} \quad (4.6)$$

where A is the boolean array representing whether or not a part is the lowest stratum at a given measure (see definition 4.8).

To fully understand this definition of the melodic interval of a stratum and why it is given, please refer to A. Lamotte's thesis, where it is fully and lengthily explained. In short, it is defined as it is because its purpose is to compute motions with respect to the lowest stratum.

- $P(p)$ - the motions of the part p .

By definition, 0 represents contrary motion, 1 represents oblique motion, and 2 represents direct motion (all defined with respect to the lowest stratum). If motion is computed for a part that is the lowest stratum at a given measure, the motion is defined to be -1 , which acts as a N/A value.

$$\begin{aligned} \forall p \in \{cf, cp_1, cp_2, cp_3\}, \quad \forall x \in \{1, 2\}, \quad \forall i \in B, \quad \forall j \in [0, m - 1), \quad x := b - i \\ motion(p)[i, j] = \begin{cases} 0 & \text{if } (M_{brut}^x(p)[i, j] > 0 > M(a)_{brut}[j]) \\ & \quad \vee (M_{brut}^x(p)[i, j] < 0 < M(a)_{brut}[j]) \\ 1 & \text{if } M_{brut}^x(p)[i, j] = 0 \oplus M(a)_{brut}[j] = 0 \\ 2 & \text{if } (M_{brut}^x(p)[i, j] > 0 \wedge M(a)_{brut}[j] > 0) \\ & \quad \vee (M_{brut}^x(p)[i, j] < 0 \wedge M(a)_{brut}[j] < 0) \\ & \quad \vee (M_{brut}^x(p)[i, j] = 0 = M(a)_{brut}[j]) \end{cases} \\ P(p)[i, j] = \begin{cases} -1 & \text{if } A(p)[j] \\ motion(p)[i, j] & \text{if } \neg A(p)[j] \end{cases} \end{aligned} \quad (4.7)$$

$x := b - i$ represents the fact that the motion is obtained between the current note and the first note of the next measure. For example, with quarter notes, the gap between the third note and the first note of the next measure is defined as: $b = 4, i = 2$ and $x = 4 - 2 = 2$. [1]

One more helper array is defined, which is mostly useful for calculating motions, because we need **one** stratum to be the lowest stratum at any given measure, even if multiple notes sound in unison as the lowest (remember that there is a bijection between parts and strata). This array is $A(p)$, the boolean array representing whether or not part p is the lowest stratum or not :

$\forall j \in [0, m):$

$$\begin{aligned}
A(cf)[j] &= \begin{cases} \top & \text{if } N(cf)[0, j] = N(a)[0, j] \\ \perp & \text{else} \end{cases} \\
A(cp_1)[j] &= \begin{cases} \top & \text{if } (N(cp_1)[0, j] = N(a)[0, j]) \wedge \neg A(cf)[j] \\ \perp & \text{else} \end{cases} \\
A(cp_2)[j] &= \begin{cases} \top & \text{if } (N(cp_2)[0, j] = N(a)[0, j]) \wedge \neg A(cf)[j] \wedge \neg A(cp_1)[j] \\ \perp & \text{else} \end{cases} \\
A(cp_3)[j] &= \begin{cases} \top & \text{if } \neg A(cf)[j] \wedge \neg A(cp_1)[j] \wedge \neg A(cp_2)[j] \\ \perp & \text{else} \end{cases}
\end{aligned} \tag{4.8}$$

About the Fifth Species

Before defining the derived arrays (H, M, P), we had said that theoretically, the formalization could be entirely based upon the N array. This was true for the four first species, because the nature of the notes stay constant, so we only need the pitches. In the fifth species, this is not true anymore, as the natures of the notes can vary across the composition. Consequently, we need a new array describing the nature of each note, in addition to the already existing array N which contains the pitches.

We define the S array :

$$\begin{aligned}
&\forall \rho \in positions(m) \\
S[\rho] &= \begin{cases} 0 & \text{if } N[\rho] \text{ is not constrained by any species} \\ 1 & \text{if } N[\rho] \text{ is constrained by the first species} \\ 2 & \text{if } N[\rho] \text{ is constrained by the second species} \\ 3 & \text{if } N[\rho] \text{ is constrained by the third species} \\ 4 & \text{if } N[\rho] \text{ is constrained by the fourth species} \end{cases}
\end{aligned} \tag{4.9}$$

Costs : Formalizing Preferences

In Fux's work, some rules are strict, like the use of consonances on the downbeat, and some rules are just indications, or preferences. In fact, as the number of voices increases, some rules that were previously strict are relaxed into being preferences to accommodate the complexity of composing multiple voices. All rules cannot be met at the same time.

Preferences, in constraint programming, are implemented as costs. Costs are variables that serve to evaluate the quality of a solution. When multiple solutions are found, the solver uses costs to rank the quality of these solutions, and the branch and bound algorithm is designed to only find solutions that are better than the ones previously found during the search.

FuxCP has multiple different costs, implemented as variables and variable arrays. Constraints are posted to link costs to the variables of the solution, checking whether the preference is met and setting the cost accordingly.

With the implementation of costs, the solver now solves a minimization problem, instead of a mere satisfaction problem.

In the mathematical formalization of the constraints in this thesis, costs are denoted with $\mathcal{C}$. It is important to note that most costs are in fact variable arrays, having one cost per measure. The constraints set the variable at index j to 0 if the preference is

respected, and to a positive integer value otherwise (the value itself depends on the user preferences). The multiple cost variables that are defined ($\mathcal{C}_{prefer-harmonic-triad}$, $\mathcal{C}_{succ_p_cons}$ etc.) are all part of the set of all costs, $\mathcal{C}$. Chapter 8 talks about how these costs are used to guide the solver towards the best solution for a given problem.

4.2.2 Obsolete Variable Array : IsCfB

This boolean array, introduced by T. Wafflard, was defined as true when the *cantus firmus* is lower than the considered part and false otherwise:

$$\begin{aligned} \forall p \in \{cp_1, cp_2\} \quad \forall i \in \mathcal{B}(p), \forall j \in [0, m) \\ IsCfB(p)[i, j] = \begin{cases} \top & \text{if } N(p)[i, j] \geq N(cf)[0, j] \\ \perp & \text{otherwise} \end{cases} \end{aligned} \quad (4.10)$$

It was kept by A. Lamotte in his set of variables because the goal was to make his formalization compatible with all the rules introduced previously by T. Wafflard. However, this was a mistake because in all of the rules in which it was used in Wafflard's thesis, its goal is ultimately to know which part is the lowest stratum, not if a part is above or below the *cantus firmus*. These two are of course equivalent when dealing with two voices, but not anymore with three voices.

A good illustration of the problem that this generalization induces is rule **1.H4** - *the key tone is tuned according to the first note of the cantus firmus*. In other words, the tonic of the piece is the first note of the *cantus firmus*, and the bass at the first and last measures must be the tonic. This was formalized mathematically by Wafflard as

$$\begin{aligned} \neg IsCfB[0, 0] &\implies H[0, 0] = 0 \\ \neg IsCfB[0, m-1] &\implies H[0, m-1] = 0 \end{aligned} \quad (4.11)$$

which means *if the cantus firmus is above the considered part, the harmonic interval in the first and last measures should be zero*. This is problematic because it would also apply to the middle voice if the *cantus firmus* was the uppermost part, even though this rule is meant to apply only to the bass. Therefore, the formalization of the rule was adapted as such :

$$\begin{aligned} N(a)[0, 0] \bmod 12 &= N(cf)[0, 0] \bmod 12 \\ N(a)[0, m-1] \bmod 12 &= N(cf)[0, 0] \bmod 12 \end{aligned} \quad (4.12)$$

This new formalization correctly captures the true meaning of the rule. This example, far from being isolated, shows that building on previous work is a meticulous task, and every rule must be fully understood down to the original intention of Fux, beyond the mathematical formalization, before coding it into a solver.

After thoroughly revisiting each rule and ensuring that their mathematical formalizations accurately capture their intrinsic meaning in all applicable configurations (as outlined in Chapter 6), it became clear that the **IsCfB** variable array was no longer relevant in any of the rules. This observation reflects a fundamental reality: whether the *cantus firmus* is above or below a given counterpoint has no bearing on the beauty of the music. The concepts of *cantus firmus* and counterpoint exist solely for the composer. Once the piece is written, it is irrelevant which voice was the *cantus firmus* during composition. The true concept that **IsCfB** was meant to represent when introduced by Wafflard was identifying which part is the bass (or lowest stratum, in the

context of this work). This concept, by contrast, is essential to the beauty of music.

For these reasons, the **IsCfB** variable array has been deprecated and replaced by the more general variable array **A**.

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

Table 4.1: Musical intervals with the number of semitones, examples, and type of interval

Chapter 5

Rules for Four Voice Counterpoint

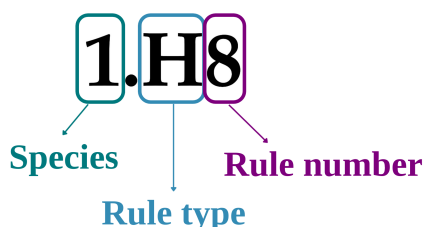
Before diving into the formalization of Fux's rules for four voice counterpoint, a general observation must be made. In the realm of four-voice counterpoint, the complexity and richness of the musical texture significantly increase. Unlike simpler counterpoint where rules can often be strictly adhered to, the intricate interplay between four voices introduces a multitude of considerations that must be balanced. Fux himself acknowledges that while the rules should be followed as much as possible, there are instances where it is simply impossible to adhere to every single rule due to the demands of melody or the constraints of the *cantus firmus*. He states:

"The rules given in the First Book concerning progressions and motion should be followed so far as possible, [...]. I said: so far as possible. For one is sometimes forced to accept a hidden succession of fifths or octaves on account of the requirements of the melody, or of the imitation, or of the limiting effect of the cantus firmus. However, the less one departs from the general rules, the more perfect a four part composition will be." Mann [11, p.109]

Therefore, many of the rules that were strictly applied in two and three voice counterpoint are now **relaxed** into preferences, implemented as costs to be minimized by the solver.

Moreover, some rules are not explicitly stated by Fux, but can be deduced from the implications of explicit rules, or from Fux's examples. Therefore, for each species, we will first detail the rules that are explicitly laid out by Fux, then present the implicit rules of that species.

About rule nomenclature : As we are building upon the sets of rules proposed by T. Wafflard and A. Lamotte, it seemed obvious to keep the same nomenclature as them. Moreover, most of their rules for two and three voices still apply in four voice composition, either strictly or in a relaxed manner. The following diagram shows the logic used for naming rules.



In other words, rule **1.H8** is the eighth harmonic rule of the first species. The letters **G**, **S**, **H**, **M**, and **P** respectively denote General, Structural, Harmonic, Melodic and Motion rules.

Let us now begin reading through the Third Part of *Gradus ad Parnassum*, and extracting the rules that Fux states about four-voice counterpoint composition. For each explicit rule, we will first cite Fux, then rephrase it to make it a clear and concise rule in English (which is rarely the case in Fux's work). Once we have a clear idea of what Fux concretely meant, we will formalize this rule mathematically, using the concepts and variables laid out in the previous chapter. Finally, we will show how the rule is implemented in C++.

5.1 First Species



Figure 5.1: Example of a four-voice counterpoint composition in the first species. Mann [11, p.116]

5.1.1 Explicit Rules

1.H8 [PREF] *The harmonic triad (in its natural order, with the octave) should be used as much as possible.*

"A four part chord will consist of a third, fifth, and octave. Whenever one cannot use the octave [...], one must double the third or, more rarely, the fifth." Mann [11, p.109]

"Any consonance should, if possible, occupy its proper place in the natural order." Mann [11, p.111]

This rule takes into account multiple preferences. 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 (see Figure 5.2). 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. [26]

Joseph.— What is this proper place of the consonances, which you have mentioned?

Aloys.— It is the order that results from the harmonic division of the octave.³ Obviously the fifth is produced first by this division; and by a further division of the fifth, the third is produced. This order should be observed when placing the consonances, except when some special reason—usually the progression to the following measures—is against it. Let me give you an example showing the natural order of the consonances:

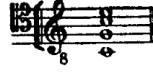


FIG. 161

You see, firstly, the fifth resulting from the harmonic division of the octave; secondly, the octave that we had taken as the basis; and thirdly, the third or rather tenth, which results from the division of the fifth.

Figure 5.2: Explanation of the natural order of consonances in the harmonic triad. Mann [11, p.111]

Throughout the mathematical formalization of this complex constraint, we will use the following convention to make notations lighter and easier to read:

- $H_b[j]$ denotes $H(b)[0, j]$, the harmonic interval between the lowest stratum and the second lowest stratum;
- $H_c[j]$ denotes $H(c)[0, j]$; and
- $H_d[j]$ denotes $H(d)[0, j]$;

where j is an index ranging over $[0, m - 1]$. This shorthand will simplify our expressions without losing clarity.

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.

$$\begin{aligned}
 &\forall j \in [0, m - 1]: \\
 &\quad (\forall H \in \{H_b[j], H_c[j], H_d[j]\} \ (H \notin \{3, 4\})) \\
 &\quad \vee (\forall H \in \{H_b[j], H_c[j], H_d[j]\} \ (H \notin \{7\})) \\
 &\quad \vee (\exists H \in \{H_b[j], H_c[j], H_d[j]\} \ (H \notin \text{Cons}_{h\text{-triad}})) \\
 &\quad \implies \mathcal{C}_{\text{prefer-harmonic-triad}}[j] = \text{not_harmonic_triad_cost} \tag{5.1}
 \end{aligned}$$

Better: no octave, doubling the fifth.

$$\begin{aligned}
 &\forall j \in [0, m - 1]: \\
 &\quad \left(\sum_{H \in \{H_b[j], H_c[j], H_d[j]\}} \mathbf{1}_{\{H=7\}} = 2 \right) \implies (\mathcal{C}_{\text{prefer-harmonic-triad}}[j] = \text{double_fifths_cost}) \tag{5.2}
 \end{aligned}$$

Better yet: no octave, doubling the third.

$\forall j \in [0, m-1):$

$$\left(\sum_{H \in \{H_b[j], H_c[j], H_d[j]\}} \mathbf{1}_{\{H \in \{3,4\}\}} = 2 \right) \implies (\mathcal{C}_{prefer-harmonic-triad}[j] = double_thirds_cost) \quad (5.3)$$

In this case, we must also check that the two thirds are of the same nature (minor or major).

$$\begin{aligned} \forall j \in [0, m-1): \quad & (H_b[j] \in \{3,4\} \wedge H_c[j] \in \{3,4\}) \implies (H_b[j] = H_c[j]) \\ \forall j \in [0, m-1): \quad & (H_b[j] \in \{3,4\} \wedge H_d[j] \in \{3,4\}) \implies (H_b[j] = H_d[j]) \\ \forall j \in [0, m-1): \quad & (H_c[j] \in \{3,4\} \wedge H_d[j] \in \{3,4\}) \implies (H_c[j] = H_d[j]) \end{aligned} \quad (5.4)$$

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

$$\begin{aligned} \forall j \in [0, m-1): \\ & (H_b[j] = 7) \wedge (H_c[j] \in \{3,4\}) \wedge (H_d[j] = 0) \\ & \iff \mathcal{C}_{prefer-harmonic-triad}[j] = triad_with_octave_cost \end{aligned} \quad (5.5)$$

$$\begin{aligned} \forall j \in [0, m-1): \\ & (H_b[j] \in \{3,4\}) \wedge (H_c[j] = 0) \wedge (H_d[j] = 7) \\ & \iff \mathcal{C}_{prefer-harmonic-triad}[j] = triad_with_octave_cost \end{aligned} \quad (5.6)$$

$$\begin{aligned} \forall j \in [0, m-1): \\ & (H_b[j] \in \{3,4\}) \wedge (H_c[j] = 7) \wedge (H_d[j] = 0) \\ & \iff \mathcal{C}_{prefer-harmonic-triad}[j] = triad_with_octave_cost \end{aligned} \quad (5.7)$$

$$\begin{aligned} \forall j \in [0, m-1): \\ & (H_b[j] = 0) \wedge (H_c[j] \in \{3,4\}) \wedge (H_d[j] = 7) \\ & \iff \mathcal{C}_{prefer-harmonic-triad}[j] = triad_with_unison_cost \end{aligned} \quad (5.8)$$

$$\begin{aligned} \forall j \in [0, m-1): \\ & (H_b[j] = 0) \wedge (H_c[j] = 7) \wedge (H_d[j] \in \{3,4\}) \\ & \iff \mathcal{C}_{prefer-harmonic-triad}[j] = triad_with_unison_cost \end{aligned} \quad (5.9)$$

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

$$\begin{aligned} \forall j \in [0, m-1): \\ & (H_b[j] = 7) \wedge (H_c[j] = 0) \wedge (H_d[j] \in \{3,4\}) \\ & \iff \mathcal{C}_{prefer-harmonic-triad}[j] = 0 \end{aligned} \quad (5.10)$$

```

1 //H8 : harmonic triads are preferred, adapted for 4 voices
2
3 for(int i = 0; i < triadCostArray.size(); i++){
4
5     IntVar H_b = upper1->getHInterval()[i*4];
6     IntVar H_c = upper2->getHInterval()[i*4];
7     IntVar H_d = upper3->getHInterval()[i*4];
8
9
10    BoolVar H_b_is_third    = expr(*this, H_b==MINOR_THIRD || H_b==MAJOR_THIRD);
11    BoolVar H_b_is_fifth    = expr(*this, H_b==PERFECT_FIFTH);
12    BoolVar H_b_is_octave   = expr(*this, H_b==UNISSON);
13
14    BoolVar H_c_is_third    = expr(*this, H_c==MINOR_THIRD || H_c==MAJOR_THIRD);
15    BoolVar H_c_is_fifth    = expr(*this, H_c==PERFECT_FIFTH);
16    BoolVar H_c_is_octave   = expr(*this, H_c==UNISSON);
17
18    BoolVar H_d_is_third    = expr(*this, H_d==MINOR_THIRD || H_d==MAJOR_THIRD);
19    BoolVar H_d_is_fifth    = expr(*this, H_d==PERFECT_FIFTH);
20    BoolVar H_d_is_octave   = expr(*this, H_d==UNISSON);
21
22
23    // worst case : not a harmonic triad with at least a third and a fifth
24    BoolVar no_fifth_or_no_third = expr(*this, H_b_is_third + H_c_is_third + H_d_is_third == 0
25    || H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 0);
26    BoolVar note_outside_harmonic_triad = expr(*this, H_b_is_third + H_b_is_fifth +
27    H_b_is_octave == 0 || H_c_is_third + H_c_is_fifth + H_c_is_octave == 0 || H_d_is_third
28    + H_d_is_fifth + H_d_is_octave == 0);
29
30    rel(*this, (note_outside_harmonic_triad || no_fifth_or_no_third) >> (triadCostArray[i] ==
31    not_harmonic_triad_cost));
32
33    // now we are left with only combinations with at a third, a fifth, and another note of
34    // the harmonic triad.
35    // Doubling the fifth
36    rel(*this, expr(*this, H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 2) >> (triadCostArray
37    [i] == double_fifths_cost));
38
39    // Doubling the third, and ensure it is the same type of third (not a major and a minor)
40    rel(*this, expr(*this, H_b_is_third + H_c_is_third + H_d_is_third == 2) >> (triadCostArray
41    [i] == double_thirds_cost));
42    rel(*this, (H_b_is_third && H_c_is_third) >> (H_b == H_c));
43    rel(*this, (H_b_is_third && H_d_is_third) >> (H_b == H_d));
44    rel(*this, (H_c_is_third && H_d_is_third) >> (H_c == H_d));
45
46    // triad with octave
47    rel(*this, (H_b_is_fifth && H_c_is_third && H_d_is_octave) >> (triadCostArray[i] ==
48    triad_with_octave_cost)); // 5 3 8
49    rel(*this, (H_b_is_third && H_c_is_octave && H_d_is_fifth) >> (triadCostArray[i] ==
50    triad_with_octave_cost)); // 3 8 5
51    rel(*this, (H_b_is_third && H_c_is_fifth && H_d_is_octave) >> (triadCostArray[i] ==
52    triad_with_octave_cost)); // 3 5 8
53    rel(*this, (H_b_is_octave && H_c_is_third && H_d_is_fifth) >> (triadCostArray[i] ==
54    triad_with_unison_cost)); // 8 3 5
55    rel(*this, (H_b_is_octave && H_c_is_fifth && H_d_is_third) >> (triadCostArray[i] ==
56    triad_with_unison_cost)); // 8 5 3
57
58    // Best case : 5 8 3
59    rel(*this, (H_b_is_fifth && H_c_is_octave && H_d_is_third) >> (triadCostArray[i] == 0));
60
61 }

```

Listing 5.1: Gecode implementation of H8 constraint

5.1.2 Implicit Rules

G6 [PREF] *Chromatic melodies are to be avoided.* This relaxation of the previous rule, *chromatic melodies are forbidden*, comes from the fact that Fux occasionally uses chromatic melodies in his examples (like in Fig. 160 of [11]). How should this rule be

relaxed? The obvious choice would be to put a new cost, and set it to a value whenever there is a chromatic melody. However, the OM interface already has a large array of costs, and adding complexity is unnecessary when it can be avoided. Here, we just have to notice that there is already a cost for borrowing notes, and chromatic melodies always feature borrowed notes, by definition of the concept of scales. Therefore, any chromatic melody will come with a cost by default, without us having to define a new cost and add it to the user interface.

1.H5 Suspended rule: *The voices cannot play the same note at the same time except in the first and last measures.* This rule was suspended to give more freedom to the solver, especially when trying to maximize the number of harmonic triads used. Moreover, when composing in four voices, the voice ranges overlap more, which makes disallowing unisons much more costly. Lastly, a unison between two voices out of four is less unpleasant than if it was two voices out of three or two.

The suspension of this rule, however, brings back another rule of three voice composition : **1.H11** - *Octaves should be preferred over unisons*, which did not need an explicit mathematical constraint because rule **1.H5** was active. Now that it is deactivated, we would need an explicit constraint to enforce **1.H11**. Fortunately, **1.H8** already takes care of enforcing the preference for octaves over unisons. The *triad_with_unison_cost* must only be set higher than the *triad_with_octave_cost*.

1.H7 Suspended rule: *The harmonic interval of the penultimate measure must be either a minor third, a perfect fifth, a major sixth, or an octave.* This rule suspension was deduced from Fux's examples, which also included major thirds.

1.H10 Suspended rule: *Tenths are prohibited in the last chord*

This rule is suspended because it is too restrictive when writing in four voices. As has been detailed in rule **1.H8**, one of the main goals of (three and) four-part composition is to use the harmonic triad as much as possible, and the natural order has the third pushed above the octave, which is a tenth. This reasoning is confirmed in Fux's examples, where tenths are often used in the last chord (a good example is in figure 5.1).

1.M6 *Melodic intervals cannot be greater or equal to a sixth, except for octaves.*

This is another form of "relaxation": four-part composition is too complex to adhere strictly to rules such as bounding the melodic intervals. Consequently, this rule has to be relaxed. The first form of relaxation we have seen is to relax a rule into a preference. This approach is not suited to this rule, because leaping a seventh or a ninth is something extremely hard to do for a singer, and is overall not desirable melodically. On the other hand, leaping an octave is something that is often done and is not prohibited by general musical rules. Therefore, this rule stays a strict rule and not a preference, but octave leaps are now allowed.

The new mathematical formalization of this rule is given below.

$$\forall p \in \{cp_1, cp_2, cp_3\}, \forall j \in [0, m-1] : \quad (M[0, j] \leq 8) \vee (M[0, j] = 12) \quad (5.11)$$

Once more, the reasoning that lead to the modification of this rule is confirmed in Fux's examples (for example in fig. 170 of [11], page 115).

1.P5 Suspended rule: *Each part starts distant from the lowest stratum.*

Again, this is a rule that limits the possibilities of four voice counterpoint composition, and since Fux does not necessarily follow it in his examples, it has been suspended.

5.2 Second Species

This species, as a reminder, consists of two notes for every note in the *cantus firmus*, so two half notes for every whole note. One of Fux's examples is given below.

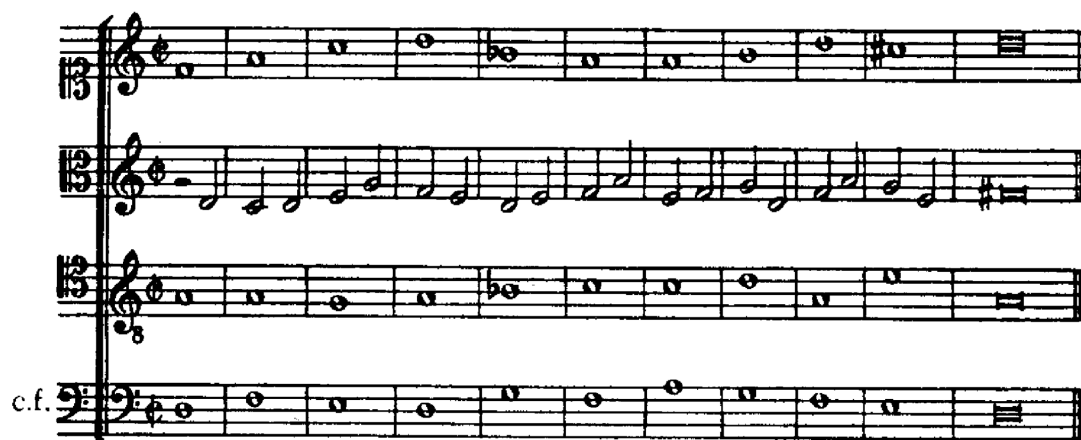


Figure 5.3: Example of a four-voice counterpoint composition in the second species. Mann [11, p.118, fig.176]

The chapter of *Gradus ad Parnassum* that describes second species composition in four voices is extremely short : only three paragraphs long. It begins with the following guideline :

"Here you must recall what has been said previously, in three part composition, concerning this species of counterpoint. All this must be kept in mind when composing in four parts, and there is no difference except that there the two half notes must agree with two whole notes; here, however, with three. For the rest, one should observe what has been said about four part composition of note against note, as far, though, as the restricted nature of this species permits. " Mann [11, p.117]

In other words, we must keep the rules that were given for three voice composition in the second species, and keep to the rules of the first species for four part composition as much as possible. No new explicit rule is given in this species, and the complete set of rules will be given in the next chapter, so let us move on to the third species.

5.3 Third Species



Figure 5.4: Example of a four-voice counterpoint composition in the third species. Mann [11, p.122, fig.180]

The chapter of *Gradus ad Parnassum* that talks about third species counterpoint in four voices starts in a similar way as the previous one :

"The requirements of this species may be recalled from what has been said about it in two and three part composition. Nothing else need be added, except that the concords are to be dealt with differently. Just as the concords had to be managed with one whole note in two part composition, and with two whole notes in three part composition, so now in four part composition the four quarters must be concordant with three whole notes according to the principles of harmony which you already know. "

Mann [11, p.120]

Note : This passage just expresses an extension of rule 3.H5 - *The quarter notes must be concordant with respect to the whole notes.*

We would be tempted to reason as we did with the second species and to deduce that no additional explicit rules must be added, but when taking a closer look we can decipher some important remarks made by Fux, which should influence the way we compose third species counterpoint in four voices.

The first main theme that can be extracted from this chapter of the book is that the necessity to compose the other counterpoints (those which are not in the third species) using only whole notes becomes very constraining.

"This progression could not be managed otherwise because one must necessarily use whole notes, and therefore it may be tolerated. [...] This holds also for the examples of the previous species in which there are many places that would be considered faulty if it were not for the necessity of using whole notes." Mann [11, p.121]

Now we may ask ourselves, why do we necessarily have to have only whole notes in all of the other parts? In other words, why do all of the other counterpoints have to be of the first species? Here one must recall that *Gradus ad Parnassum* is written as a dialogue between a master and a student, and is thus meant to teach counterpoint, instead of just providing an exhaustive list of rules (which would, paradoxically, have

simplified our task!). This will to gradually teach counterpoint to his student is put into shape with the five species of counterpoint, which are increasingly complex. Fux says himself that the species of counterpoint are given only for purposes of study. It can thus be deduced that keeping whole notes in all of the other parts than the one we are focusing on is a matter of simplicity for his student, so that he can focus on one part of the species he is learning. However, we will see at the end that Fux does indeed encourage his student to use different species for the different parts, but this is after having finished all five species.

Let us now move on to the two additional explicit rules for four-part composition in the third species.

5.3.1 Explicit Rules

3.H7 *It is now preferred to double the third or the fifth on the downbeat, rather than to use a unison.*

"A unison on the downbeat detracts considerably from the complete harmony of the composition. " Mann [11, p.121]

This rule can be easily taken into account with the *triad_with_unison_cost* in rule **1.H8**. In the context of this citation, it is chosen to double the third instead of using a unison, so when writing for the third species, the *triad_with_unison_cost* should be superior to the *double_thirds_cost* and the *double_fifths_cost*.

3.P2 *Progressions from one perfect consonance to another by direct motion are to be tolerated in the inner voices rather than in the outer voices.*

"[Progressions from one perfect consonance to another by similar motion] are to be tolerated in the inner voices rather than in the outer voices." Mann [11, p.123]

This induces a modification of rule **1.P1** - *Reaching a perfect consonance by direct motion should be avoided*. The previous formalization of this rule was the following :

$$\begin{aligned} \forall j \in [0, m-1) : \\ P[0, j] = 2 \wedge H[0, j+1] \in \text{Cons}_p \iff \mathcal{C}_{\text{direct_move_to_p_cons}}[j] = 8 \end{aligned} \quad (5.12)$$

The rule is now :

$$\begin{aligned} \forall j \in [0, m-1) : \\ P(b)[0, j] = 2 \wedge H(b)[0, j] \in \text{Cons}_p \wedge H(b)[0, j+1] \in \text{Cons}_p \iff \mathcal{C}_{\text{direct_move_to_p_cons}}[j] = 2 \\ P(c)[0, j] = 2 \wedge H(c)[0, j] \in \text{Cons}_p \wedge H(c)[0, j+1] \in \text{Cons}_p \iff \mathcal{C}_{\text{direct_move_to_p_cons}}[j] = 2 \\ P(d)[0, j] = 2 \wedge H(d)[0, j] \in \text{Cons}_p \wedge H(d)[0, j+1] \in \text{Cons}_p \iff \mathcal{C}_{\text{direct_move_to_p_cons}}[j] = 8 \end{aligned} \quad (5.13)$$

Although Fux says this in the third chapter (about the third species), it is formulated in a way that suggests that it is applicable in all context of four voice counterpoint.

5.3.2 Implicit Rules

There is no particular implicit rule that can be deduced from these new rules or from the examples. However, one piece of feedback from the teacher to his pupil does guide us in knowing what is really the priority for Fux when composing four-voice counterpoint in the third species :

"You have given proof of the remarkable progress that you have made, and that of all things that have been explained in the course of so many lessons nothing has escaped your memory. For you have always used a complete harmonic triad on the first quarter, and from there you have had three remaining quarters move in such a way that the following measure was approached simply. Moreover, you have correctly observed the rules of harmony and of the different species by providing a full harmonic triad, namely, a third and a fifth, on the downbeat or beginning of each measure.
" Mann [11, p.122]

We can deduce from this passage that Fux is particularly attentive about the use of the harmonic triad on downbeats, and we can adjust the costs and preferences accordingly.

5.4 Fourth Species



Figure 5.5: Example of a four-voice counterpoint composition in the fourth species. Mann [11, p.132, fig.196]

We have now arrived to the fourth species : the ligature, or syncopation. As a reminder, syncopation creates an off-balance rhythm through the accenting of normally unaccented beats. Here it is done by "delaying" whole notes so that they occur on the upbeat instead of the downbeat, resulting in ligatures which make sure they sound all the way to the next upbeat.

In the chapter concerning four voice composition with a fourth species counterpoint, Fux begins by stating the general guideline, which is similar to the one he stated in the chapter in three voices. This guideline is to compose using the consonances which would also be used if the ligatures with removed (because the ligature is nothing but a delaying of the note following, and this delay does not alter anything with respect to the consonances). So the first rule we note is that here too, the harmonic triad in its natural form should be used as much as possible, as was first introduced by rule 1.H8. This rule requires no new explicit mathematical formulation, because this logic (of treating the ligature as a mere delay of a whole note) is already built into how the fourth species is coded.

Resolving conflicts : allowing the division of a whole note

While we do treat this species as a "delayed" first species where we can, it must not be forgotten that this delay causes two different notes to sound together with the real whole notes of the other parts. This can sometimes create conflict, especially in four voices. The most common example of such conflict is when the ligature causes a seventh on the downbeat, and that another part has a whole note constituting a fifth. This is problematic because, following rule 4.P1, the seventh will resolve into a sixth on the upbeat, causing a harsh dissonance with the fifth. The solution that Fux proposes is to divide the whole note sounding the fifth into two half notes, as can be seen in the example below.

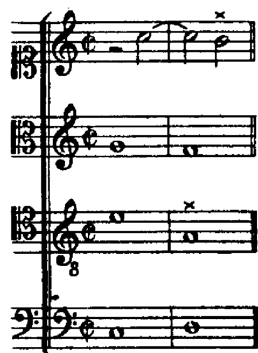


Figure 5.6: Problem : dissonance between the fifth and the sixth

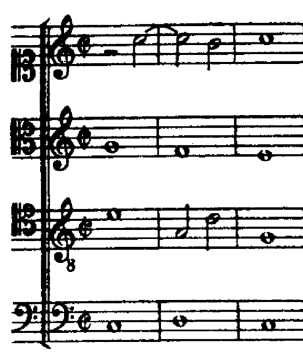


Figure 5.7: Solution : dividing the whole note

This decision is quite interesting, especially because in the third species it was explicitly said multiple times that the necessity of composing with whole notes was extremely constraining, and that important rules could be relaxed to satisfy this necessity. However, in this species, when we encounter the same problem, dividing the whole note is allowed.

"A number of cases will occur [...] wherein one cannot help making a division. Therefore, the rule that one must write three whole notes cannot be strictly observed in this species." Mann [11, p. 129]

This division is of course to be avoided if possible, so its implementation will go with a cost. However, in the scope of this thesis, it was chosen not to implement this exception, as it introduces a variety of changes that would be too costly for the small feature they give. Moreover, as was explained previously, the ultimate goal of these species of counterpoint is not for them to be used with whole notes only, but to be used with other rich counterpoints. The whole notes were there for pedagogical purposes. When composing with other species of counterpoints in the other parts, the problem disappears because there can already be multiple notes per measure.

There are a few other rules expressed by Fux in the chapter regarding this species, but nothing that is not already taken into account by previous rules (remember, Fux likes to repeat important points). We can now move on to the fifth and last species of counterpoint.

5.5 Fifth Species



Figure 5.8: Example of a four-voice counterpoint composition in the fifth species. Mann [11, p.135, fig.200]

"Nothing new need be added here, except that the fourth voice must also be in whole notes and must be worked in according to the principles of four part composition. [...] When I say that the whole notes must remain undivided, I mean whenever this is possible. However, you will notice that where there was an opportunity the rule was strictly observed. " Mann [11, p.135]

This passage of *Gradus ad Parnassum* carries two messages. The first was already present in the chapters about the third and second species : no new explicit rule is added, but the newly added fourth voice now needs to be taken into account while composing. This is done mainly with the principles introduced with the fourth species, especially about the use of the harmonic triad.

The second message is about the division of notes. Like in the fourth species, some rules are considered by Fux to be more important than the strict use of whole notes. Again, this rule was not implemented because it was judged too costly in rethinking of the whole code architecture for the little benefit it brings. Moreover, as soon as we start composing with multiple counterpoints of different species, the feature becomes useless : the other species have more notes per measure, so the dilemma of dividing a whole note or not disappears. If there is still one first species counterpoint in the other counterpoints, the conflict can be avoided using the added freedom of having multiple notes per measure in the other species.

5.6 Combining Different Species



Figure 5.9: Example of a four-voice counterpoint composition combining different species. Mann [11, p.137, fig.204]

"As we have now completed the five species singly, I want to urge you to write them also in combination. Keeping the same cantus firmus you may combine, for example, half notes, quarters, and ligatures. Thus each part will have its own characteristic motion, and the whole composition will possess a wonderful variety. [...] In doing this you should make an effort to take into account at once what has been said about each species separately and to observe strictly what applies to the voices in their relation to each other. "

This is where it gets interesting, because by combining different species of counterpoint we immediately get a whole new dimension of variety and beauty. The difficulty is knowing which rule should be applied when, because until now all rules had been applied only with whole notes in Fux's examples. It was therefore decided by A. Lamotte to make the first note of each measure the important note (and essentially the only note) of the measure when applying rules from another counterpoint. However, this convention is not enough to determine the exact rules to apply with a given configuration of counterpoints. The goal of the next chapter is to provide the complete set of rules, and to detail exactly in which configurations the rules apply, so that no ambiguity remains.

Chapter 6

Complete Set of Rules and Their Scopes

Rewriting the whole formalizations of A. Lamotte and T. Wafflard in C++, and extending to four voices, has given us a general overlook on the formalization and on the different rules. This step back that we had to take enabled us to see the big picture of this constraint programming formalization more clearly, and to gain perspective upon how the different concepts and rules interact. Therefore, we were able to merge together multiple rules which essentially had the same meaning or goal, correct mistakes that were made when trying to convert a rule from two to three voices, and overall create a smoother and more harmonious formalization across all of the different counterpoint problem configurations.

Moreover, Fux is often redundant in *Gradus ad Parnassum*. This is most likely done on purpose, as the work is written as a dialogue between a pupil and his student, so the key concepts have to be repeated in order for the student to understand them fully. This repetition resulted in rules being formalized multiple times, as happened for instance with rules **1.M2** of T. Wafflard and **1.M6** of A. Lamotte, which both come from the same idea in Fux's work : major sixth melodic leaps are forbidden. In this renewed set of rules, we have only included the rules that bring about a new constraint, and taken out (or only mentioned) redundant and useless rules.

Lastly, the different rules are given throughout the book, and it is sometimes tricky to know exactly when they should apply. For example, some rules that are given in the chapter about third species composition should actually apply more broadly than just to the third species, and this is insight that can only be gained by having a complete overlook over the whole book and the entire set of rules contained inside of it, which we are the first in this project to be fortunate enough to have.

This chapter presents the final and complete set of rules, wrapping up the mathematical formalization of Fux's theory of counterpoint. For each rule, the simple English formulation is given, followed by the mathematical formulation (when applicable), and the scope of the rule (detailing in what specific situations the rule should apply).

6.1 General and Implicit Rules

Rules **G1** through **G3** do not have mathematical formalizations, because they are not implemented as constraints, but rather built into the very structure of the program. Moreover, they apply in every configuration (any number of voices, in any species). For more details on the rules, their logic, or where they come from in *Gradus ad Parnassum*, refer to the corresponding thesis (T. Wafflard [1] for two-voice rules, A. Lamotte [2] for three-voice rules, or the previous chapter of this thesis for four-voice rules.

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

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

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

The two following rules are taken into account by the creation of $\mathcal{N}_{all}^{\mathcal{R}}$, the set of all pitches available to the counterpoint notes, which is a function of the voice range, the tonic (first note of the *cantus firmus*), and the borrowing mode (see Appendix A).

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 rule is very similar to rule **1.M5** - *Each part should stay in its voice range*. As no difference is made between the two in the implementation (and they are essentially expressing the same idea), **1.M5** will not be mentioned again in this list.

Rules **G6** and **G7** are more melodic than general rules, but they were implicit, and for historical reasons we will keep them as **G** rules.

G6 *Chromatic melodies are forbidden for two and three voice composition, and to be avoided for four voice composition.*

$$\begin{aligned} & \forall \rho \in \text{positions}(m-2) \\ \mathbf{2V \text{ and } 3V:} \quad & (M_{brut}[\rho] = 1 \wedge M_{brut}[\rho+1] = 1) \iff \perp \\ & (M_{brut}[\rho] = -1 \wedge M_{brut}[\rho+1] = -1) \iff \perp \end{aligned} \quad (6.1)$$

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.*

$$\begin{aligned} & \forall \rho \in \text{positions}(m-1) \\ Mdeg_{costs}[\rho] = & \begin{cases} cost_{secondMdeg} & \text{if } M[\rho] \in \{0, 1, 2\} \\ cost_{thirdMdeg} & \text{if } M[\rho] \in \{3, 4\} \\ cost_{fourthMdeg} & \text{if } M[\rho] = 5 \\ cost_{tritoneMdeg} & \text{if } M[\rho] = 6 \\ cost_{fifthMdeg} & \text{if } M[\rho] = 7 \\ cost_{sixthMdeg} & \text{if } M[\rho] \in \{8, 9\} \\ cost_{seventhMdeg} & \text{if } M[\rho] \in \{10, 11\} \\ cost_{octaveMdeg} & \text{if } M[\rho] = 12 \end{cases} \end{aligned} \quad (6.2)$$

This formalization also takes care of the following rules:

- **1.M1** - *Tritone melodic intervals are forbidden.* The cost for tritones, $cost_{tritoneMdeg}$, is set very high by default.
- **1.M3** - *Steps are preferred to skips.* The cost for steps is set significantly lower than the cost for skips.

We can conclude that this general melodic rule defined by T. Wafflard englobes the formalizations of many other melodic rules, which we can now omit to make this list of rules more concise and less repetitive.

This rule applies to all configurations.

6.2 Constraints of the First Species

6.2.1 Harmonic Rules

1.H1 *All notes on the downbeat are consonant with the notes (on the downbeat) of the lowest stratum. (and previously 2.H1 - Thesis harmonies cannot be dissonant)*

$$\forall j \in [0, m) \quad H[0, j] \in Cons \quad (6.3)$$

This rule applies to all configurations.

1.H2 *In two voice composition, the first harmonic interval must be a perfect consonance.*

$$\mathbf{2V:} \quad H[0, 0] \in Cons_p \quad (6.4)$$

This rule applies to all species in two voices.

1.H3 (and previously G8 and 2.H5) *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.*

$$\begin{aligned} \mathbf{2V:} \quad & H[0, m-1] \in Cons_p \\ \mathbf{3V, 4V:} \quad & \forall s \in \{b, c, d\}: H(s)[0, m-1] \in Cons_{h_triad} \end{aligned} \quad (6.5)$$

This rule applies to all configurations. Remember that for all configurations, no matter the species of the counterpoints, the last measure always contains a whole note.

1.H4 (and previously G9) *The key tone is tuned according to the first note of the cantus firmus. In other words, the lowest stratum at the first and last notes must be the tonic.*

$$\begin{aligned} N(a)[0, 0] \bmod 12 &= N(cf)[0, 0] \bmod 12 \\ N(a)[0, m-1] \bmod 12 &= N(cf)[0, 0] \bmod 12 \end{aligned} \quad (6.6)$$

This rule applies to all configurations.

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

$$\begin{aligned} & \forall p_1, p_2 \in \{cf, cp_1, cp_2\}, \text{ with } p_1 \neq p_2, \\ & \forall i \in \{0, 1, 2, 3\} \text{ and } \forall j \in [1, m-1]: \quad N(p_1)[i, j] \neq N(p_2)[i, j] \end{aligned} \quad (6.7)$$

This rule applies for thesis notes to all species in two and three voices.

1.H6 and 4.H4 and 4.P4 *For thesis notes, imperfect consonances are preferred to perfect consonances, and fifths are preferred to octaves.*

"The fifth is a perfect consonance, the octave a more perfect one, and the unison the most perfect of all; and the more perfect a consonance, the less harmony it has." Mann [11, p.97]

$$\begin{aligned} & \forall j \in [0, m) \\ Pcons_{costs}[j] &= \begin{cases} cost_{Pcons_oct} & \text{if } H[0, j] = 0 \\ cost_{Pcons_fifth} & \text{if } H[0, j] = 7 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \quad (6.8)$$

This rule applies to all configurations.

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.

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

$$\mathbf{2V:} \quad H[\rho] = \begin{cases} 9 & \text{if } A(cf)[\rho] \\ 3 & \text{otherwise} \end{cases} \quad (6.9)$$

where ρ represents the penultimate index of any counterpoint.

$$\mathbf{3V:} \quad H[0, m-1] \in \{0, 3, 7, 9\} \quad (6.10)$$

This rule applies to the penultimate **note** (not measure!) for every species in two or three voices.

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.

$$\mathbf{3V:} \quad \forall j \in [0, m-1]: \quad (H(b)[0, j] \notin \{3, 4\}) \vee (H(c)[0, j] \neq 7) \iff C_{\text{prefer-harmonic-triad}}[j] = 1 \quad (6.11)$$

4V : Long and complex mathematical formalization, see **1.H8**.

This rule applies to all thesis chords in three and four part composition, in all species.

1.H10 In three voice composition, tenths are prohibited in the last chord.

$$H_{\text{brut}}[0, m-1] > 12 \implies H[0, m-1] \notin \{3, 4\} \quad (6.12)$$

This rule applies to all species in three voices.

1.H11 Octaves should be preferred over unisons. This rule is taken into account by **1.H5** for two and three voices, and by **1.H8** for four voices.

6.2.2 Melodic Rules

1.M2 Melodic intervals cannot exceed a minor sixth interval, but octave leaps are allowed in three and four voices.

$$\mathbf{2V:} \quad \forall j \in [0, m-1) \quad M[0, j] \leq 8 \quad (6.13)$$

$$\mathbf{3v \text{ and } 4V:} \quad \forall j \in [0, m-1) \quad (M[0, j] \leq 8) \vee (M[0, j] = 12) \quad (6.14)$$

This rule applies to all first species counterpoints.

1.M4 The notes of each part should be as diverse as possible.

$$\forall p \in \{cp_1, cp_2, cp_3\}, \quad \forall j \in [0, m), \quad \forall k \in [j+1, \min(j+3, m-1)] : \quad N(p)[0, j] = N(p)[0, j+k] \iff C_{\text{variety}}[j+m*k] = 2 \quad (6.15)$$

For this rule, it is important to clarify that Fux is vague when talking about the variety of the notes. Therefore, the implementation of this rule is a judgement made by Lamotte and ourselves. In the future, the constraint could be revisited, whether for musical reasons or for optimization. For example, one might use a cardinality constraint for efficiency purposes.

This rule applies to all configurations in three and four voices.

6.2.3 Motion Rules

1.P1 and 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.*

$$\mathbf{2V:} \quad \forall j \in [0, m-1) \quad H[0, j+1] \in \text{Cons}_p \implies P[0, j] \neq 2 \quad (6.16)$$

$$\mathbf{3V:} \quad \forall j \in [0, m-1) : \quad P[0, j] = 2 \wedge H[0, j+1] \in \text{Cons}_p \iff \mathcal{C}_{\text{direct_move_to_p_cons}}[j] = 8 \quad (6.17)$$

$$\mathbf{4V:} \quad \forall j \in [0, m-1) : \quad \begin{aligned} P(b)[0, j] = 2 \wedge H(b)[0, j] \in \text{Cons}_p \wedge H(b)[0, j+1] \in \text{Cons}_p &\iff \mathcal{C}_{\text{direct_move_to_p_cons}}[j] = 2 \\ P(c)[0, j] = 2 \wedge H(c)[0, j] \in \text{Cons}_p \wedge H(c)[0, j+1] \in \text{Cons}_p &\iff \mathcal{C}_{\text{direct_move_to_p_cons}}[j] = 2 \\ P(d)[0, j] = 2 \wedge H(d)[0, j] \in \text{Cons}_p \wedge H(d)[0, j+1] \in \text{Cons}_p &\iff \mathcal{C}_{\text{direct_move_to_p_cons}}[j] = 8 \end{aligned} \quad (6.18)$$

This rule applies to all configurations.

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

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

$$\forall j \in [0, m-1) \quad P_{\text{costs}}[j] = \begin{cases} \text{cost}_{\text{con}} & \text{if } P[0, j] = 0 \\ \text{cost}_{\text{obl}} & \text{if } P[0, j] = 1 \\ \text{cost}_{\text{dir}} & \text{if } P[0, j] = 2 \end{cases} \quad (6.19)$$

This rule applies to all configurations.

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.*

$$\begin{aligned} i &:= \max(\mathcal{B}), \forall j \in [0, m-1), \forall p \in \{cf, cp_1, cp_2, cp_3\} \\ H(p)[0, j+1] = 0 \wedge P(p)[i, j] = 0 \wedge M_{\text{brut}}(p)[i, j] < -4 \wedge \neg A(p) &\iff \perp \end{aligned} \quad (6.20)$$

where i is the last beat index in a measure.

1.P4 *Successive perfect consonances should be avoided.*

$$\begin{aligned} &\forall p_1, p_2 \in \{cf, cp_1, cp_2, cp_3\}, \text{ where } p_1 \neq p_2, \quad \forall j \in [0, m-1): \\ &(H(p_1, p_2)[0, j] \in Cons_p) \wedge (H(p_1, p_2)[0, j+1] \in Cons_p) \\ &\implies C_{succ_p_cons} = 2 \end{aligned} \quad (6.21)$$

This rule applies to all configurations, except in the second species for three and four voices, where rule **2.P3** allows for an exception.

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.*

$$\begin{aligned} &\forall j \in [0, m-1): \\ &\bigvee_{p \in \{cf, cp_1, cp_2, cp_3\}} P(p)[0, j] \in \{0, 1\} \end{aligned} \quad (6.22)$$

This rule applies to all configurations in three or four voices.

1.P7 *It is prohibited to use successive ascending sixths on a direct upwards motion.*

$$\begin{aligned} &\forall j \in [1, m-1), \quad \forall p_1, p_2 \in \{cf, cp_1, cp_2, cp_3\} \text{ where } p_1 \neq p_2, \quad \text{sixth} := \{8, 9\}: \\ &(H(p_1, p_2)[0, j-1] \notin \text{sixth}) \vee (H(p_1, p_2)[0, j] \notin \text{sixth}) \\ &\vee M_{brut}(p_1)[0, j] \leq 0 \vee M_{brut}(p_2)[0, j] \leq 0 \end{aligned} \quad (6.23)$$

This rule applies to all configurations in three or four voices.

6.3 Constraints of the Second Species

6.3.1 Harmonic Rules

2.H2 *Arsis harmonies cannot be dissonant except if there is a diminution*¹.

$$\begin{aligned} &\forall j \in [0, m-1), \forall p : \text{species}(p) = 2 \\ IsDim(p)[j] &= \begin{cases} \top & \text{if } M^2(p)[0, j] \in \{3, 4\} \wedge M^1(p)[0, j] \in \{1, 2\} \wedge M^1(p)[2, j] \in \{1, 2\} \\ \perp & \text{otherwise} \end{cases} \\ &\forall j \in [0, m-1) \quad \neg IsCons(p)[2, j] \implies IsDim(p)[j] \end{aligned} \quad (6.24)$$

This rule applies to all numbers of voices, on second species counterpoints.

Together with **1.H1**, these two rules define the harmonic essence of the second species : there are two notes per measure, the first of which (on the downbeat) is consonant, and the second of which (on the upbeat) can be dissonant if it is part of a diminution.

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

2.H3 and 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.

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

$$penultthesis_{cost} = \begin{cases} cost_{penultthesis} & \text{if } H[0, m-2] \neq 7 \\ 0 & \text{otherwise} \end{cases} \quad (6.25)$$

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

2.H6 and 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*.

6.3.2 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.

$$\forall j \in [0, m-1), \forall M_{cf}[j] \neq 0$$

$$M[0, j] = 12 \implies (H[0, j] \leq 4) \wedge (\neg A(cp)[j] \iff M_{cf}[j] > 0) \quad (6.26)$$

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

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.

$$\mathbf{2V:} \quad \forall \rho \in positions(m) \quad N[\rho] \neq N[\rho+1] \quad (6.27)$$

$$\begin{aligned} & \forall p : species(p) = 2, \forall j \in [1, m-1), \quad j \neq m-2 : \\ \mathbf{3v \text{ and } 4V:} & ((N[2, j-1] \neq N[0, j]) \wedge (N[0, j] \neq N[2, j])) \\ & \wedge \\ & ((N[2, m-3] \neq N[0, m-2]) \vee (N[0, m-2] \neq N[2, m-2])) \end{aligned} \quad (6.28)$$

This rule applies to all configurations, to second species counterpoints.

6.3.3 Motion Rules

2.P1 and 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.

$$\forall j \in [0, m-1) \quad P_{real}[j] = \begin{cases} P[2, j] & \text{if } M[0, j] > 4 \\ P[0, j] & \text{otherwise} \end{cases} \quad (6.29)$$

Note : The rule concerning the battuta octave (**1.P3**) always preceives motion from the last note of the previous measure, but this is already explicitly formulated in the formalization of the rule, so it does not impact this rule or need a new formulation.

This rule applies to all counterpoints of the second species.

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.

$\forall p_1, p_2 \in \{cf, cp_1, cp_2, cp_3\}$ where $p_1 \neq p_2$ and $species(p_1) = 2 \vee species(p_2) = 2$,
 $\forall j \in [0, m - 2)$:

$$Cost_{succ_p_cons} = \begin{cases} 0 & \text{if } (H(p_1, p_2)[0, j] \notin Cons_p) \vee (H(p_1, p_2)[0, j + 1] \notin Cons_p) \\ 0 & \text{if } (H(p_1, p_2)[0, j] = 5) \wedge (H(p_1, p_2)[0, j + 1] = 5) \\ & \wedge (H(p_1, p_2)[2, j] = 3) \vee (H(p_1, p_2)[2, j] = 4) \\ 2 & \text{otherwise} \end{cases} \quad (6.30)$$

This rule applies to all configurations in three and four voice composition in which second species counterpoints are involved.

6.4 Constraints of the Third Species

6.4.1 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.

$$\begin{aligned} & \forall j \in [0, m - 1) \\ & \left(\bigwedge_{i=0}^3 M[i, j] \leq 2 \right) \wedge \left(\bigwedge_{i=0}^3 M_{brut}[i, j] > 0 \vee \bigwedge_{i=0}^3 M_{brut}[i, j] < 0 \right) \\ & \implies IsCons[2, j] \end{aligned} \quad (6.31)$$

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

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.

$$\begin{aligned} & \forall j \in [0, m - 1) \\ & IsCons[2, j] \vee (IsCons[1, j] \wedge IsCons[3, j] \wedge IsDim[j]) \end{aligned} \quad (6.32)$$

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

This rule applies in all configurations, to third species counterpoints. It translates a very important concept in counterpoint (which was already present in the second species) : all dissonances are surrounded by consonances.

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.

$$\begin{aligned} & \forall j \in [0, m - 1) \\ & Cambiata_{costs}[j] = \begin{cases} cost_{Cambiata} & \text{if } IsCons[1, j] \wedge IsCons[2, j] \wedge M[1, j] \leq 2 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \quad (6.33)$$

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

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.

$$A(cp)[m-2] \implies H[0, m-2] = 3 \quad (6.34)$$

This rule applies in two voices when the counterpoint is of the third species. It was decided that it would not be applicable in three and four voices, because it would not make any musical sense anymore, unlike the three previous rules.

3.H6 *If the harmonic triad could not be used on the downbeat, it should be used on the second or third beat.*

$$\begin{aligned} \forall j \in [0, m-1): \\ (H[0, j] \notin Cons_{h_triad}) \wedge (H[1, j] \notin Cons_{h_triad}) \wedge (H[2, j] \notin Cons_{h_triad}) \quad (6.35) \\ \iff cost_{harmonic-triad-3rd-species}[j] = 1 \end{aligned}$$

This rule applies in all three and four voice configurations in which there is a third species counterpoint.

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.* As was explained in the previous chapter (**3.H7**), all is needed for this rule to be taken into account is to set the cost coefficients accordingly.

This rule applies only in four voice composition, when a three-voice counterpoint is involved.

6.4.2 Melodic Rules

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

$$\begin{aligned} \forall \rho \in positions(m-2) \\ MtwoSame_{costs}[i, j] = \begin{cases} cost_{MtwobSame} & \text{if } M^2[\rho] = 0 \\ 0 & \text{otherwise} \end{cases} \quad (6.36) \end{aligned}$$

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

6.4.3 Motion Rules

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

$$\forall j \in [0, m-1) \quad P_{real}[j] = P[3, j] \quad (6.37)$$

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

6.5 Constraints of the Fourth Species

6.5.1 Harmonic Rules

4.H1 *Arsis harmonies must be consonant.*

$$\forall j \in [0, m-1) \quad H[2, j] \in Cons \quad (6.38)$$

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

4.H2 *In two part composition, if the fourth species counterpoint is the lowest stratum, then no harmonic seventh interval can occur.*

$$\forall j \in [1, m-1) \quad A(cp)[j] \implies H[0, j] \notin \{10, 11\} \quad (6.39)$$

This rule applies to two voice compositions in which the counterpoint is of the fourth species.

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

$$H[0, m-2] \in \begin{cases} \{10, 11\} & \text{if } A(cf)[m-2] \\ \{1, 2\} & \text{otherwise} \end{cases} \quad (6.40)$$

This rule applies to two voice compositions in which the counterpoint is of the fourth species.

6.5.2 Melodic Rules

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

$$\forall j \in [0, m-1) \quad NoSync_{costs} = \begin{cases} cost_{NoSync} & \text{if } M[2, j] \neq 0 \\ 0 & \text{otherwise} \end{cases} \quad (6.41)$$

This rule applies in all configurations, to fourth species counterpoints. 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.*

$$\forall j \in [0, m-1) \quad MtwomSame_{costs} = \begin{cases} cost_{MtwomSame} & \text{if } N[2, j] = N[2, j+2] \\ 0 & \text{otherwise} \end{cases} \quad (6.42)$$

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

6.5.3 Motion Rules

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

$$\forall j \in [1, m-1) \quad \neg IsCons[0, j] \implies M_{brut}[0, j] \in \{-1, -2\} \quad (6.43)$$

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

4.P2 *If the fourth species counterpoint is not the lowest stratum then no second harmony can be preceded by a unison/octave harmony.*

$$\forall j \in [1, m-1) \quad \neg A[j+1] \implies H[2, j] \neq 0 \wedge H[0, j+1] \notin \{1, 2\} \quad (6.44)$$

This rule is a syncopation variant of rule **1.P1**, no perfect consonances can be reached by direct motion. It applies only to octaves because fifths are considered more harmonious, and Fux says that "in the case of fifths, the retardation can mitigate the effect of parallel motion. Successions of fifths may therefore be used with syncopations." For a more complete explanation, as usual, refer to the thesis that introduced the rule, in this case T. Wafflard's thesis.

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

4.P3 In three and four voice composition, successive fifths are allowed when using ligatures.

$\forall p_1, p_2 \in \{cf, cp_1, cp_2, cp_3\}, \quad \text{with } p_1 \neq p_2 \text{ and } species(p_1) = 4 \vee species(p_2) = 4,$
 $\forall j \in [0, m - 2):$

$$Cost_{succ_p_cons} = \begin{cases} 0 & \text{if } (H(p_1, p_2)[0, j] \notin Cons_p) \vee (H(p_1, p_2)[0, j + 1] \notin Cons_p) \\ 0 & \text{if } (H(p_1, p_2)[0, j] = 5) \wedge (H(p_1, p_2)[0, j + 1] = 5) \\ 2 & \text{otherwise} \end{cases} \quad (6.45)$$

This rule applies to all fourth species counterpoints in three or four voice configurations.

4.P5 In three and four part composition, stationary movement in the bass implies dissonance in the fourth species part.

$$\begin{aligned} \forall j \in [0, m - 1) : \\ M(a)[0, j] \neq 0 &\iff H[2, j] \in Cons \\ M(a)[0, j] = 0 &\iff H[2, j] \in Dis \end{aligned} \quad (6.46)$$

This rule applies to all fourth species counterpoints in three or four voice configurations.

4.P6 In three and four part composition, a note provoking a hidden fifth gets replaced by a rest.

$$\begin{aligned} \forall j \in [1, m - 1) : \\ H[0, j] = 7 \wedge P[0, j] = 2 &\iff N(0, j - 1) = \emptyset \end{aligned} \quad (6.47)$$

Remember that this is implemented as an *a posteriori* rule, so it is only taken into account *after* the solution is generated by the solver (cf. A. Lamotte's thesis). This rule applies in all configurations, to fourth species counterpoints.

6.6 Constraints of the Fifth Species

Remember that the fifth species works in a different way than the four first ones. It uses the **S** array to determine the species of each note, in other words the nature (duration) of the note and the rules that apply to it. The **Rhythm** rules that are listed below apply to the **S** array, from which is deducted the set of rules to use for each note using equations of subsection 6.6.

Rhythm Rules (concerning the **S** array)

To make the formalization more readable, we define IsS_x :

$$\begin{aligned} \forall x \in \{0, 1, 2, 3, 4\}, \forall \rho \in positions(m) \\ IsS_x[\rho] = \begin{cases} \top & \text{if } S[\rho] = x \\ \perp & \text{otherwise} \end{cases} \end{aligned} \quad (6.48)$$

Now on to the rules.

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

$$\begin{aligned} \forall j \in [0, m) \\ \neg IsS_0[0, j] \quad \text{where } j \neq 0 \\ \neg IsS_0[2, j] \quad \text{where } j \neq m - 1 \end{aligned} \quad (6.49)$$

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

$$\forall i \in \{1, 3\}, \forall j \in [0, m) \quad \neg IsS_4[i, j] \quad (6.50)$$

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.

$$\begin{aligned} \forall j \in [0, m-1) \\ IsS_4[2, j] &\iff IsS_4[0, j+1] \\ IsS_4[2, j] &\implies IsS_0[3, j] \end{aligned} \quad (6.51)$$

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

$$\forall \rho \in positions(m-1) \quad IsS_3[\rho] \implies \neg IsS_0[\rho+1] \quad (6.52)$$

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

$$\forall \rho \in positions(m) \quad \neg IsS_1[\rho] \wedge \neg IsS_2[\rho] \quad (6.53)$$

5.R6 The first and penultimate measures are linked to the 4th species.

$$\begin{aligned} IsS_0[0, 0] \wedge IsS_0[1, 0] \wedge IsS_4[2, 0] \\ IsS_4[0, m-2] \wedge IsS_0[1, m-2] \wedge IsS_4[2, m-2] \end{aligned} \quad (6.54)$$

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.

$$\begin{aligned} \forall j \in [1, m-1) \\ \neg IsCons[0, j] \wedge IsS_4[0, j] \implies M_{brut}^2[0, j] \in \{-1, -2\} \wedge IsCons[2, j] \end{aligned} \quad (6.55)$$

This rule is a reflection of the fact that the note type doesn't necessarily correspond to the species rules it is constrained by. For example, a quarter note (which is normally a note of the third species) ligated to a half note in the previous measure will be constrained by a rule of the fourth species thanks to this rule.

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.

$$\forall \rho \in positions(m-1) \quad IsS_0[\rho] \implies (N[\rho] = N[\rho+1]) \quad (6.56)$$

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.

$$species(cp_1) = species(cp_2) = 5 \iff \sum_{i=0}^3 \sum_{j=0}^{m-1} (S(cp_1)[i, j] = S(cp_2)[i, j]) < \frac{s_m}{2} \quad (6.57)$$

Determining the Constraints that Apply to Each Note

$$\forall x \in \{3, 4\}, \forall cst_x \in Constraints(x), \forall V \in Variables(cst_x)$$

$$\left(\bigwedge_{\forall v \in V} IsS_x[v_{pos}] \right) \implies cst_x(V)$$

where $Constraints(x)$ is the set of constraints of the species x ,
and $Variables(cst_x)$ is the set of set of variables concerned by the constraint cst_x ,
and v_{pos} is the position of the v related note in the array N .
(6.58)

And, specifically, to agree with rules **1.H1** and **4.H1** :

$$\begin{aligned} \forall j \in [0, m) \quad IsS_3[0, j] &\implies (H[0, j] \in Cons) \\ \forall j \in [0, m - 1) \quad IsS_4[2, j] &\implies (H[2, j] \in Cons) \end{aligned} \quad (6.59)$$

To fully understand how the **S** array works and how it is parsed, please refer to T. Wafflard's thesis.

6.7 General look back on the set of rules

The following table gives an overview of the configurations in which the different rules apply. The rules that appear multiple times are the rules which have multiple formulations (for different contexts). For example, rule **1.M2** appears once in the two voices row and once in the three and four voices row. This is because it has one formulation that is applied in a two-voice context, and another that is applied in the other contexts.

	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, 3P1	4H1, 4M1, 4M2, 4P1, 4P2	5.R1-R8
2V	1H2, 1H3, 1H7, 1P1	1M2	2H3, 2M1, 2M2	3H4	4H2, 4H3	
2V3V	G6, 1H5					
3V	1H7, 1H8, 1H10, 1P1					
3V4V	1H3, 1M4, 1P6, 1P7	1M2	2M2, 2P3	3H6	4P3, 4P5	5.R9
4V	1H8, 1P1			3H7		

Table 6.1: Table of Scopes

Chapter 7

Program Architecture

This chapter delves into the program architecture of the C++ implementation. The transition from the previous Lisp-based implementation necessitated a great deal of thought throughout the process, with the core structure being re-imagined multiple times as difficulties arose throughout the development of the code. The final code leverages object-oriented programming principles like inheritance and polymorphism, as well as functional abstraction, to create a more maintainable and scalable system capable of tackling the complexities of four-voice counterpoint composition.

Reminder about the two stages of Constraint Programming

As was explained in Chapter 2, but is worth mentioning again, a program for constraint solving written for the Gecode solver is structured into two parts. The first part posts the constraints into the search process. This is monotonic: constraints can only be added, not removed. The second part runs the solver using the constraints that have been added. In FuxCP, the first part follows the structure of the Fux theory of counterpoint. This theory is quite complex in reason of the different configurations possible and the intervention of abstract concepts like strata.

This chapter focuses on the first part, which adds the constraints. This is the most complex part, since it defines all the constraints used in the Fux theory. This part is purely structural: it has a straightforward control flow that monotonically adds constraints to the search process. The structure of the C++ program implementing the first part therefore directly follows the structure of the Fux theory.

The second part is the search process. Once all the constraints have been added, then the solver is run, which defines the search for solutions that satisfy the constraints. This is a separate part of the C++ code, which is explained in the next chapter. It defines the search strategy and the search heuristics. Note that it is possible in the second part to add additional constraints during the search process, which are constraints that have not been defined in the first part. This is important when implementing sophisticated search strategies such as branch-and-bound and its refinements.

7.1 Inheritance-based Structure

The first point to note, before diving into the implemented inheritance-based structure, is that this structure provides an **approximation** of the workings of Fux's theory. In its reality, the theory is much more sophisticated, since it is based on human psychology and creative process, not on a rigorous structure. However, as will be shown in this section, the object-oriented structure greatly simplifies the code and is a good approximation of the complex theory.

7.1.1 Centralizing common structural elements

From the introduction of the very first concepts used in the formalization of Fux's theory, namely voices, parts and strata (see section 4.1), an inheritance-based structure was already implied : parts and strata are instances of a higher concept, voices. It seems natural that a C++ implementation of parts and strata should leverage the structural similarities of these two concepts, factorizing them into one central mother class, *Voice*. This structure can be declined further : a part could be the *cantus firmus*, or a counterpoint of any species.

Making use of the common structural elements is the most obvious motivation for using an inheritance-based structure. However, when delving deeper into the challenges of a theory as complex as the one introduced in *Gradus ad Parnassum*, one finds that this structure can be of great help for other difficulties as well.

7.1.2 Posting the right set of constraints

As was shown in the previous chapter, and summarized in table 6.1, the set of rules to be applied depends mainly on two parameters : species and number of voices. It is no coincidence that *Gradus ad Parnassum* is structured in three parts (two, three, and four voices), with each part containing five chapters, one for every species. In the Lisp implementation, a massive dispatch function was used to handle the creation of counterpoint objects and the application of constraints, manually checking the context (species, number of voices, etc.) for each constraint. This approach, although it is correct, resulted in cumbersome and difficult-to-maintain code.

A much more elegant solution is to use class inheritance to create counterpoints and automatically apply the right constraints in the constructors of these counterpoint objects. Upon the creation of, say, a *SecondSpeciesCounterpoint* object, the constructor of the class first calls the constructor of the mother class, *Part*, which posts the constraints common to all parts. The *SecondSpeciesCounterpoint* constructor then goes on to post second species specific constraints. In this way, we can "isolate" the different contexts without having to manually check (with "if" statements) in which configuration we are.

While this flow of operations is already considerably more elegant than in the Lisp implementation, one problem persists : the set of constraints to be applied depends not only on the species, but also on the number of voices in the composition. A naive solution would be to create a distinct subclass for each combination of species and number of voices, resulting in a proliferation of classes like *FirstSpeciesTwoVoiceCounterpoint*, *SecondSpeciesThreeVoiceCounterpoint*, and so on. This would involve multiple inheritance, because *SecondSpeciesThreeVoiceCounterpoint* would inherit both from *SecondSpeciesCounterpoint* and from *TwoVoiceCounterpoint*. This approach would lead to a bloated, overly complex, and error-prone class hierarchy with multiple dozens of similar sub-classes.

Managing Context with Multiple Constructors

The chosen solution strikes a tradeoff between the Lisp-like approach, which avoids inheritance, and the overly complex approach of creating subclasses for every specific context. This trade-off uses inheritance where it is adequate but not when it increases program complexity. It does so by retaining subclasses for each species while implementing multiple constructors within each species class, tailored to different numbers of voices. Each species class has three constructors: one for two voices, one for three

voices, and one for four voices. Each of these constructors begins by calling a shared, internal constructor (called General Constructor in Figure 7.1) that applies constraints common to all instances of that species of counterpoint, regardless of the number of voices. That way, we isolate all of the contexts without having a distinct class for each one, and we can ensure that the appropriate set of constraints is applied automatically when a new counterpoint object is instantiated. The need for manual checks and conditional logic is eliminated, making the code both more elegant and more robust.

In summary, the final structure handles the two dimensions of context by keeping a declination of classes on the species dimension (for reasons of structural similarity), and deals with the "number of voices" dimension by implementing, for each species subclass, a constructor for each number of voices.

Figure 7.1 presents a UML-style diagram depicting the final structure of the classes that represent voices. Getter functions, basic constructors, and other usual functions like `to_string` and `clone` are omitted to alleviate the diagram. Abstract classes are shown in green.

Note that `SecondSpeciesCounterpoint` and `ThirdSpeciesCounterpoint` are both children of `FirstSpeciesCounterpoint`, instead of `Part`. This is because most of the constraints that apply to first species counterpoints apply to thesis notes in all configurations. The second and third species are the only other species that have notes on all downbeats, so they are children of the first species to illustrate that and avoid code duplication for those constraints.

Note about the creation and linking of Stratum objects : As was explained in section 4.1, strata are an abstract concept that represent the relative order of the pitches of the parts at any given measure. Therefore, they are linked to the parts, and there is a bijection at each measure between the parts and strata. This mapping is quite simply implemented with a sorting constraint, in accordance with the mathematical formulation provided in equation 4.1. This is done in the `setStrata` method in the `CounterpointProblem` class (which will be introduced in the next subsection). This method is general, for any number of voices. As a result, the implementation of the strata does not add additional complexity to the structure.

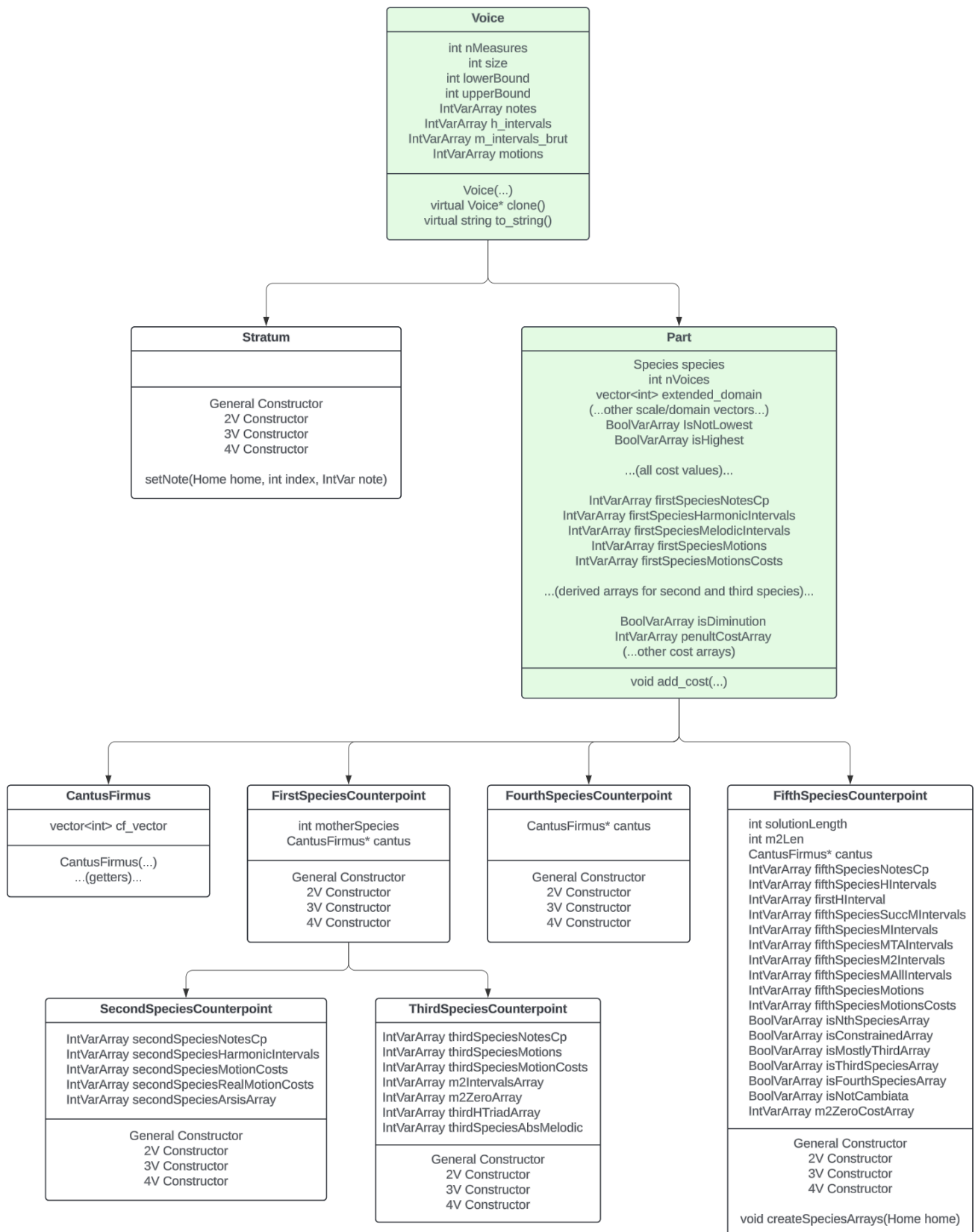


Figure 7.1: UML-style diagram of the Voice class hierarchy.

7.1.3 Hierarchy of the main Problem classes

We now have an optimal structure that takes care of the creation of all of the different individual voices involved, as well as posting the right constraints in an elegant manner. Now one must remember that some constraints apply not to a specific counterpoint, but to the composition as a whole, such as the use of the harmonic triad. These rules depend on the number of voices in the problem. Again, where previous implementations had one centralized Problem object, and manually checked which of these rules should be posted, we can make use of clever inheritance to make the code more elegant and robust. We define an abstract mother class, `CounterpointProblem`, which is a child of the Gecode class `Space` (or more specifically `IntLexMinimizeSpace` or `IntMinimizeSpace` to design the minimization problem). Three classes implement `CounterpointProblem`, namely `TwoVoiceCounterpoint`, `ThreeVoiceCounterpoint` and `FourVoiceCounterpoint`. The following UML-style diagram, Figure 7.2, illustrates this structure.

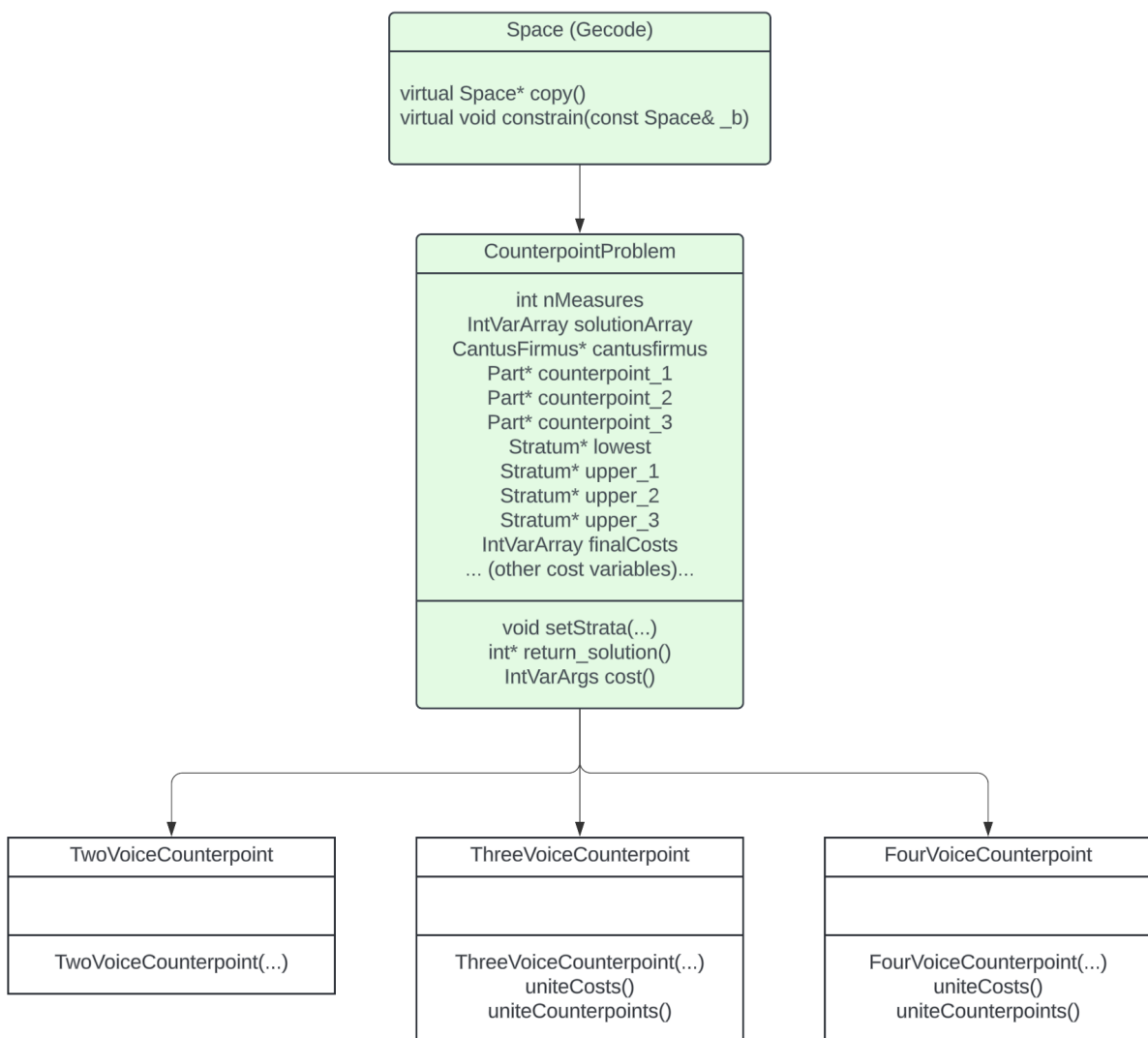


Figure 7.2: UML-style diagram of the `CounterpointProblem` class hierarchy.

Note that polymorphism¹ is used to store the counterpoint objects in the `CounterpointProblem`

¹Polymorphism is an object-oriented programming concept that refers to the ability of a variable, function or object to take on multiple forms. A language that features polymorphism allows developers

class : they are stored as pointers to Part objects, but are instances of a certain species of counterpoint.

7.2 Polymorphism

Another powerful feature of the C++ implementation is the use of polymorphism. In addition to storing species counterpoints as pointers to Part objects, polymorphism is also used through the implementation of virtual functions. For example, the `to_string()` method, which generates a textual representation of a voice, is defined as a virtual method in the Voice base class. Each subclass overrides this method to provide additional details specific to that species or other particular voice, but also calls the base class method to include the general information. This layered approach ensures that the output of the `to_string()` method contains all of the necessary information, and that the information relative to higher classes in the hierarchy is consistent between calls to the method from different subclasses.

Similarly, the `clone()` method is also defined as virtual in the Voice class, and overridden by the subclasses. The overridden method in each subclass ensures that all species-specific properties are copied correctly, while the base class method handles the cloning of universal properties. Accurate cloning of objects is especially important in the context of constraint programming, where Spaces are copied between different steps of the search.

7.3 Functional Abstraction

In addition to inheritance and polymorphism, our C++ implementation makes extensive use of functional abstraction to further simplify the code. Every constraint is encapsulated within a function, allowing it to be easily reused across different contexts. Moreover, constraints that apply between different pairs of parts can be called from different contexts, with the counterpoint objects on which to apply the constraint given as argument.

This approach not only reduces code duplication but also makes it easier to modify or extend the code. If a given rule changes, we only need to update the logic in one place, and the new behavior will automatically be applied wherever that function is called.

7.4 Flow

To clarify the flow happening inside of the software when creating a specific problem, this section provides an example problem and describes the functions called to construct the CSP step by step. Let us take the example of a user creating a three-voice counterpoint problem, with a counterpoint of the fourth species and another of the first species.

1. **The user gives the problem parameters.** This is usually done through Open-Music, but the implementation can also be used straight from C++ (from the `main.cpp` file). If done through OM, these problem parameters are passed through

to access objects of different types through the same interface. [27]

CFFI to the `create_problem` function. When using C++ directly, the same function is called straight away.

2. **The `create_problem` function dispatches the creation of the appropriate `CounterpointProblem` object.** In the case of our example, a `ThreeVoiceCounterpoint` object is created.
3. **The `ThreeVoiceCounterpoint` constructor is invoked.** It takes care of calling the necessary functions to construct the right CSP :
 - (a) First, the `ThreeVoiceCounterpoint` constructor calls the `CounterpointProblem` constructor, which creates the general variables and arrays.
 - (b) It then moves on to create the `CantusFirmus` object, then the counterpoint objects. Each of the counterpoints is created by calling the `create_counterpoint` function with the right arguments, including species and voice range. This function, similarly to the `create_problem` function, calls the right constructor given the context. In this case, the function is called twice :
 - For creating the first species counterpoint object. It calls the appropriate counterpoint constructor : the three-voice constructor of the `FirstSpeciesCounterpoint` class. This constructor begins by calling the general (internal) `FirstSpeciesCounterpoint` constructor, then posts the constraints specific to first species counterpoints in three voice compositions. The general constructor, before posting its constraints, calls the `Part` constructor, which starts by continuing the hierarchy by calling the `Voice` constructor. In other words, each constructor starts by calling its super-constructor, before executing the code of its specificity.
 - For creating the fourth species counterpoint object, the same logic is followed.
 - (c) Finally, the `ThreeVoiceCounterpoint` constructor posts the constraints that apply to the whole three-part composition.
4. **The CSP is now constructed**, and the `CounterpointProblem` pointer is returned by the `create_problem` function. This pointer is passed on to the interface, because it will later be used to start the search for solutions.

7.5 Conclusion

Fux's theory of counterpoint lends itself very well to an object-oriented programming formalization, but its reality stays more complicated and intricate than the object-oriented structure alone. Some constraints have exceptions in some situations, and sometimes these exceptions have to be checked manually, the old way. In the final analysis, the Fux theory is based on human psychology and creative process, which is more complex than object-oriented programming. However, the evolution from the single dispatch function to the elegant structure we now have shows that object-oriented programming provides a good approximation of the reality of Fux's theory.

Chapter 8

Solution Search, Examples and Evaluation

8.1 Verifying that the CSP is correctly defined

Now that we have created an implementation capable of defining a CSP for a given counterpoint composition problem, it is time to verify that it does indeed behave as expected. This involves two parts : first, testing that the solution is sound and complete, and second, verifying that good solutions have lower costs than bad solutions.

8.1.1 Completeness : Accepting Correct solutions

In order to verify that our constraints are as faithful to Fux's examples as possible, we have taken the illustrations given in *Gradus ad Parnassum* and encoded them into test cases. It should be noted that a small number of these tests fail because FuxCP defines a discrete set of voice ranges which are relative to the *cantus firmus* (below, far below, above, ...). This is obviously not a reality of Fux's theory, so occasionally he has a voice that does not fit perfectly within these voice ranges. However, for the vast majority of examples, FuxCP's voice ranges represent Fux's voice ranges well, in which case the encoded tests were an extremely helpful tool to verify that the constraints reflect Fux's intentions.

Figure 8.1 and the following code show an instance of how Fux's examples are implemented as tests. Here we test the two-voice example given in Figure 22 of *Gradus ad Parnassum*. The tests are implemented in the `fuxTest.cpp` file. This file contains a class called `FuxText` with attributes necessary for the individual implementation of each test : the *cantus firmus* used, the counterpoint notes, the species etc. Once the example is encoded, we add it to the dispatcher in the constructor of the class. When we pass the number of the test (and optionally the index up until which note we want to fix the example) as argument, the constructor calls the function and initialises its attributes. Then, in the `main.cpp` file, we call the attributes and fix the variables of the problem. If a solution is found, Fux's example is accepted by the constructed CSP and the test has passed. Nineteen of Fux's examples have been implemented as tests, and there are tested examples for all numbers of voices and all individual species. Fux gives many more examples, so more could be implemented in the future if necessary.



Figure 8.1: Figure 22 of *Gradus ad Parnassum*[11]

```

1 void FuxTest::test_2v_1sp_fig22_setter(int i){
2     cout << "Fig. 22" << 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     idx = i;
8     borrowMode = 1;
9 }

```

Listing 8.1: C++ implementation of Figure 22

8.1.2 Soundness : Rejecting Incorrect Solutions

It is also necessary to make sure that incorrect solutions are not accepted by the program. Therefore, another series of tests is implemented which explicitly fixes an incorrect combination of notes which must be rejected by the program. If the program returns at least one solution, we can be sure that the targeted constraint does not work and that it has to be corrected.

These tests are also implemented in the FuxTest class in fuxTest.cpp. To run them, the number of the constraint must be passed as argument, for example "1H1". The FuxTest class constructor then sets the necessary attributes and calls the function. In these types of functions, a new CounterpointProblem object is created with the correct parameters. Twelve tests targeting important individual constraints are available in the testing framework. All constraints were of course tested when implemented, but some of the tests were not implemented in the testing framework itself, they were just development tests. The following code shows the testing of constraint 1.H1.

```

1 CounterpointProblem* FuxTest::test_1sp_H1(){
2     spList = {FIRST_SPECIES};
3     v_type = {2};
4     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
5     general_params, specific_params,
6     importance, borrowMode);
7     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 72); // Fixing the second
8     note of the counterpoint to a dissonance (the second note of the cantus firmus is 62).
9     return problem;
10 }

```

Listing 8.2: C++ implementation of the test for constraint 1.H1

```

1 FuxTest::FuxTest(char* test){
2
3     cantusFirmus = {60, 62, 65, 64, 67, 65, 64, 62, 60};
4     size = cantusFirmus.size();
5     // [the setting of additional parameters is omitted because they are not relevant to the
6     test.]
7     CounterpointProblem* problem;
8     problem = dispatcher(test);
9     BAB<CounterpointProblem> e(problem);
10    int nb_sol = 0;
11    while(CounterpointProblem* pb = e.next()){
12        nb_sol++;
13        cout << pb->to_string() << endl;
14        delete pb;
15        if (nb_sol >= 1) // this while loop may seem useless here, but is kept in case future
16        tests need to generate more than one solution.
17        break;
18    }
19    if(nb_sol==0){

```

```

18     cout << "This constraint works. It prohibits a forbidden configuration." << endl;
19 } else {
20     cout << "This constraint does NOT work. It allows a forbidden configuration." << endl;
21 }
22
23 }

```

Listing 8.3: FuxTest class constructor for constraint testing

For a complete guide on how to run the tests of the testing framework, please refer to Appendix B.3.

8.2 Searching for Good Solutions

We have verified (to the best of our ability, a complete verification is impossible due to the sometimes vague nature of Fux's work) that the constructed CSP is sound and complete, and that the costs work as intended. It is now time to explore how we can guide the program when navigating through the immense sea of solutions to make it find an acceptable solution in a reasonable amount of time.

8.2.1 Type of Minimization

We have looked into two different types of cost minimization : minimizing the global cost and minimizing in lexicographical order. Both of these different approaches yielded different results when generating several consecutive solutions.

- Minimizing the global cost : in this minimization, we sum all the costs together into one global cost, then minimize it. This type of reduction does not take into account which costs exactly are reduced, it just aims to achieve the lowest possible cost. This mode of minimization can be selected by setting all the costs to the same importance.
- Lexicographical minimization : this is the only type of minimization which takes user preferences into consideration. By passing giving an order of priority for each cost, the user can decide which cost should be minimized with the most importance. The solver then first tries to minimize the cost with the highest priority before minimizing the cost with the second highest priority, etc. When two costs have the same importance, they are added together and treated as one cost. Therefore, if we make all the costs have the same importance we end up with a global cost minimization.

8.2.2 Branching

The branching strategies for four voices has not changed from the three voice strategies. We are branching for all species on the lowest stratum notes and the notes of all the species combined. Then there are special branching strategies added for the fourth and fifth species. For both species, we are also branching on the syncopation cost array in order to make sure that we have a maximum of ligatures in the composition. Then, for the fifth species, branching is also done on the cambiata cost array and the species array.

8.3 Using Next Solution or Best Solution

In the previous sections we have verified the constraints and explored the different ways of minimizing costs. Now let us have a look at the different ways of generating

solutions. The FuxCP solver is equipped with two different functionalities : next solution and best solution (more details about the interface are given in Appendix B). Next solution will immediately return the next best minimization found, which usually is only a small reduction in one cost. Best solution on the other hand will loop on the next solution until it is stopped manually (or runs out of better solutions to generate). Once it is stopped, it will return a solution which was minimized for the amount of time that it ran.

Let us take a look at a simple first species counterpoint that the solver returned upon pressing next solution. We see that the solution is very basic. There are many half tones, many notes are borrowed off-scale, and there are in general many musical concepts present which Fux discourages. All of this happens because the branch and bound algorithm was not able to explore the branches yet. What happens is that we get the lowest possible integer value for each note which is inside the range allowed by the constraints.

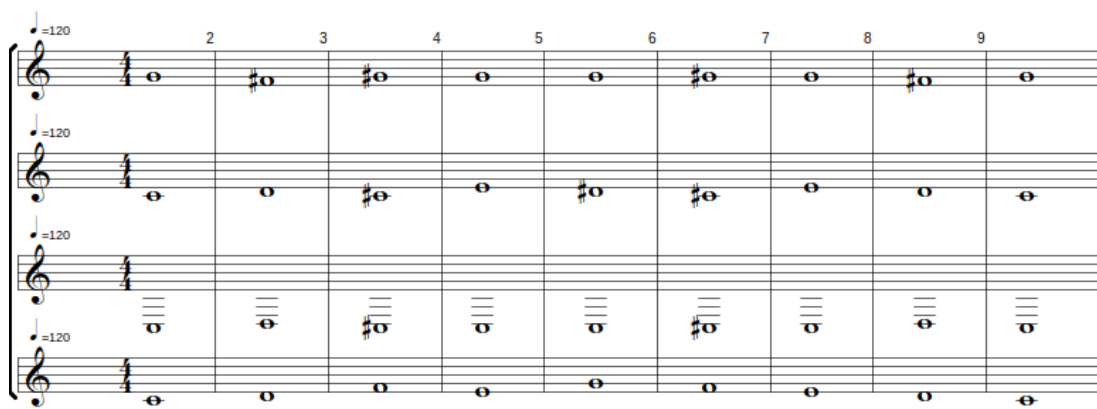


Figure 8.2: First species counterpoint generated through next solution

The next solution was generated after more run time. We can see that even though we are still far from an optimal result, the example is more optimized. However, the number of borrowed notes is still too high. We can deduce that this cost must be set higher, which will be done for the rest of this evaluation. This kind of reasoning is exactly how FuxCP is meant to be used : if a solution is not to a composer's liking because of some preference being ignored too often, the cost corresponding to that preference has to be set higher.

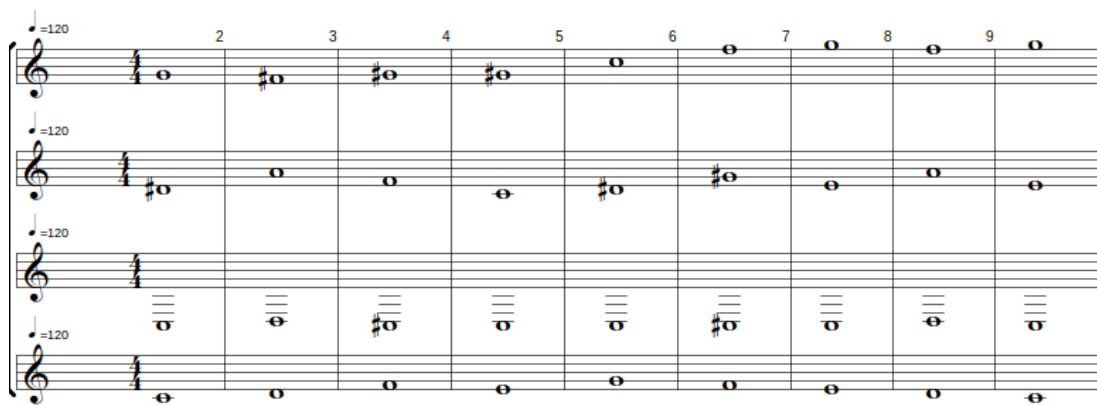


Figure 8.3: First species counterpoint generated through best solution

8.4 Evaluation by species

This section's goal is to generate solutions by using the solver and analyse them by keeping Fux's principles in mind. It is important to take into consideration that the solver is not yet well optimized, so the solutions can take quite a while to be generated. This drawback of the solver can particularly be seen in four voice generation, as some possible solutions can take tens of minutes before being found. Therefore, the following examples are not as close to Fux's examples as we would have liked, but as we tested the constraints separately beforehand, we can still be sure that they are correct according to his principles. For each species, all of the other counterpoints are of the first species, to give the same setting as the examples of *Gradus ad Parnassum*. All of the evaluations had the following voice types : *really far above*, *above*, *below*, and the costs were the default costs as given by the CP-PARAMS interface (except for the borrow cost which was set higher, as mentioned in the previous section). The *cantus firmus* that was used for all of the evaluations is the following :

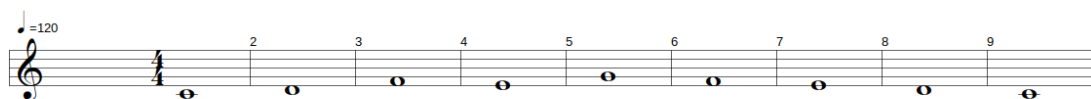


Figure 8.4: *Cantus firmus* used for evaluation

8.4.1 First species evaluation



Figure 8.5: Example of a first species counterpoint in four part composition as generated by the solver (Click [here](#) to listen to the example)

We find that this first species already shows an advantage of four part composition : even with basic notes and melodies it is already quite rich. The added layer that four part composition brings to the table when compared to three part composition really helps to make the first species more interesting to listen to. We see that, just like Fux, the solver also decided to repeat a note three times in one counterpoint. Theoretically this goes against the minimization of the variety cost, however it does not have that big of a negative impact on the music overall because of the movements of the other parts.

When taking a look at the costs of this solution, we see that, overall, they are relatively low, though there is one cost which stands out as being high : the cost for reaching a perfect consonance through direct motion. Here we must remember that it is impossible to minimize all of the costs at once, which is precisely why preferences are set.

8.4.2 Second species evaluation

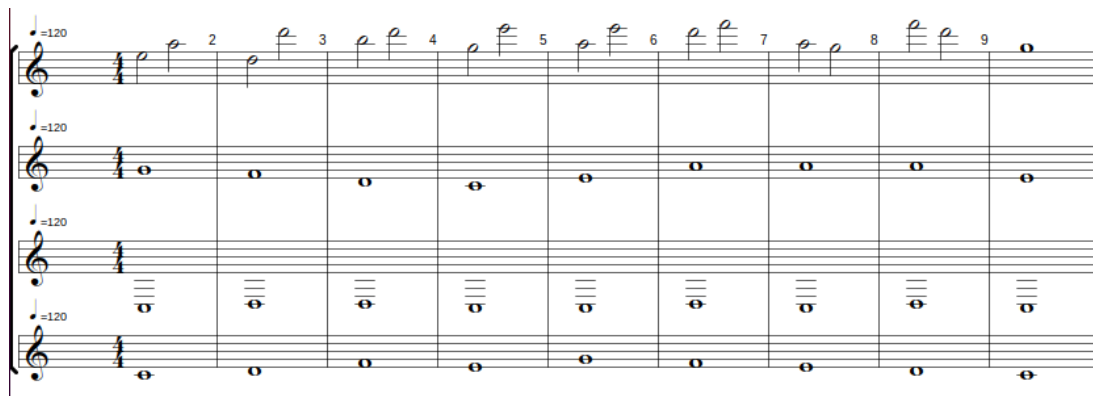


Figure 8.6: Example of a second species counterpoint in four part composition as generated by the solver (Click [here](#) to listen to the example)

Comparing this solution with the Fux examples brings out one notable difference : the triad is not used as much as would be liked by Fux. This is due to the solver not having the time to optimise it. As with the other solutions, the variety is not as rich as with the examples of Fux, but it is to be expected at this point. It also seems that the examples in *Gradus ad Parnassum* show more coherence and more fluid motions across the measures. In this solution, there are constant spikes between the thesis and arsis notes. This contrasts with Fux's examples, which are more fluid.

8.4.3 Third species evaluation



Figure 8.7: Example of a third species counterpoint in four part composition as generated by the solver (Click [here](#) to listen to the example)

This solution brings the same lack of triad with it as we saw in the second species evaluation. All things considered it is a pretty decent solution that was generated, though it does lack some finesse. Noticeably the third measure is not musically interesting

and the repetition of the C and D notes hinder the solution quality. Some of the big leaps in melodic intervals also seem slightly jarring.

8.4.4 Fourth species evaluation



Figure 8.8: Example of a fourth species counterpoint in four part composition as generated by the solver ([Click here to listen to the example](#))

The fourth species counterpoint that the solver generated is overall satisfactory. It does suffer from the same drawbacks as some previous examples, like a lack in triads and variety, but other than that it holds up well compared to what Fux shows for his fourth species. As discussed previously, the solution does not include the doubling of the notes in the first species as Fux suggests.

8.4.5 Fifth species evaluation



Figure 8.9: Example of a fifth species counterpoint in four part composition as generated by the solver ([Click here to listen to the example](#))

Rhythmically, the fifth species is the highlight of *Gradus ad Parnassum*. The solver gives decent results, but it lacks the musical ear of a good composer to make the composition truly interesting. It falls flat in some areas, and the solver does not bring out its full potential. Here again some personal preferences could be able to return better results, but, to take a look at where the solver fails to reproduce the musicality of Fux, we kept his preferences in place.

8.5 Combining different species

For this chapter we used the solver to generate some counterpoints which show some of the many species combinations which are possible with the solver. The amount of possibilities using four voices has drastically increased compared to the previous versions of FuxCP. Here again we find that the more creative we try to become with the combinations of species, the higher the risk that, musically, the results which the solver returns are going to be less interesting.

The following illustrations are a couple of solutions obtained by experimenting with different species combinations.

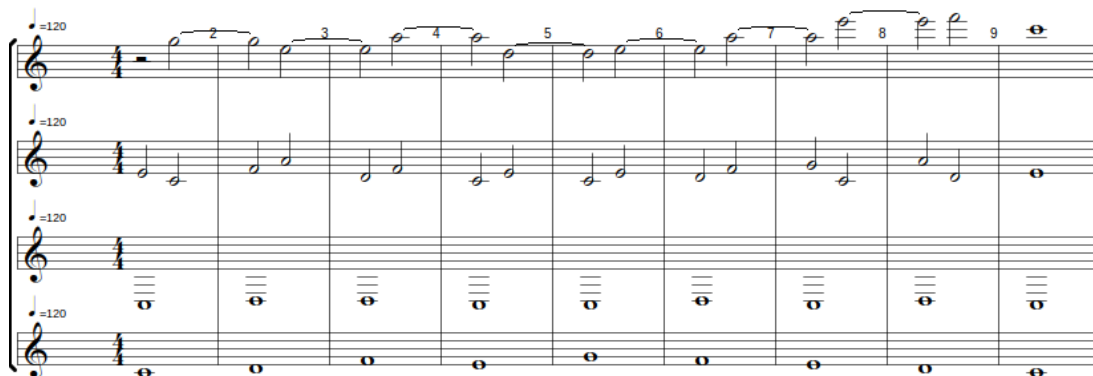


Figure 8.10: 4th, 2nd and 1st species counterpoints ([Click here to listen to the example](#))

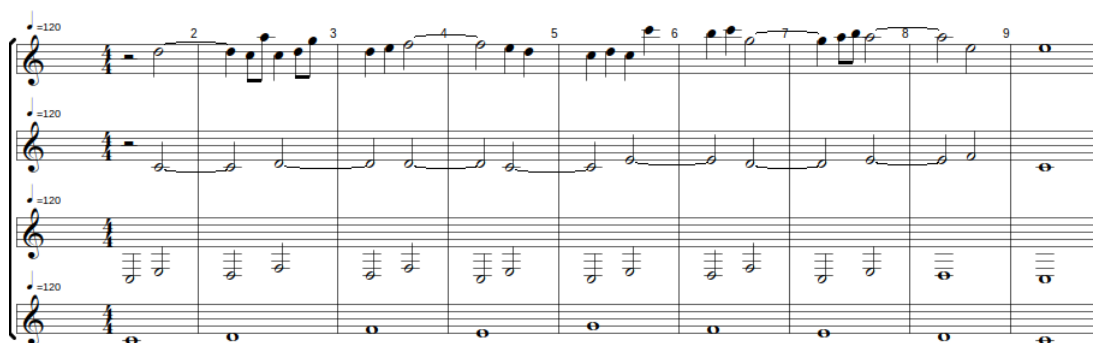


Figure 8.11: 5th, 4th and 2nd species counterpoints ([Click here to listen to the example](#)). **Note** : The eighth note skips in the second measure resulted from a bug that has since been fixed. Eighth notes are only used to add mordent or fill in third skips, so they always constitute steps, never leaps.

8.6 Conclusion

Analyzing these examples has shown us that our model is capable of generating interesting results, although in general, they fall short of great composition. These results show that while Fux's strict rules are followed, further optimization and branching heuristics are required to fully take into account the preferences, by achieving a better minimization of the costs.

Chapter 9

Obstacles Encountered, Known Issues and Future Improvements

As all developers know, a complex project almost inevitably comes with its share of unforeseen challenges and limitations. This chapter lays out the obstacles encountered during the development of FuxCP, the known issues that remain unresolved, and the potential avenues for future improvements. By documenting these aspects, we not only provide a transparent account of the difficulties faced but also lay the groundwork for further advancements in the project. The insights presented here are meant to guide future researchers and developers who wish to build upon this work, ensuring that the lessons learned are not lost and that the system can continue to evolve towards its full potential.

9.1 Obstacles encountered during the work on this thesis

"If you run into difficulties, as can hardly be avoided, understand that the way to perfection is through hardship. Without an adversary one cannot be victorious, and for the contest one must arm oneself with unflagging zeal and steadfastness and patience." - Johann Joseph Fux Mann [11, p. 113]

Throughout the course of this project, as with any valuable endeavor, obstacles and difficulties arose. This section details the most significant challenges faced. By understanding these challenges, future developers can better prepare for similar issues and avoid potential pitfalls.

- **Thoroughly understanding the tools, previous software and theses.** This first challenge was not unforeseen, but simply bigger than anticipated. Before starting any work of our own, it was necessary to thoroughly understand the previous theses and software we would work with. Being unfamiliar with Lisp, OpenMusic, and Gecode, the time needed to get used to these complex tools and languages was more than anticipated. Moreover, the previous work was divided into two theses, and the rules and code that were introduced in both did not always blend together seamlessly. Thankfully, the fact that we had to completely re-write the code forced us to generalize it, so future programmers continuing this project will not face this issue.
- **Unifying the theoretical formalization framework and the complete set of rules** was, in the beginning, not an objective of this work. When starting the migration from Lisp to C++, the methodology was to follow the calls to the GiL Library and translate them directly into Gecode, keeping the previous constraints as they were. It soon became clear that some of these constraints were not written to be general and to apply to broader context, and that the formalization framework had to be rethought in some aspects. While understanding every

previous rule to its core and questioning its implementation was time consuming, this process brought us to understand Fux's intentions more broadly, which in turn resulted in sharper insight when formalizing the new, four-voice rules.

- **Re-writing the code to implement a full inheritance-based structure.** The initial C++ translation kept the same style as the Lisp implementation when it came to dispatching counterpoint creation and constraint posting. As was explained in Chapter 7, this was a naive implementation that did not leverage the newly accessible paradigms of C++. As a result, the code had to be completely re-written although our implementation at the time was already functional for the first three species.

In summary, facing these obstacles did not come without a reward. Each major challenge forced us to have a broader understanding of Fux's theory and its formalization, and without them this work would not be as complete.

9.2 Known Issues

Although the vast majority of bugs and issues were resolved with careful coding and testing, some minor issues remain. These issues are only problems in specific configurations, and do not prevent the use of FuxCP.

- **Rule 4.P6 is not implemented in the current version of FuxCP.** The reason for this is that this rule is an *a posteriori* rule. This means that it is taken into account *after* a solution is generated by the solver, so it is not implemented as a constraint of the CSP. In the previous implementation, the rule was implemented in the parsing function that took the solution values and created the OpenMusic voice objects. This was possible because since the CSP was completely defined from Lisp, the rule could access all of the variables used during the search. In our new implementation, this would theoretically be possible, but would involve adding new CFFI functions to the Lisp/C++ interface, which goes against the principle behind migrating the entire code to C++. Instead, the rule should be implemented directly in C++, but the Lisp parsing functions would have to be re-imagined to accommodate these changes.
- **The method of cost minimization (linear combination, maximum minimization) cannot be toggled from OpenMusic.** However, we have left the button on the interface because it is an important part of the cost analysis that has to be done in future improvements.
- **About potential minor OpenMusic bugs.** It must be noted that the new version of FuxCP underwent more extensive testing through the C++ interface than through the OpenMusic user interface. Therefore, some bugs might be left for some edge cases when using FuxCP through OpenMusic, but for the vast majority of cases there should not be any problems.

9.3 Future Improvements

9.3.1 Solution Search

- **Cost analysis :** Although the model assigns costs correctly for each preference (lower costs correspond to better solutions in that aspect), the relative importance of these costs should be further analyzed to determine which preferences

Fux would have given. How important is variety compared to using harmonic triads? How important is it to use steps instead of leaps as often as possible? These are questions that should be analyzed using Fux's examples coupled with musical reasoning.

- **Constraint Optimization :** While constraints were encoded with efficiency kept in mind, it was not the primary objective. Some constraints can still be encoded in a more optimal manner. For example, the variety preference constraint could be re-imagined to use a global cardinality constraint.
- **Heuristics :** Even with a more optimal encoding of the constraints speeding up the search, it would be impossible for the solver to search through every solution. Even for a four-voice composition comprised of only whole notes (three first-species counterpoints) and with a *cantus firmus* of length 9, the solution space already surpasses the tens of millions of solutions. Using a branch and bound algorithm is far from enough to find a near-optimal solution in a reasonable amount of time. Therefore, heuristics must be defined to guide the search towards the better solutions.

9.3.2 FuxCP User Experience

- **Estimating the distance to the best solution :** as mentioned above when talking about heuristics, finding the optimal solution seems out of reach for many configurations. To improve user experience in these cases, a distance function could be implemented, estimating the global cost of the best solution and showing the user how far the current solution is from this estimation. This function could be estimated using the costs of Fux's examples, and generalized to other *cantus firmus* lengths and configurations by incorporating these parameters in the formula.
- **Enabling the composer to give musical ideas as additional constraints to FuxCP** Another way to combat the computational complexity, and at the same time to provide more features to composers, is to allow them to suggest musical ideas to the system. For example, if a composer already knows which motions they want at given measures, they could provide that information to FuxCP, which would then force these motions as new constraints, thereby reducing the solution space.
- **FuxCP as a tool to validate counterpoint compositions** A final idea to make full use of the defined constraint system is to allow FuxCP to also be a validation tool, instead of just a generation tool. Composers wishing to evaluate their compositions with respect to the constraints and preferences of Fux's theory could input these compositions, and the solver would output whether the solution was accepted by the constraints, and if so, show its different cost values.

9.3.3 Beyond *Gradus ad Parnassum*

As explained above, a few little tweaks are required to fully finish the formalization of *Gradus ad Parnassum*: implementing the *a posteriori* rule **4.P6** directly in C++ and analyzing which costs Fux would have prioritized. It could also be decided to implement the rule about dividing a whole note when composing in the fourth species, but as explained in Chapter 5, we believe this rule to be superfluous as it only serves in the apprenticeship context of composing with only whole notes in the other parts.

With *Gradus ad Parnassum* then being completely formalized, other rules from other works could also be included, broadening the system to more than just Fux's theory.

Chapter 10

Conclusion

This thesis, building upon the works of Thibault Wafflard and Anton Lamotte, set out to further advance the work of formalizing Fux's *Gradus ad Parnassum*. More specifically, the goal of this thesis was twofold : to migrate the FuxCP implementation to C++, and to extend it to allow four-voice composition, with all of the new rules and considerations this implies. We are pleased to conclude that these two objectives have been fulfilled.

Along the way, some unexpected additional tasks arose, such as the generalization of the complete set of rules and a full reimagining of the architecture of an already functional program. These additional tasks necessitated careful thought and rigorous methodology, but were essential to ensure that this work provides a solid and unified foundation for future expansion. To further strive for that goal, a testing framework was implemented for the first time in FuxCP, to ensure that the model does in fact reflect Fux's intentions.

While we can reflect positively on the progress made, challenges remain to make FuxCP an effective tool for composing counterpoint. Constraint programming involves two key tasks : modelization, then optimization. This thesis has finished the task of modelizing Fux's theory, translating it from a dialogue written in natural language in the 18th century to a comprehensive set of rigorous mathematical rules, and finally to a constraint programming model in Gecode. Now, the second task remains : optimizing the search for solutions. The vast solution space in four-voice counterpoint, with its rich diversity of possibilities, poses a significant challenge for the solver, which cannot feasibly explore every option. Future work will need to focus on defining heuristics and analyzing solution costs to guide the solver towards better solutions.

In summary, this work represents a significant step toward developing a tool fully capable of composing high-quality counterpoint in the style of Fux by establishing a complete and well-defined constraint model. The next step is now to guide the solver through the search space with clever heuristics, drawing on both musical and mathematical reasoning, and optimizing the constraints' encoding to enhance search efficiency.

In closing, we feel a deep sense of gratitude to have contributed to this project, and hope to see it continue to grow in the coming years. It is an honor to play even a small part in the endless human pursuit of beauty, through science and art, and to add our efforts to the eternal journey of understanding and creating harmony.

Bibliography

- [1] Thibault Wafflard. “FuxCP: a constraint programming based tool formalizing Fux’s musical theory of counterpoint”. Prom. by Peter Van Roy. MA thesis. École polytechnique de Louvain, Université catholique de Louvain, 2023. URL: <http://hdl.handle.net/2078.1/thesis:40739>.
- [2] Anton Lamotte. “FuxCP; Constraint Programming Formalisation of Three-Voice Counterpoint According to Fux.” Prom. by Peter Van Roy. MA thesis. École polytechnique de Louvain, Université catholique de Louvain, 2024. URL: <http://hdl.handle.net/2078.1/thesis:43960>.
- [3] IRCAM STMS Lab. *OpenMusic*. Version 7.2. URL: <https://openmusic-project.github.io>.
- [4] Guido Tack and Mikael Zayenz Lagerkvist. *Generic Constraint Development Environment*. Version 6.2.0. Gecode. URL: <https://www.gecode.org/index.html>.
- [5] Damien Sprockeels. *Gecode interface Lisp*. GitHub. May 3, 2023. URL: <https://github.com/sprockeelsd/GiL>.
- [6] Guerino Mazzola, Maria Mannone, and Yan Pang. “Short History of the Relation Between Mathematics and Music”. In: *Cool Math for Hot Music: A First Introduction to Mathematics for Music Theorists*. Cham: Springer International Publishing, 2016, pp. 5–34. ISBN: 978-3-319-42937-3. DOI: 10.1007/978-3-319-42937-3_2. URL: https://doi.org/10.1007/978-3-319-42937-3_2.
- [7] Francois Pachet and Pierre Roy. “Harmonisation automatique et programmation par contraintes”. In: (Jan. 2001).
- [8] *Tonality Harmony, Scales Keys* | *Britannica*. en. URL: <https://www.britannica.com/art/tonality>.
- [9] Roland John Jackson. *counterpoint*. URL: <https://www.britannica.com/art/counterpoint-music>.
- [10] *Johann Joseph Fux: a concise biography*. URL: <https://baroquemusic.org/biofux.html>.
- [11] Alfred Mann. *The Study of Counterpoint. From Johann Joseph Fux’s Gradus ad Parnassum*. English. Ed. and trans. Latin by Alfred Mann. Revised Edition. 500 Fifth Avenue, New York, N.Y. 10110: W.W. Norton & Company, 1971. ISBN: 0-393-00277-2. URL: http://www.opus28.co.uk/Fux_Gradus.pdf.
- [12] Jean-Pierre Briot and François Pachet. “Deep learning for music generation: challenges and directions”. In: *Neural Computing and Applications* (2020). URL: <https://doi.org/10.1007/s00521-018-3813-6>.
- [13] Bill Schottstaedt. *Automatic Species Counterpoint*. Research Report STAN-M-19. Departement of Music, Stanford University, 1984. URL: <https://ccrma.stanford.edu/files/papers/stanm19.pdf>.
- [14] John Polito, Jason M Daida, and Tommaso F Bersano-Begey. “Musica ex machina: Composing 16th-century counterpoint with genetic programming and symbiosis”. In: *Evolutionary Programming VI: 6th International Conference, EP97 Indianapolis, Indiana, USA, April 13–16, 1997 Proceedings* 6. Springer. 1997, pp. 113–123.

- [15] Andres Garay Acevedo. “Fugue composition with counterpoint melody generation using genetic algorithms”. In: *International symposium on computer music modeling and retrieval*. Springer. 2004, pp. 96–106.
- [16] Gabriel Aguilera et al. “Automated generation of contrapuntal musical compositions using probabilistic logic in Derive”. In: *Mathematics and Computers in Simulation* 80.6 (2010), pp. 1200–1211. ISSN: 0378-4754. DOI: <https://doi.org/10.1016/j.matcom.2009.04.012>.
- [17] Dorien Herremans and Kenneth Sörensen. “Composing first species counterpoint with a variable neighbourhood search algorithm”. In: *Journal of Mathematics and the Arts* 6.4 (2012), pp. 169–189. DOI: [10.1080/17513472.2012.738554](https://doi.org/10.1080/17513472.2012.738554).
- [18] Maciej Komosinski and Piotr Szachewicz. “Automatic species counterpoint composition by means of the dominance relation”. In: *Journal of Mathematics and Music* 9.1 (2015), pp. 75–94. DOI: <https://doi.org/10.1080/17459737.2014.935816>.
- [19] Ars Nova Software. *Counterpointer*. Developed and produced by Ars Nova Software. Address: PO Box 3333, Kirkland, WA 98083-3333, USA. Available on Microsoft Store and Mac App Store. 2019. URL: <https://www.ars-nova.com/counterpointer3.html>.
- [20] Alex Ding, En-Hua Holtz, John Chung, and Orion Bloomfield. *Counterpointer*. Project for Brown University’s CSCI 0320 course. 2021. URL: <https://github.com/counter-pointer/counterpointer>.
- [21] Peter van Beek Francesca Rossi and Toby Walsh. *Handbook of Constraint Programming*. Ed. by Elsevier. First Edition. 2006. URL: https://www.dcs.gla.ac.uk/~pat/cpM/papers/CP_Handbook-20060315-final.pdf.
- [22] IRCAM. *IRCAM - Institut de recherche et coordination acoustique/musique*. IRCAM official website. <https://www.ircam.fr/>.
- [23] Coralie Diatkine. *OpenMusic Documentation*. IRCAM. June 21, 2011. URL: <https://support.ircam.fr/docs/om/om6-manual/co/OM-Documentation.html>.
- [24] James Bielman and Luís Oliveira. *CFFI: The Common Foreign Function Interface*. 2005. URL: <https://cffi.common-lisp.dev/>.
- [25] Mickaël Tits. *Development of an optical motion capture setup for feature extraction and statistical analysis of the pianist’s expert gestures*. Nov. 2015. DOI: [10.13140/RG.2.1.1015.4961](https://doi.org/10.13140/RG.2.1.1015.4961). URL: https://www.researchgate.net/publication/283460243_Master_Thesis_Mickael_Tits_2014_-_Development_of_an_optical_motion_capture_setup_for_feature_extraction_and_statistical_analysis_of_the_pianist's_expert_gestures.
- [26] Frank Herrlinger. *The Harmonic Series The Circle of Fifths*. 2021. URL: <https://musicintervaltheory.academy/learn-how-to-write-music/harmonic-series-circle-of-fifths/>.
- [27] Margaret Rouse. *What is polymorphism in programming? - definition from Techopedia*. 2023. URL: <https://www.techopedia.com/definition/28106/polymorphism-general-programming>.

On the Use of AI Tools in Writing This Manuscript

In the writing of this manuscript, ChatGPT was used as a tool to revise and refine the text. Its primary function was not scientific reasoning or generating new ideas, but

rather to rephrase and extend existing ideas to convey information more clearly. All content presented in this paper is either our original work or is properly referenced.

Appendix A

Constants and Functions used in the mathematical formalization

Constants

- **species**(p)¹ - the species of part p — by definition, $species(cf) = 0$,
- **n**(p) - the number of notes in part p ,
- **s_m**(p) - the maximum number of notes contained in part p , if all notes were quarter notes, excepted the last one (that is always a whole note),
- **lb**(p) - the lower bound of the range of part p ,
- **ub**(p) - the upper bound of the range of part p ,
- **$\mathcal{R}$** (p) - the range of part p , i.e. $\mathcal{R}(p) := [lb(p), ub(p)]$,
- **borrow**(p) - the borrowing scale² of part p ,
- **$\mathcal{N}$** (p) - the borrowed notes of part p , where $\mathcal{N}^{\mathcal{R}}(p) = \mathcal{N} \cap \mathcal{R}$
- **$\mathcal{B}$** (p) - the set of beats on which a note is played³ according to the species of part p ,
- **b**(p) - the number of beats⁴ in a measure according to the species of part p ,
- **d**(p) - the duration of a note⁵ according to the species of part p ,
- **m** - the number of measures in the composition, which is the same for all voices,
- **Dis** - the set of all dissonances, i.e. $\{1, 2, 5, 6, 10, 11\}$
- **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. $\{0, 3, 4, 7\}$.

¹This has nothing to do with $S(p)$. This constant describes the species of the whole part. If p is a fifth species counterpoint, then $species(p) = 5$

²Remember that Fux mainly uses notes without a flat or sharp, which means that if the composition begins with a C, it will be written in Ionian mode, if it begins with a G, in Mixolydian mode, etc. Borrowing mode is a way for counterpoint to access notes that are not originally in its mode. For example, a counterpoint in E (i.e. Phrygian mode) that has the major mode (E major, i.e. Ionian E) as its borrowing mode will also have access to the notes F#, C#, G# and D#.

³To make it clearer: 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: $\{0, 1, 2, 3\}$.

⁴Thus, it is always equal to the size of the set $\mathcal{B}(p)$.

⁵For the first species, it is equal to 1, as each note is a whole note. For the second species, it is $\frac{1}{2}$, for the third, it is $\frac{1}{4}$, for the fourth, it is $\frac{1}{2}$, and for the fifth, it is $\frac{1}{4}$. It is always equal to $\frac{1}{b(p)}$.

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

$$\begin{aligned}
\mathcal{N}_{key} &:= buildScale(key, scale) \\
\mathcal{N}_{brw} &:= \begin{cases} \emptyset & \text{if } borrow = none \\ buildScale(Cf[0] \bmod 12, "borrowed") & \text{if } borrow = major \\ buildScale([Cf[0] + 3] \bmod 12, "borrowed") & \text{if } borrow = minor \end{cases} \\
\mathcal{N}_{all} &:= \mathcal{N}_{key} \cup \mathcal{N}_{brw} \\
\mathcal{N}_{(all, key, brw)}^{\mathcal{R}} &:= \mathcal{N}_{(all, key, brw)} \cap \mathcal{R}
\end{aligned} \tag{A.1}$$

Functions

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

$$nextm(x) = \begin{cases} 1 + nextm(x - 4) & \text{if } x \geq 4 \\ 0 & \text{otherwise} \end{cases} \tag{A.2}$$

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

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

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

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

$$\begin{aligned}
positions(upto) &= \bigcup_{\forall i \in \mathcal{B}, \forall j \in [0, upto)} [i, j] \\
&\text{s.t. } \forall x \in [1, 3], \forall y \in [1, upto) \\
&\quad [i, j] <_s [i + x, j] <_s [i, j + y] \\
&\text{where } <_s \text{ means the sorting order}
\end{aligned} \tag{A.4}$$

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

$$\forall z \in \mathbb{N}^+, \forall \rho = [i, j] \in \text{positions}(\text{upto})$$

$$\rho + z = [i + zd, j + \text{nextm}(i + zd)]$$

where $\text{nextm}()$ is a function that returns the correct number of measure(s) to add.
(A.5)

Appendix B

User Guide

This manual provides an overview of FuxCP, covering the installation process, its use within OpenMusic and a description of the costs displayed in the interface. Although FuxCP is designed to be compatible with all platforms, it relies on GiL, which was developed for MacOS and Linux. Unfortunately, GiL was not developed for Windows due to compatibility issues between the 32-bit Lisp licence used by OpenMusic and the 64-bit Gecode Windows version. Although a 64-bit version of OpenMusic is now available, the complete installation has not been tested on Windows.

B.1 Installing FuxCP

B.1.1 Prerequisites

To use FuxCP you need to download and install the following tools:

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

And download the following library

- FuxCP : <https://github.com/diegodepatoul/FuxCP4>

B.1.2 Loading FuxCP in OpenMusic

In order to use the above libraries, OpenMusic must be running. When opening any workspace, locate the toolbar at the top of the interface. Click on the "Windows" button, highlighted in the figure B.1, and select "Library" from the drop-down menu. This will bring up a new window. From the toolbar of this window, select 'File' and then 'Add Remote Library'. Navigate through your file system to find the path where the previously downloaded FuxCP library is stored. Once located, click on the FuxCP folder and click "open". The library should then appear under the "Libraries" folder in the "Library" window, as shown in Figure B.2. Right click on "fuxcp" and select "Load Library". If no errors occur, the setup is complete.

However, if an error occurs, it may be a linking problem with the Gecode library. For MacOS users, a script from the `c++` folder (inside the FuxCP folder) can be used. Edit the path to Gecode within the script to match your system configuration. Linux users should add the Gecode library to the `LD_LIBRARY_PATH` variable. Go to the `/etc/ld.so.conf.d` folder and create a new `.conf` file if one does not already exist. In this file, add the full path to the Gecode library, save it, and run `sudo ldconfig` to update the system with the new library. In the `c++` folder, run `make so` for Linux or `make dylib` for Mac to ensure that the libraries are compiled. Don't forget to restart OpenMusic and don't lose hope. Following these steps should ensure that FuxCP works properly.

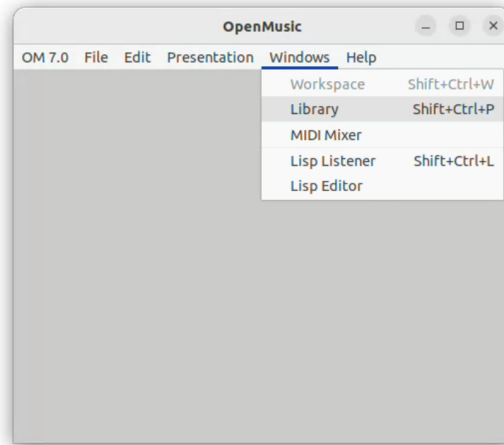


Figure B.1: Opening the "Library" window in OpenMusic.

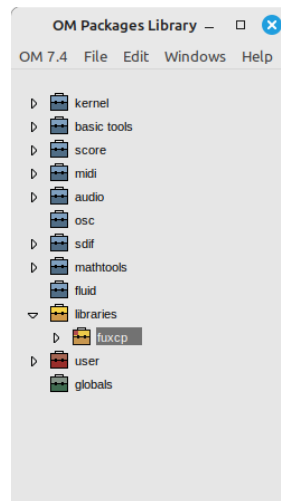


Figure B.2: Loading the "fuxcp" library in OpenMusic.

B.2 Using FuxCP in OpenMusic

Setup

Using FuxCP in OpenMusic is straightforward. There is a single block that contains the entire graphical interface of the tool. This block or class is called `cp-params`.

Using the example patch An example patch, `FuxCP_4v_example.omp`, is provided in the 'Example Patches' folder on GitHub. This patch is the same as the one shown in the figure B.3. To use it, right-click anywhere in the OpenMusic workspace and select *ImportFile*. Select the sample patch and double-click to open it. Voilà, ready to use!

Setting up your own patch If for some reason the example patch is not available, you can set up your own patch as follows. Right-click anywhere in the OpenMusic workspace and select *New...* → *NewPatch*. Double-click your patch. Once in your patch, right-click anywhere in the patch and select *Classes* → *Libraries* → *FuxCP* → *Solver* → *CP – PARAMS*. Alternatively you can just double click anywhere in the patch, type `"fuxcp::cp-params"` and press enter. This also works for "poly", "voice"

and "x-append".

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

If you want to get the whole composition in one object, you have to do some fiddling with OpenMusic. The simplest way to do this is shown in Figure B.3, and works as follows: get the POLY object returned by `CP-PARAMS` on its third output, split this object in three (if composing three counterpoints), then get the *cantus firmus*, which is the second output of `CP-PARAMS`, and put all the voices back together in the desired order using `x-append` functions.

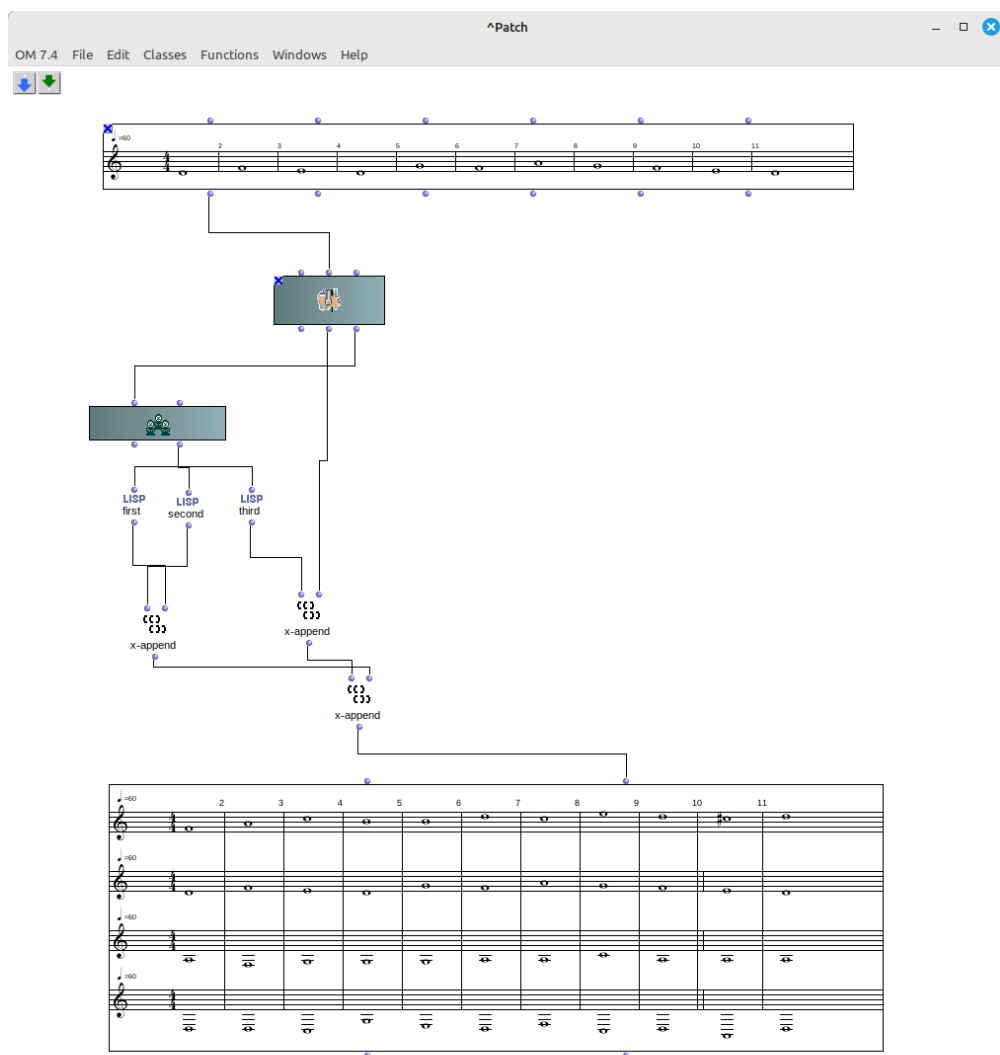


Figure B.3: View of a patch using `fuxcp : cp-params` in OpenMusic.

Listening to the solution

OpenMusic does not have built-in sound. You will need to use a third party application to listen to the result. Here is a solution that works to listen to the music: having installed TiMidity++¹, run the following command before opening OpenMusic:

```
timidity -iA -B2,8 -Os
```

Then go to *OpenMusic* → *Preferences* → *MIDI* → *Ports setup* → *Output devices* and select "TiMidity port 0".

Use the interface

But how do you use the interface? Simply double-click on the block to bring it up. The interface is sorted from left to right, so the preferences are divided into several categories: "General preferences", "Melodic preferences", "Species-Specific Preferences", "Solver Configuration" and, in the bottom right corner, "Solver Launcher" (see figure B.4).

Choose the preferences You will notice that there are almost always two settings for each preference. The first is the importance: it corresponds to the priority the solver will give to reducing the value of this cost. An importance of 1 means that it will be the absolute priority of the solver, whereas an importance of 14 means that the solver will minimise this cost if it doesn't affect the other costs. The second setting is the value: it determines the actual value of the cost corresponding to the preference. It is very useful when two preferences are set to have the same importance, in which case their respective cost values will have an effect. For example, if cost *A* and cost *B* both have an importance of 1, but cost *A* has a very high cost compared to *B*, *A* will affect the quality of the solution more than *B*, even though they both have the same importance. If two costs have the same importance, the yellow panel (bottom left) allows you to choose how to combine them: either by linear combination or by maximum minimisation. The default costs are supposed to represent Fux's preferences. However, please remember that the full cost analysis was not part of the scope of this thesis.

Start the search Pressing the "Next Solution" button will display the solution as a pop-up. What appears on the screen are the counterpoints. The *cantus firmus* must be added manually by using the 'x-append' functions. To see the complete composition, you must evaluate the 'poly' object. To do so, right click on it, and click 'evaluate'. Alternatively, you can click on it and press 'v'.

The other option is to press the "Best Solution" button. This will start a search that will only stop when the best solution has been found (which can take hours). You can evaluate the output object at any time to see what the best result is so far, and this will not stop the search, so you can see how the solution improves step by step, and stop the search when it has produced something you are happy with. After stopping the search, if the search has not finished, you can press "next solution" again to move solution by solution once again.

The button "Stop" allows you to stop the search. This button can take a few seconds to actually stop the search. If a search takes too long, we recommend you to stop the search, change the voice range and start again. Please note that the preferences do not affect the speed of finding the first solution. The first solution is the first valid solution and is not affected by the costs. Only the subsequent solutions can be affected by

¹TiMidity++ is a software synthesizer that can play MIDI files without a hardware synthesizer, click here to install.

the preferences. Therefore, for satisfactory solutions, it is recommended to click "best solution", and to stop after a short while to evaluate.

B.3 Running directly from C++

It is of course possible to run the code directly from C++, through the `main.cpp` file, although it was not designed to be user-friendly. This way of interacting with the solver is meant for developers, the users.

To launch the solver with the current configuration (problem parameters can be changed in the `main.cpp` file), run `make run_linux` or `make run_mac_os` depending on your OS. Make sure that you are in the `FuxCP/c++` folder. This will compile the project, generate an executable file called `Main`, and execute it.

To use the testing framework, make sure that the project has been compiled and that the `Main` executable file exists (if not, refer to the previous point). Then, to run a test of one of Fux's examples, execute

- `./Main <test number>` to execute a test in its entirety.
- `./Main <test number> <n>` to execute the test by fixing the first `n` measures to the values of the respective illustration that Fux gives. This will generate multiple solutions that have the first measures identical to Fux's example. Using this functionality shows how a composer could, in future improvements of FuxCP, give part of a composition and ask the solver to generate the rest. This widely reduces the solution space, allowing the solver to find far more optimal solutions than if it had to generate everything from scratch.

To run a test targeting a specific constraint, pass the name of the constraint (for example "1H1") as argument when executing the `Main` file.

OM 7.4

File

Windows

CP-PARAMS

General preferences

Borrowed notes

Importance

8

High cost

Value

Harmonic fifths on the downbeat

Importance

7

Low cost

Value

Harmonic octaves on the downbeat

Importance

5

Low cost

Value

Successive perfect consonances

Importance

2

Medium cost

Value

Repeating notes

Importance

9

Medium cost

Value

No harmonic triad

Importance

3

Medium cost

Value

Direct motion to perf. consonance

Importance

14

Last resort

Value

Motion cost

Importance

12

Medium cost

Value

---Direct motion

Medium cost

---Oblique motion

Low cost

---Contrary motion

No cost

Apply specific penultimate note rules

Yes

First choose the importance of each preference (1 being the most important and 14 being the least important). The solver will give priority to the most important preferences. The cost value is taken into account if two costs have the same importance.

If two costs are ranked the same, perform between them at:

Linear combination

Melodic Preferences

Melodic cost

Importance

13

No cost

Value

---Steps

Low cost

Value

---Third skips

Low cost

Value

---Fourth leaps

Forbidden

Value

---Tritone leaps

Medium cost

Value

---Fifth leaps

Medium cost

Value

---Sixth leaps

Medium cost

Value

---Seventh leaps

Low cost

Value

---Octave leaps

Low cost

Value

Solver Configuration

First voice species

1st

First voice range

Really far above

Second voice species

1st

Second voice range

Above

Third voice species

1st

Third voice range

Below

Borrowing mode

Major

Minimum % of skips

Second species specific pref.

Penultimate downbeat note is a fifth

Importance

6

Last resort

Value

Third species specific pref.

No cambiatas

Importance

11

High cost

Value

Force contrary motion after skip

No

No harmonic triad in 2nd/3rd beat

Importance

4

Medium cost

Value

Third and fourth species specific pref.

Same note in downbeat and upbeat

Importance

10

Low cost

Value

Fourth species specific pref.

No ligatures

Importance

1

Last resort

Value

Fifth species specific pref.

Many quarters (left) or many syncopations (right)

Solver Launcher

Save Config

Stop

Next Solution

Best Solution

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

Name	Description	Default value
Borrowed notes	Preference for borrowed notes outside the diatonic scale. A high cost means as few borrowed notes as possible.	High cost
Harmonic fifths on the downbeat	High cost means as few harmonic fifths on the downbeats as possible.	Low cost
Harmonic octaves on the downbeat	High cost means as few harmonic octaves on the downbeats as possible.	Low cost
Successive perfect consonances	High cost means as few successive perfect consonances as possible.	Medium cost
Repeating notes	High cost means as few repeating notes as possible, i.e. as many different notes as possible. This cost corresponds to the variety cost.	Medium cost
No harmonic triad	High cost means as many harmonic triads as possible. Note that the different costs for rule 1.H8 in four voices are calculated from this parameter.	Medium cost
Direct motion to perfect consonance	High cost means as few direct motions to perfect consonances as possible.	Last resort
Direct motion	High cost means as few direct motions as possible.	Medium cost
Oblique motion	High cost means as few oblique motions as possible.	Low cost
Contrary motion	High cost means as few contrary motions as possible.	No cost
Apply specific penultimate note rules	Force all rules on the notes of the penultimate measure. This applies only to two-part composition and refers to the penultimate note having to be a major sixth or a minor third.	Yes
Steps	High cost means as few steps as possible.	No cost
Third skips	High cost means as few third skips as possible.	Low cost
Fourth leaps	High cost means as few fourth leaps as possible.	Low cost
Tritone leaps	High cost means as few tritone leaps as possible.	Forbidden
Fifth leaps	High cost means as few fifth leaps as possible.	Medium cost
Sixth leaps	High cost means as few sixth leaps as possible.	Medium cost
Seventh leaps	High cost means as few seventh leaps as possible.	Medium cost
Octave leaps	High cost means as few octave leaps as possible.	Low cost
2nd: Penultimate downbeat note is a fifth	High cost means trying to ensure that the penultimate downbeat is not a fifth.	Last resort
3rd: No cambiatas	A high cost means as many cambiatas as possible	High cost
3rd: Force contrary motion after skip	Force that a melodic skip or leap is followed by a melodic step in a contrary motion.	No
3rd: No harmonic triad in 2nd/3rd beat	High cost means as many harmonic triads as possible on the 2nd and 3rd beat.	Medium cost
3rd& 4th: Same note in downbeat and upbeat two beats apart	High cost means as many different notes in the downbeat and upbeat.	Low cost
4th: No ligatures	High cost means as few not-ligatured notes, i.e. as many ligatures as possible.	High cost
5th: Many quarters or many syncopations	Determines the minimum percentage of quarter notes or syncopations in the fifth species. Pushing the slider all the way to one side is not recommended.	<center>
Voice species	Determines the type of counterpoint that the tool will generate.	1st and 1st
Voice range	Determines around which pitch the counterpoint will be generated depending on the pitch of the first note of the <i>cantus firmus</i> .	Above and very far above
Minimum % of skips	Determines, depending on the counterpoint size, the percentage of melodic intervals larger than one step.	0%
Borrowing mode	Type of scale from which notes can be borrowed to generate counterpoint. The first note of the <i>cantus firmus</i> determines the tonic of this scale. If none is selected, only natural notes are used. Applies everywhere except the penultimate bar.	Major
Save Config	Saves all established preferences and allows you to start a new search for this configuration later.	-
Next Solution	Starts or continues the search for the previously saved configuration. Displays a new window with the first better solution found. Displays an error message if no solution can be found.	-
Stop	Pause the search. This may take a few seconds to take effect.	-
Best Solution	Starts or continues the search for the previously saved configuration. Does not display a window, but returns the best solution found so far, accessible by evaluating the output of cp-params. Displays an error message if no other solution can be found.	-
Linear combination or maximum minimisation	Choose whether the equally important costs are combined according to a linear combination or according to a maximum minimisation. Note that this toggle does not have an effect in the current version of FuxCP, as mentioned in Chapter 9.	Linear combination

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

Appendix C

Complete FuxCP Code

It is required for us to include all of the code of FuxCP in the appendix of the manuscript. Please remember that everything is available on GitHub (<https://github.com/dieodepatoul/FuxCP4>). This appendix serves to leave an additional trace of the code.

C.1 Counterpoint header files

C.1.1 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     unordered_map<string, int> prefs;
48     vector<vector<string>> costLevels;
49 }
```

```

50     IntVar globalCost;
51     // vector<int> species;          /// the species of the counterpoint to generate
52
53 public:
54     /**
55      * Constructor of the class.
56      * @param cf a vector<int> representing the cantus firmus.
57      * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
58      * @param lb the lowest note possible for the counterpoints in MIDI
59      * @param ub the highest note possible for the counterpoints in MIDI
60      */
61     CounterpointProblem(vector<int> cf, int v_type, vector<int> m_costs, vector<int> g_costs,
62         vector<int> s_costs, vector<int> imp, int nV);
63
64     CounterpointProblem(CounterpointProblem& s);
65     virtual IntLexMinimizeSpace* copy();
66
67     virtual string to_string() const;
68
69     /**
70      * Constrain method for bab search
71      * @todo modify this function if you want to use branch and bound
72      * @param _b a space to constrain the current instance of the Problem class with upon
73      * finding a solution
74      */
75     void constrain(const IntLexMinimizeSpace& _b);
76
77     IntVarArgs cost() const;
78
79     /// Getters
80     //Part* getCounterpoint(){ return counterpoint_1; }
81     ///destructor
82     virtual ~CounterpointProblem(){ delete cantusFirmus; delete lowest; delete counterpoint_1;
83         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
98     Stratum* getLowest();
99
100     int* get_species_array_5sp(int ctp_index);
101     int* get_extended_cp_domain(int ctp_index);
102     int get_ext_cp_domain_size(int ctp_index);
103
104     void setStrata();
105 };
106
107 /**
108  * Creates a search engine for the given problem
109  * Should only be used when using OM, otherwise you can create the solver etc in the main file
110  * @todo Modify this function to add search options etc
111  * @param pb an instance of the Problem class representing a given problem
112  * @param type the type of search engine to create (see enumeration in headers/gecode_problem.
113  *        .hpp)
114  * @return a search engine for the given problem
115  */
116 Search::Base<CounterpointProblem>* make_solver(CounterpointProblem* pb, int type);
117
118 /**
119  * Returns the next solution space for the problem
120  * Should only be used when using OM

```

```

119 * @param solver a solver for the problem
120 * @return an instance of the Problem class representing the next solution to the problem
121 */
122 CounterpointProblem* get_next_solution_space(Search::Base<CounterpointProblem>* solver);
123
124 #endif //MYPROJECT_COUNTERPOINTPROBLEM_HPP

```

C.1.2 FourVoiceCounterpoint.hpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #ifndef MYPROJECT_FOURVOICECOUNTERPOINT_HPP
6 #define MYPROJECT_FOURVOICECOUNTERPOINT_HPP
7
8 #include "../Utilities.hpp"
9 #include "CounterpointProblem.hpp"
10 #include "../Parts/FirstSpeciesCounterpoint.hpp"
11 #include "../Parts/SecondSpeciesCounterpoint.hpp"
12 #include "../Parts/ThirdSpeciesCounterpoint.hpp"
13 #include "../Parts/CantusFirmus.hpp"
14
15 /**
16  * This class models a counterpoint problem with 4 voices.
17  */
18 class FourVoiceCounterpoint : public CounterpointProblem{
19 protected:
20
21     vector<Species> species;          /// the species of the counterpoints to generate
22
23 public:
24     /**
25      * Constructor of the class.
26      * @param cf a vector<int> representing the cantus firmus.
27      * @param sp the species of the counterpoint. it takes values from the enum "species" in
28      * headers/Utilities.hpp
29      * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
30      * @param lb the lowest note possible for the counterpoint in MIDI
31      * @param ub the highest note possible for the counterpoint in MIDI
32      */
33     FourVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int> v_type, vector<int>
34     m_costs, vector<int> g_costs,
35     vector<int> s_costs, vector<int> imp, int bm);
36
37     FourVoiceCounterpoint(FourVoiceCounterpoint& s);
38     IntLexMinimizeSpace* copy() override;
39
40     string to_string() const override;
41
42     /// Getters
43     // Part* getCounterpoint(){ return counterpoint; }
44
45     /// destructor : mother class destructor is sufficient, nothing to add
46
47     void uniteCounterpoints();
48
49     void uniteCosts();
50 };
51 #endif

```

C.1.3 ThreeVoiceCounterpoint.hpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4

```

```

5 #ifndef MYPROJECT_THREEVOICECOUNTERPOINT_HPP
6 #define MYPROJECT_THREEVOICECOUNTERPOINT_HPP
7
8 #include "../Utilities.hpp"
9 #include "CounterpointProblem.hpp"
10 #include "../Parts/FirstSpeciesCounterpoint.hpp"
11 #include "../Parts/SecondSpeciesCounterpoint.hpp"
12 #include "../Parts/ThirdSpeciesCounterpoint.hpp"
13 #include "../Parts/CantusFirmus.hpp"
14
15 /**
16  * This class models a counterpoint problem with 3 voices.
17  */
18 class ThreeVoiceCounterpoint : public CounterpointProblem{
19 protected:
20
21     vector<Species> species;          /// the species of the counterpoints to generate
22
23 public:
24     /**
25      * Constructor of the class.
26      * @param cf a vector<int> representing the cantus firmus.
27      * @param sp the species of the counterpoint. it takes values from the enum "species" in
28      * headers/Utilities.hpp
29      * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
30      * @param lb the lowest note possible for the counterpoint in MIDI
31      * @param ub the highest note possible for the counterpoint in MIDI
32      */
33     ThreeVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int> v_type, vector<int>
34         m_costs, vector<int> g_costs,
35         vector<int> s_costs, vector<int> imp, int bm);
36
37     ThreeVoiceCounterpoint(ThreeVoiceCounterpoint& s);
38     IntLexMinimizeSpace* copy() override;
39
40     string to_string() const override;
41
42     /// Getters
43     // Part* getCounterpoint(){ return counterpoint; }
44
45     ///destructor : mother class destructor is sufficient, nothing to add
46
47     void uniteCounterpoints();
48
49     void uniteCosts();
50 };
51 #endif //MYPROJECT_THREEVOICECOUNTERPOINT_HPP

```

C.1.4 TwoVoiceCounterpoint.hpp

```

1 //
2 // Created by Damien Sprockeels on 13/06/2024.
3 // Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
4 //
5
6 #ifndef MYPROJECT_TWVOICECOUNTERPOINT_HPP
7 #define MYPROJECT_TWVOICECOUNTERPOINT_HPP
8
9 #include "../Utilities.hpp"
10 #include "CounterpointProblem.hpp"
11 #include "../Parts/FirstSpeciesCounterpoint.hpp"
12 #include "../Parts/SecondSpeciesCounterpoint.hpp"
13 #include "../Parts/CantusFirmus.hpp"
14 #include "../constraints.hpp"
15
16 /**
17  * This class models a counterpoint problem with 2 voices.
18  */
19 class TwoVoiceCounterpoint : public CounterpointProblem{
20 protected:

```

```

21     Species species;          /// the species of the counterpoint to generate
22
23 public:
24     /**
25      * Constructor of the class.
26      * @param cf a vector<int> representing the cantus firmus.
27      * @param sp the species of the counterpoint. it takes values from the enum "species" in
28      * headers/Utilities.hpp
29      * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
30      * @param lb the lowest note possible for the counterpoint in MIDI
31      * @param ub the highest note possible for the counterpoint in MIDI
32      */
33     TwoVoiceCounterpoint(vector<int> cf, Species sp, int v_type, vector<int> m_costs, vector<
34     int> g_costs, vector<int> s_costs, vector<int> imp, int bm);
35
36     TwoVoiceCounterpoint(TwoVoiceCounterpoint& s);
37     IntLexMinimizeSpace* copy() override;
38
39     string to_string() const override;
40
41     ///destructor : mother class destructor is sufficient, nothing to add
42
43 };
44
45 #endif //MYPROJECT_TWVOICECOUNTERPOINT_HPP

```

C.2 Voice header files

C.2.1 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     protected:
13         vector<int> cf_vector;
14     public:
15         CantusFirmus(Home home, int size, vector<int> cf, Stratum* low, int v_type, vector<int>
16         > m_costs, vector<int> g_costs, vector<int> s_costs,
17         int nV);
18
19         string to_string() const override;
20
21         CantusFirmus(Home home, CantusFirmus& s); // clone constructor
22         CantusFirmus* clone(Home home) override;
23
24         IntVarArray getFirstHInterval() override;
25
26         IntVarArray getFirstMInterval() override;
27
28         IntVarArray getMotions() override;
29
30         IntVarArgs getFirstNotes() override;
31
32         int getHIntervalSize() override;
33 };
34
35
36 #endif //MYPROJECT_CANTUSFIRMUS_HPP

```

C.2.2 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
42      * constructor from the part class.
43      * Additionally, it sets all the 1st species specific variables as well as the general
44      * rules that have to be applied
45      * regardless of the species. It does not apply 1st species specific rules and does not
46      * post branching.
47      * todo maybe add a Stratum object or IntVarArray for the lowest voice
48      * @param nMes the number of measures in the composition
49      * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
50      * @param lb the lower bound for the counterpoint
51      * @param ub the upper bound for the counterpoint
52      * @param k the key of the composition
53      * @param mSpecies the species from which this is called.
54      * @param low the lowest stratum
55      * @param c the cantus firmus
56      * @param v_type the voice type of the counterpoint
57      * @param m_costs the user-defined melodic costs
58      * @param g_costs the user-defined general costs
59      * @param s_costs the user-defined specific costs
60      * @param bm parameter specifying if borrow Mode is enabled or not
61      * @param nV the number of voices - as it is a 2 voice constructor, this parameter
62      * contains the number of voices
63      */
64     FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species
65         mSpecies, Stratum* low, CantusFirmus* c, int v_type
66         , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV);
67
68     FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low
69         , CantusFirmus* c, int v_type, vector<int> m_costs
70         , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
71
72 }
```

```

66     FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low
67     , CantusFirmus* c, int v_type, vector<int> m_costs
68     , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
69
70     FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low
71     , CantusFirmus* c, int v_type, vector<int> m_costs
72     , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);
73
74     /**
75     * This function returns a string with the characteristics of the counterpoint. It calls
76     * the to_string() method from
77     * the Part class and adds 1st species specific characteristics.
78     * @return a string representation of the current instance of the FirstSpeciesCounterpoint
79     * class.
80     */
81     string to_string() const override;
82
83     /// Copy constructor. This needs to copy all useful attributes and update variables. Must
84     call the super copy constructor NO LONGER NEEDED
85     // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
86     // Copy function
87     // virtual Space *copy() override;
88
89     FifthSpeciesCounterpoint(Home home, FifthSpeciesCounterpoint& s); // clone constructor
90     FifthSpeciesCounterpoint* clone(Home home) override;
91
92     virtual IntVarArray getBranchingNotes() override;
93
94     IntVarArray getFirstHInterval() override;
95
96     IntVarArray getMotions() override;
97
98     IntVarArray getFirstMInterval() override;
99
100     int getHIntervalSize() override;
101
102     void createSpeciesArrays(Home home);
103
104 };
105 #endif

```

C.2.3 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
16 * making a constructor that takes the number of voices as a parameter
17 * todo maybe it should take a Stratum (object or just IntVarArray) for the lowest voice or
18 * something like that depending on the formalization
19 */
20 class FirstSpeciesCounterpoint : public Part{
21 protected:
22     Species motherSpecies; // The species from which this is called.
23     CantusFirmus* cantus;
24 public:
25     /**
26     * General constructor. It takes the mother species as an argument and calls the super
27     constructor from the part class.
28     */
29

```

```

25 * Additionally, it sets all the 1st species specific variables as well as the general
26 * rules that have to be applied
27 * regardless of the species. It does not apply 1st species specific rules and does not
28 * post branching.
29 * todo maybe add a Stratum object or IntVarArray for the lowest voice
30 * @param nMes the number of measures in the composition
31 * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
32 * @param lb the lower bound for the counterpoint
33 * @param ub the upper bound for the counterpoint
34 * @param k the key of the composition
35 * @param mSpecies the species from which this is called.
36 * @param low the lowest stratum
37 * @param c the cantus firmus
38 * @param v_type the voice type of the counterpoint
39 * @param m_costs the user-defined melodic costs
40 * @param g_costs the user-defined general costs
41 * @param s_costs the user-defined specific costs
42 * @param bm parameter specifying if borrow Mode is enabled or not
43 * @param nV the number of voices - as it is a 2 voice constructor, this parameter
44 * contains the number of voices
45 */
46 FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species
47 mSpecies, Stratum* low, CantusFirmus* c, int v_type
48 , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV);
49
50 /**
51 * 2 VOICES CONSTRUCTOR. This constructor is only used when creating a counterpoint of the
52 * first species. It calls the other constructor with
53 * FIRST_SPECIES as the mother species. Additionally, it posts 1st species specific
54 * constraints.
55 * @param nMes the number of measures in the composition
56 * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
57 * @param lb the lower bound for the counterpoint
58 * @param ub the upper bound for the counterpoint
59 * @param k the key of the composition
60 * @param mSpecies the species from which this is called.
61 * @param low the lowest stratum
62 * @param c the cantus firmus
63 * @param v_type the voice type of the counterpoint
64 * @param m_costs the user-defined melodic costs
65 * @param g_costs the user-defined general costs
66 * @param s_costs the user-defined specific costs
67 * @param bm parameter specifying if borrow Mode is enabled or not
68 * @param nV the number of voices - as it is a 2 voice constructor, this parameter is
69 * passed to the general constructor
70 */
71 FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low
72 , CantusFirmus* c, int v_type, vector<int> m_costs
73 , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
74
75 /**
76 * 3 VOICES CONSTRUCTOR. This constructor is only used when creating a counterpoint of the
77 * first species. It calls the other constructor with
78 * FIRST_SPECIES as the mother species. Additionally, it posts 1st species specific
79 * constraints.
80 * @param nMes the number of measures in the composition
81 * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
82 * @param lb the lower bound for the counterpoint
83 * @param ub the upper bound for the counterpoint
84 * @param k the key of the composition
85 * @param mSpecies the species from which this is called.
86 * @param low the lowest stratum
87 * @param c the cantus firmus
88 * @param v_type the voice type of the counterpoint
89 * @param m_costs the user-defined melodic costs
90 * @param g_costs the user-defined general costs
91 * @param s_costs the user-defined specific costs
92 * @param bm parameter specifying if borrow Mode is enabled or not
93 * @param nV1 the number of voices - this parameter is there to differentiate it from
94 * other number of voices
95 * @param nV2 the number of voices - as it is a 3 voice constructor, this parameter is
96 * passed to the general constructor
97 */

```

```

86 FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low
87 , CantusFirmus* c, int v_type, vector<int> m_costs
88 , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
89
90 /**
91  * 4 VOICES CONSTRUCTOR. This constructor is only used when creating a counterpoint of the
92  * first species. It calls the other constructor with
93  * FIRST_SPECIES as the mother species. Additionally, it posts 1st species specific
94  * 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 differentiate it from
109  * other number of voices
110  * @param nV2 the number of voices - this parameter is there to differentiate it from
111  * other number of voices
112  * @param nV3 the number of voices - as it is a 4 voice constructor, this parameter is
113  * passed to the general constructor
114  */
115 FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum* low
116 , CantusFirmus* c, int v_type, vector<int> m_costs
117 , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);
118
119 /**
120  * This function returns a string with the characteristics of the counterpoint. It calls
121  * the to_string() method from
122  * the Part class and adds 1st species specific characteristics.
123  * @return a string representation of the current instance of the FirstSpeciesCounterpoint
124  * class.
125  */
126 string to_string() const override;
127
128 /// Copy constructor. This needs to copy all useful attributes and update variables. Must
129 /// call the super copy constructor NO LONGER NEEDED
130 // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
131 /// Copy function
132 // virtual Space *copy() override;
133
134 FirstSpeciesCounterpoint(Home home, FirstSpeciesCounterpoint& s); // clone constructor
135 FirstSpeciesCounterpoint* clone(Home home) override;
136
137 virtual IntVarArray getBranchingNotes() override;
138
139 IntVarArray getFirstHInterval() override;
140
141 IntVarArray getMotions() override;
142
143 IntVarArray getFirstMInterval() override;
144
145 int getHIntervalSize() override;
146
147 };
148
149 #endif //FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP

```

C.2.4 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
15  * 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
17  * something like that depending on the formalization
18  */
19
20 class FourthSpeciesCounterpoint : public Part{
21 protected:
22
23     CantusFirmus* cantus;
24     IntVarArray fourthSpeciesNotesCp;
25     IntVarArray fourthSpeciesHIntervals;
26     IntVarArray fourthSpeciesMelodicIntervals;
27     IntVarArray m2IntervalsArray;
28     IntVarArray firstHInterval;
29     IntVarArray m2ZeroArray;
30     IntVarArray sol;
31
32 public:
33     /**
34     * General constructor. It takes the mother species as an argument and calls the super
35     * constructor from the part class.
36     * Additionally, it sets all the 1st species specific variables as well as the general
37     * rules that have to be applied
38     * regardless of the species. It does not apply 1st species specific rules and does not
39     * post branching.
40     * todo maybe add a Stratum object or IntVarArray for the lowest voice
41     * @param nMes the number of measures in the composition
42     * @param cf the cantus firmus todo maybe it should be a CantusFirmusObject
43     * @param lb the lower bound for the counterpoint
44     * @param ub the upper bound for the counterpoint
45     * @param k the key of the composition
46     * @param mSpecies the species from which this is called.
47     * @param low the lowest stratum
48     * @param c the cantus firmus
49     * @param v_type the voice type of the counterpoint
50     * @param m_costs the user-defined melodic costs
51     * @param g_costs the user-defined general costs
52     * @param s_costs the user-defined specific costs
53     * @param bm parameter specifying if borrow Mode is enabled or not
54     * @param nV the number of voices - as it is a 2 voice constructor, this parameter
55     * contains the number of voices
56     */
57     FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Species
58     mSpecies, Stratum* low, CantusFirmus* c, int v_type
59     , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV);
60
61     FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum*
62     low, CantusFirmus* c, int v_type, vector<int> m_costs
63     , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
64
65     FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum*
66     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     FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb, int ub, Stratum*
70     low, CantusFirmus* c, int v_type, vector<int> m_costs
71     , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);
72
73     /**
74     * This function returns a string with the characteristics of the counterpoint. It calls
75     * the to_string() method from
76     * the Part class and adds 1st species specific characteristics.
77     * @return a string representation of the current instance of the FirstSpeciesCounterpoint
78     * class.
79     */
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99

```

```

66     */
67     string to_string() const override;
68
69     /// Copy constructor. This needs to copy all useful attributes and update variables. Must
70     call the super copy constructor      NO LONGER NEEDED
71     // FirstSpeciesCounterpoint(FirstSpeciesCounterpoint &s);
72     /// Copy function
73     // virtual Space *copy() override;
74
75     FourthSpeciesCounterpoint(Home home, FourthSpeciesCounterpoint& s); // clone constructor
76     FourthSpeciesCounterpoint* clone(Home home) override;
77
78     virtual IntVarArray getBranchingNotes() override;
79
80     IntVarArray getFirstHInterval() override;
81
82     IntVarArray getMotions() override;
83
84     IntVarArray getFirstMInterval() override;
85
86     int getHIntervalSize() override;
87 };
88
89 #endif //FUXCP_BASE_FIRSTSPECIESCOUNTERPOINT_HPP

```

C.2.5 Part.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_COUNTERPOINT_HPP
7 #define FUXCP_BASE_COUNTERPOINT_HPP
8
9 #include "../Utilities.hpp"
10 #include "../Voice.hpp"
11 #include "../Stratum.hpp"
12 #include "../constraints.hpp"
13
14 #include "gecode/kernel.hh"
15 #include "gecode/int.hh"
16 #include "gecode/search.hh"
17 #include "gecode/minimodel.hh"
18 #include "gecode/set.hh"
19
20 using namespace Gecode;
21 using namespace Gecode::Search;
22 using namespace std;
23
24 /// abstract class!!! Should not be instantiated
25 /// This class represents a part, so it creates all the variables associated to that part and
26 /// posts the constraints that are species independent
27 class Part : public Voice {
28     protected:
29         Species species;
30         int nVoices;
31         int borrowMode;
32         int voice_type;
33         //Stratum* lowest;
34         IntVarArray melodicDegreeCost;
35         IntVarArray fifthCostArray;
36         IntVarArray octaveCostArray;
37         BoolVarArray is_off;
38         IntVarArray offCostArray;
39         IntVarArray costs;
40         IntVarArray varietyCostArray;
41         IntVarArray directCostArray;
42         BoolVarArray isConsonance;
43         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     //First Species specific variables
88     IntVarArray firstSpeciesNotesCp;
89     IntVarArray firstSpeciesHarmonicIntervals;
90     IntVarArray firstSpeciesMelodicIntervals;
91     IntVarArray firstSpeciesMotions;
92     IntVarArray firstSpeciesMotionCosts;
93
94     //Second species specific variables
95     BoolVarArray isDiminution;
96     IntVarArray penultCostArray;
97     IntVarArray secondSpeciesMotions;
98     IntVarArray secondSpeciesMelodicIntervals;
99     IntVarArray secondSpeciesRealMotions;
100
101     //Third species specific variables
102     IntVarArray thirdSpeciesHarmonicIntervals;
103     IntVarArray thirdSpeciesMelodicIntervals;
104     BoolVarArray is5QNArray;
105     IntVarArray cambiataCostArray;
106
107     //Fourth species specific variables
108     BoolVarArray isNoSyncopeArray;
109     IntVarArray snycopeCostArray;
110 public:
111     Part(Home home, int nMes, Species sp, vector<int> cf, int lb, int ub, int v_type,
112         vector<int> m_costs, vector<int> g_costs,
113         vector<int> s_costs, int nV, int bm);

```

```

114 // 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))
115 Part(Home home, Part& s); // clone constructor
116
117 /// must be implemented in the child classes, returns the variables to branch on
118 // virtual IntVarArray getBranchingNotes();
119
120 virtual Part* clone(Home home) override;
121
122 virtual string to_string() const;
123
124 virtual IntVarArray getBranchingNotes();
125
126 virtual IntVarArray getFirstHInterval();
127
128 virtual IntVarArray getFirstMInterval();
129
130 virtual IntVarArray getMotions();
131
132 int getSpecies() { return species; }
133
134 IntVarArray getPartNotes();
135
136 IntVarArray getCosts();
137
138 BoolVarArray getIsHighest();
139
140 int getSuccCost();
141
142 int getTriadCost();
143
144 int getVarietyCost();
145
146 IntVar getVarietyArray(int idx);
147
148 virtual int getHIntervalSize();
149
150 BoolVarArray getConsonance();
151
152 BoolVarArray getIsNotLowest();
153
154 int getSecondCost();
155 int getThirdCost();
156 int getFourthCost();
157 int getTritoneCost();
158 int getFifthCost();
159 int getSixthCost();
160 int getSeventhCost();
161 int getOctaveCost();
162 int getHFifthCost();
163 int getHOctaveCost();
164 int getDirectCost();
165 int getBorrowCost();
166 int getPenultCost();
167 int getDirectMoveCost();
168 int getCambiataCost();
169
170 IntVarArray getMelodicDegreeCost();
171
172 IntVarArray getFirstSpeciesHIntervals();
173
174 IntVarArray getFirstSpeciesNotes();
175
176 IntVarArray getFifthCostArray();
177
178 IntVarArray getOctaveCostArray();
179
180 IntVarArray getFirstSpeciesMIntervals();
181
182 IntVarArray getFirstSpeciesMotions();
183
184 IntVarArray getDirectCostArray();
185

```

```

186     BoolVarArray getIsOffArray();
187
188     vector<int> getOffDomain();
189
190     vector<int> getExtendedDomain();
191
192     IntVarArray getOffCostArray();
193
194     BoolVarArray getIsDiminution();
195
196     IntVarArray getPenultCostArray();
197
198     IntVarArray getSecondSpeciesMotions();
199
200     IntVarArray getSecondSpeciesMIntervals();
201
202     IntVarArray getSecondSpeciesRealMotions();
203
204     BoolVarArray getIs5QNArray();
205
206     IntVarArray getThirdSpeciesHIntervals();
207
208     IntVarArray getThirdSpeciesMIntervals();
209
210     IntVarArray getCambiataCostArray();
211
212     BoolVarArray getNoSyncope();
213
214     IntVarArray getSyncopeCostArray();
215
216     IntVarArray getSpeciesArray();
217
218     void add_cost(Home home, int idx, IntVarArray to_be_added, IntVarArray costs);
219
220     vector<string> getCostNames();
221 };
222
223
224 #endif //FUXCP_BASE_COUNTERPOINT_HPP

```

C.2.6 SecondSpeciesCounterpoint.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_SECONDSPECIESCOUNTERPOINT_HPP
7 #define MYPROJECT_SECONDSPECIESCOUNTERPOINT_HPP
8
9
10 #include "FirstSpeciesCounterpoint.hpp"
11 #include "../Utilities.hpp"
12
13 ///This class extends the first species class because most rules of the first species are also
14   valid for the second species
15 /**
16  * This class represents a counterpoint of the second species. It inherits from the
17   FirstSpeciesCounterpoint class.
18  */
19 class SecondSpeciesCounterpoint : public FirstSpeciesCounterpoint {
20 protected:
21     IntVarArray secondSpeciesNotesCp;          /// The notes of the counterpoint that
22     have to follow the rules for the 2nd species
23     IntVarArray secondSpeciesHarmonicIntervals; /// The harmonic intervals between the
24     notes that have to follow the 2nd species rules and the cantus firmus
25     IntVarArray secondSpeciesMotionCosts;
26     IntVarArray secondSpeciesRealMotionCosts;
27     IntVarArray secondSpeciesArsisArray;
28
29 public:

```

```

26  /**
27   *
28   * @param size
29   * @param cf
30   * @param lb
31   * @param ub
32   * @param k
33   */
34
35  SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec,
36  Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
37  , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
38
39  SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low
40  , CantusFirmus* c, int v_type, vector<int> m_costs
41  , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
42
43  SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low
44  , CantusFirmus* c, int v_type, vector<int> m_costs
45  , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
46
47  SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low
48  , CantusFirmus* c, int v_type, vector<int> m_costs
49  , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);
50
51  string to_string() const override;
52
53  // SecondSpeciesCounterpoint(SecondSpeciesCounterpoint &s); No longer a copy, now a
54  // clone (since it is not a space, and copy constructor are for spaces, while cloning
55  // functions are just used to create a deep copy of an object)
56
57  // virtual Space *copy() override;
58
59  SecondSpeciesCounterpoint(Home home, SecondSpeciesCounterpoint &s); // clone constructor
60  SecondSpeciesCounterpoint* clone(Home home) override;
61
62  IntVarArray getFirstHInterval() override;
63
64  IntVarArray getMotions() override;
65
66  IntVarArray getFirstMInterval() override;
67
68  IntVarArray getBranchingNotes() override;
69
70  int getHIntervalSize() override;
71
72 };
73
74 #endif //MYPROJECT_SECONDSPICESCOUNTERPOINT_HPP

```

C.2.7 ThirdSpeciesCounterpoint.hpp

```

1  //
2  // Created by Luc Cleenewerk and Diego de Patoul.
3  //
4
5  #ifndef MYPROJECT_THIRDSPECIESCOUNTERPOINT_HPP
6  #define MYPROJECT_THIRDSPECIESCOUNTERPOINT_HPP
7
8
9  #include "FirstSpeciesCounterpoint.hpp"
10 #include "../Utilities.hpp"
11
12 class ThirdSpeciesCounterpoint : public FirstSpeciesCounterpoint {
13 protected :
14     IntVarArray thirdSpeciesNotesCp;          /// The notes of the counterpoint that have
15     to follow the rules for the 2nd species
16     IntVarArray thirdSpeciesMotions;          /// This array is the array of REAL motions
17     IntVarArray thirdSpeciesMotionCosts;
18     IntVarArray m2IntervalsArray;

```

```

18     IntVarArray m2ZeroArray;
19     IntVarArray thirdHTriadArray;
20     IntVarArray thirdSpeciesAbsMelodic;
21
22 public:
23
24     ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, int mSpec,
25         Stratum* low, CantusFirmus* c, int v_type, vector<int> m_costs
26         , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
27
28     ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low,
29         CantusFirmus* c, int v_type, vector<int> m_costs
30         , vector<int> g_costs, vector<int> s_costs, int bm, int nV);
31
32     ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low,
33         CantusFirmus* c, int v_type, vector<int> m_costs
34         , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2);
35
36     ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb, int ub, Stratum* low,
37         CantusFirmus* c, int v_type, vector<int> m_costs
38         , vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2, int nV3);
39
40     string to_string() const override;
41
42     ThirdSpeciesCounterpoint(Home home, ThirdSpeciesCounterpoint &s); // clone constructor
43
44     ThirdSpeciesCounterpoint* clone(Home home) override;
45
46     IntVarArray getFirstHInterval() override;
47
48     IntVarArray getMotions() override;
49
50     IntVarArray getFirstMInterval() override;
51
52     IntVarArray getBranchingNotes() override;
53
54     int getHIntervalSize() override;
55 };
56 #endif

```

C.3 Other header files

C.3.1 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_arsisHarmoniesCannotBeDissonant(Home home, Part* part);
56
57  void H2_3_dissonanceImpliesDiminution(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
61           voice version is applied in the stratum constructors
62   */
63  void H3_1_endWithPerfectConsonance(Home home, Part* part);
64
65  void H3_2_penultimateNoteDomain(Home home, Part* part);
66
67  void H3_3_cambiataCost(Home home, Part* part);
68
69  //note : H5 was divided into 2 different constraints since the cantusFirmus has a smaller
70           notes array than the other parts
71
72  /**
73   * 1.H5 : Voices cannot play the same notes (except for the first and last note) (cantusFirmus
74           version)
75   */
76  void H5_1_cpAndCfDifferentNotes(Home home, Part* part, Part* cf);
77
78  /**
79   * 1.H5 : Voices cannot play the same notes (except for the first and last note) (inter-voices
80           version)
81   */
82  void H5_1_differentNotes(Home home, vector<Part*> part);
83
84  /**
85   * 1.H6 : Imperfect consonances are preferred (sets costs for fifth and octave)
86   */
87  void H6_1_preferImperfectConsonances(Home home, Part* part);
88
89  /**
90   * 1.H7 : the penultimate note must be a major sixth or a minor third depending on if it is
91           the lowest stratum or not (counterpoint version)
92   */
93  void H7_1_2v_penultimateSixthOrThird(Home home, Part* part);
94
95  /**
96   * 1.H8 : the harmonic triad should be used as much as possible (3 voices)
97   */

```

```

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
118  *         without crossing each other, then the melodic interval of
119  *         the counterpoint can be an octave leap.
120  */
121 void M1_2_octaveLeap(Home home, Part* part, Stratum* low);
122
123 void M2_2_2v_twoConsecutiveNotesAreNotTheSame(Home home, Part* part);
124
125 void M2_2_3v_melodicIntervalsNotExceedMinorSixth(Home home, vector<Part*> parts, bool
    containsThirdSpecies);
126
127 /**
128  * 1.M2 : The notes of each part should be as diverse as possible
129  */
130 void M2_1_varietyCost(Home home, vector<Part*> parts);
131
132 void P1_1_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
133
134 void P1_1_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
135
136 void P1_2_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
137
138 void P1_2_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
139
140 void P1_2_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
141
142 void P3_0_noBattuta(Home home, Part* part);
143
144 void P3_1_noBattuta(Home home, Part* part);
145
146 void P3_2_noBattuta(Home home, Part* part);
147
148 void P4_successiveCost(Home home, vector<Part*> parts, int scc_cz, IntVarArray
    successiveCostArray, vector<Species> species);
149
150 void P6_noMoveInSameDirection(Home home, vector<Part*> parts);
151
152 void P7_noSuccessiveAscendingSixths(Home home, vector<Part*> parts);
153
154 void P1_1_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part);
155
156 void H7_1_3v_penultimateSixthOrThird(Home home, Part* part);
157
158 /* =====
159  *                OTHER CONSTRAINTS
160  * =====
161  */

```

```

161
162 /**
163  * This function creates the isOff array to be able to know if a note is borrowed or not
164  */
165 void initializeIsOffArray(Home home, Part* part);
166
167 /**
168  * Two fifth species rhythms should be as diverse as possible
169  */
170 void twoFifthSpeciesDiversity_3v(Home home, Part* cp1, Part* cp2);
171
172 /**
173  * For the upper strata, there shouldn't be a minor second interval between the thesis notes
174  */
175 void noMinorSecondBetweenUpper(Home home, vector<Part*> parts);
176
177 #endif

```

C.3.2 CounterpointUtils.hpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file contains the declarations of the main functions to create problems and
4 // counterpoints.
5 //
6 #ifndef COUNTERPOINTUTILS_HPP
7 #define COUNTERPOINTUTILS_HPP
8
9 #include "Utilities.hpp"
10 #include "CounterpointProblems/TwoVoiceCounterpoint.hpp"
11 #include "CounterpointProblems/ThreeVoiceCounterpoint.hpp"
12 #include "CounterpointProblems/FourVoiceCounterpoint.hpp"
13 #include "Parts/FirstSpeciesCounterpoint.hpp"
14 #include "Parts/SecondSpeciesCounterpoint.hpp"
15 #include "Parts/ThirdSpeciesCounterpoint.hpp"
16 #include "Parts/FourthSpeciesCounterpoint.hpp"
17 #include "Parts/FifthSpeciesCounterpoint.hpp"
18
19 using namespace std;
20 using namespace Gencode;
21
22
23 /**
24  * This function creates the appropriate Part given the species of the counterpoint requested.
25  * @return a pointer to a Part object
26  */
27 Part* create_counterpoint(Home home, int species, int nMeasures, vector<int> cantusFirmus, int
    lowerBound, int upperBound, Stratum* low,
28     CantusFirmus* c, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs
    , int bm, int nV);
29
30 /**
31  * This function creates the appropriate counterpoint problem given the number of
32  * counterpoints (size of the species list) requested.
33  * @return a pointer to a counterpoint problem object
34  */
35 CounterpointProblem* create_problem(vector<int> cf, vector<Species> sp, vector<int> v_type,
    vector<int> m_costs, vector<int> g_costs,
36     vector<int> s_costs, vector<int> imp, int bm);
37
38 bool notInt(char* argv);
39
40
41 #endif

```

C.3.3 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_TEST_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 size;
28     vector<int> melodic_params;
29     vector<int> general_params;
30     vector<int> specific_params;
31     vector<int> importance;
32
33 public:
34
35     FuxTest(char* test);
36
37     FuxTest(int testNumber);
38
39     FuxTest(int testNumber, int i);
40
41     vector<Species> getSpList();
42     vector<int> getCf();
43     vector<int> getVType();
44     int getBMode();
45     vector<int> getCp();
46     int getIdx();
47
48     CounterpointProblem* test_1sp_H1();
49     CounterpointProblem* test_1sp_H2();
50     CounterpointProblem* test_1sp_H3();
51     CounterpointProblem* test_1sp_H3_2();
52     CounterpointProblem* test_1sp_H4_1();
53     CounterpointProblem* test_1sp_H4_2();
54     CounterpointProblem* test_1sp_H5();
55     CounterpointProblem* test_1sp_H7();
56     CounterpointProblem* test_1sp_H7_2();
57     CounterpointProblem* test_2sp_H2();
58     CounterpointProblem* test_3sp_H1();
59     CounterpointProblem* test_4sp_H2();
60
61     CounterpointProblem* dispatcher(char* test);
62
63     void test_2v_1sp_fig22_setter(int i);
64     void test_2v_1sp_fig23_setter(int i);
65
66     void test_2v_2sp_fig38_setter(int i);
67     void test_2v_2sp_fig39_setter(int i);
68     void test_2v_2sp_fig40_setter(int i);
69
70     void test_2v_3sp_fig55_setter(int i);
71
72     void test_2v_4sp_fig74_setter(int i);

```

```

73
74     void test_2v_5sp_fig82_setter(int i);
75
76     void test_3v_1sp_fig108_setter(int i);
77     void test_3v_1sp_fig109_setter(int i);
78     void test_3v_1sp_fig110_setter(int i);
79     void test_3v_1sp_fig111_setter(int i);
80
81     void test_3v_2sp_fig125_setter(int i);
82
83     void test_3v_3sp_fig133_setter(int i);
84
85     void test_3v_4sp_fig146_setter(int i);
86
87     void test_4v_1sp_fig166_setter(int i);
88     void test_4v_1sp_fig167_setter(int i);
89
90     void test_4v_2sp_fig176_setter(int i);
91
92     void test_4v_3sp_fig183_setter(int i);
93 };
94
95 #endif

```

C.3.4 problem_wrapper.hpp

```

1  //
2  // This file contains the declarations of the C++ interface functions.
3  //
4
5  #ifndef gecode_wrapper_hpp
6  #define gecode_wrapper_hpp
7
8  #include <stdlib.h>
9
10 #ifdef __cplusplus
11 extern "C" {
12 #endif
13
14 /**
15  * Wraps the Problem constructor.
16  * @todo modify this to include any parameters your Problem constructor requires
17  * @param size an integer representing the size of the problem
18  * @param lower_bound_domain an integer representing the lower bound of the domain of the
19  *   variables
20  * @param upper_bound_domain an integer representing the upper bound of the domain of the
21  *   variables
22  * @return A pointer to a Problem object casted as a void*
23  */
24 void* create_new_problem(int* cantusFirmus, int size, int n_cp, int* splist, int* v_types, int
25   b_mode, int min_skips, int* general_params,
26   int* motion_params, int* melodic, int* specific, int* importance, int t_off, int* scle
27   , int scale_size,
28   int* chromatic, int chrom_size, int* borrow, int borrow_size);
29
30 /**
31  * returns the size of the problem
32  * @param sp a void* pointer to a Problem object
33  * @return an integer representing the size of the problem
34  */
35 int get_size(void* sp);
36
37 void delete_pointer(void* p);
38
39 void delete_solver_pointer(void* p);
40
41 /**
42  * returns the values of the variables for a solution
43  * @param sp a void* pointer to a Problem object
44  * @return an int* pointer representing the values of the variables
45  */

```

```

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
55   *   problem
56   * @return a void* cast of a Problem* pointer
57   */
58  void* return_next_solution_space(void* solver);
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

```

C.3.5 Stratum.hpp

```

1  //
2  // Created by Luc Cleenewerk and Diego de Patoul.
3  //
4
5  #ifndef STRATUM_HPP
6  #define STRATUM_HPP
7
8  #include "Utilities.hpp"
9  #include "Voice.hpp"
10
11  #include "gecode/kernel.hh"
12  #include "gecode/int.hh"
13  #include "gecode/search.hh"
14  #include "gecode/minimodel.hh"
15  #include "gecode/set.hh"
16
17  using namespace Gecode;
18  using namespace Gecode::Search;
19  using namespace std;
20
21  /// This class represents a Stratum
22  class Stratum : public Voice{
23  protected:
24      // intvararray with species of each measure?
25
26  public:
27      Stratum(Home home, int nMes, int lb, int ub);
28
29      Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes);
30
31      Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int nV);
32
33      Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int nV1, int nV2
34      );
35
36      Stratum(Home home, Stratum& s); // clone constructor

```

```

36     Stratum* clone(Home home) override;
37
38     string to_string() const override;
39
40     void setNote(Home home, int index, IntVar note);
41
42 };
43
44
45
46 #endif //STRATUM_HPP

```

C.3.6 Utilities.hpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file contains the declarations of general utility functions.
4 //
5
6 #ifndef UTILITIES
7 #define UTILITIES
8
9 #include "gecode/kernel.hh"
10 #include "gecode/int.hh"
11 #include "gecode/search.hh"
12 #include "gecode/minimodel.hh"
13 #include "gecode/set.hh"
14
15 #include <iostream>
16 #include <vector>
17 #include <string>
18 #include <map>
19 #include <iostream>
20 #include <fstream>
21 #include <set>
22
23 using namespace std;
24 using namespace Gecode;
25
26 /*****
27  *
28  *           Useful constants
29  *
30  *****/
31
32 /** Files */
33 const string LOG_FILE = "log.txt";
34 const string STATISTICS_FILE = "statistics.txt";
35 const string STATISTICS_CSV = "statistics";
36
37 /** Types of search engines */
38 enum solver_types{
39     DFS_SOLVER, //0
40     BAB_SOLVER, //1
41     LDS_SOLVER //2
42 };
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(
67 // left_to_right_soprano_to_bass));
68
69 const vector<string> variable_selection_heuristics_names = {"Degree max", "Domain size min", "
70 Left to right",
71 "Right to left", "AFC max"};
72
73 enum value_selection{
74     VAL_MIN,          //0
75     VAL_MAX,          //1
76     VAL_MED,          //2
77     VAL_RND,          //3
78 };
79
80 // /// value selection heuristic
81 // auto branchVal = [](const Space& home, IntVar x, int i) {
82 //     return x.min();
83 // };
84
85 // /// commit function (EQ and DIFF)
86 // auto branchCommit = [](Space& home, unsigned int a, IntVar x, int i, int n){
87 //     if (a == 0U){
88 //         rel(home, x, IRT_EQ, n);
89 //     } else {
90 //         rel(home, x, IRT_NQ, n);
91 //     }
92 // };
93
94 const vector<IntValBranch> value_selection_heuristics = {INT_VAL_MIN(), INT_VAL_MAX(),
95 INT_VAL_MED(), INT_VAL_RND(1U)};
96
97 const vector<string> value_selection_heuristics_names = {"Value min", "Value max", "Median
98 value", "Value random"};
99
100 /** Voice ranges */
101 const int BASS_MIN = 40;
102 const int BASS_MAX = 60;
103 const int TENOR_MIN = 48;
104 const int TENOR_MAX = 69;
105 const int ALTO_MIN = 55;
106 const int ALTO_MAX = 75;
107 const int SOPRANO_MIN = 60;
108 const int SOPRANO_MAX = 84;
109
110 /** Melodic costs */
111 const int UNISON_COST = 0;
112 const int SECOND_COST = 1;
113 const int THIRD_COST = 3;
114 const int FOURTH_COST = 6;
115 const int FIFTH_COST = 6;
116 const int SIXTH_COST = 12;
117 const int SEVENTH_COST = 18;
118 const int OCTAVE_COST = 6;
119
120 const int MAX_MELODIC_COST = SEVENTH_COST;
121
122 /** Notes */
123 const int B_SHARP = 0;
124 const int C = 0;
125 const int C_SHARP = 1;
126 const int D_FLAT = 1;
127 const int D = 2;
128 const int D_SHARP = 3;

```

```

126 const int E_FLAT = 3;
127 const int E = 4;
128 const int F_FLAT = 4;
129 const int E_SHARP = 5;
130 const int F = 5;
131 const int F_SHARP = 6;
132 const int G_FLAT = 6;
133 const int G = 7;
134 const int G_SHARP = 8;
135 const int A_FLAT = 8;
136 const int A = 9;
137 const int A_SHARP = 10;
138 const int B_FLAT = 10;
139 const int B = 11;
140 const int C_FLAT = 11;
141
142 const vector<std::string> noteNames = {"C", "Csharp", "D", "Eb", "E", "F", "Fsharp", "G", "Ab",
    , "A", "Bb", "B"};
143
144 /** Voice positions */
145 enum voices{
146     BASS,          //0
147     TENOR,         //1
148     ALTO,          //2
149     SOPRANO        //3
150 };
151
152 const vector<std::string> voiceNames = {"Bass", "Tenor", "Alto", "Soprano"};
153
154 /** scale degrees */
155 enum degrees{
156     FIRST_DEGREE,    //0
157     SECOND_DEGREE,   //1
158     THIRD_DEGREE,    //2
159     FOURTH_DEGREE,   //3
160     FIFTH_DEGREE,    //4
161     SIXTH_DEGREE,    //5
162     SEVENTH_DEGREE   //6
163 };
164
165 const vector<std::string> degreeNames = {"First degree", "Second degree", "Third degree", "
    Fourth degree", "Fifth degree",
    "Sixth degree", "Seventh degree"};
166
167 /** Intervals */
168 // "classic" intervals
169 enum intervals{
170     UNISSON,          //0
171     MINOR_SECOND,     //1
172     MAJOR_SECOND,     //2
173     MINOR_THIRD,      //3
174     MAJOR_THIRD,       //4
175     PERFECT_FOURTH,   //5
176     TRITONE,          //6
177     PERFECT_FIFTH,    //7
178     MINOR_SIXTH,      //8
179     MAJOR_SIXTH,       //9
180     MINOR_SEVENTH,    //10
181     MAJOR_SEVENTH,    //11
182     PERFECT_OCTAVE    //12
183 };
184
185 // augmented/diminished intervals
186 const int AUGMENTED_SECOND = 3;
187 const int AUGMENTED_FOURTH = TRITONE;
188 const int DIMINISHED_FIFTH = TRITONE;
189
190 /** Chords */
191 enum chordTypes{
192     MAJOR_CHORD,          //0
193     MINOR_CHORD,          //1
194     DIMINISHED_CHORD,     //2
195     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,
213
    AUGMENTED_CHORD_INTERVALS,
    DOMINANT_SEVENTH_CHORD_INTERVALS,
214
    MAJOR_SEVENTH_CHORD_INTERVALS,
    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
220 enum chordStates{
221     FUNDAMENTAL_STATE, //0
222     FIRST_INVERSION,   //1
223     SECOND_INVERSION,  //2
224     THIRD_INVERSION    //3
225 };
226
227 const vector<std::string> stateNames = {"Fundamental state", "First inversion", "Second
    inversion", "Third inversion"};
228
229 /** Modes */
230 enum Mode {
231     IONIAN,    //0 , major mode
232     DORIAN,    //1
233     PHRYGIAN,  //2
234     LYDIAN,    //3
235     MIXOLYDIAN, //4
236     AEOLIAN,   //5 , natural minor mode
237     LOCRIAN    //6
238 };
239
240 // syntactic sugar for more commonly used modes
241 const int MAJOR_MODE = IONIAN;
242 const int MINOR_MODE = AEOLIAN;
243
244 const vector<std::string> modeNames = {"Major", "Dorian", "Phrygian", "Lydian", "Mixolydian",
    "Minor", "Locrian"};
245
246 /** Scales */
247 /// defined by the intervals between their notes
248
249 //@todo turn this into a dictionary with the name of the scale as key and the vector of
    intervals as value
250
251 const vector<int> MAJOR_SCALE = {MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND,
    MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND};
252 const vector<int> NATURAL_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND,
    MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND, MAJOR_SECOND};
253 const vector<int> HARMONIC_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND,
    MAJOR_SECOND, MINOR_SECOND, AUGMENTED_SECOND, MINOR_SECOND};
254 const vector<int> MELODIC_MINOR_SCALE = {MAJOR_SECOND, MINOR_SECOND, MAJOR_SECOND,
    MAJOR_SECOND, MAJOR_SECOND, MAJOR_SECOND, MINOR_SECOND};

```

```

255 const vector<int> BORROWED_SCALE = {PERFECT_FOURTH, MAJOR_THIRD, MAJOR_SECOND, MINOR_SECOND};
256 const vector<int> CHROMATIC_SCALE = {MINOR_SECOND, MINOR_SECOND, MINOR_SECOND, MINOR_SECOND,
    MINOR_SECOND, MINOR_SECOND, MINOR_SECOND, MINOR_SECOND, MINOR_SECOND,
257     MINOR_SECOND, MINOR_SECOND, MINOR_SECOND};
258
259 /** Part related */
260 enum Species{
261     FIRST_SPECIES, //0
262     SECOND_SPECIES, //1
263     THIRD_SPECIES, //2
264     FOURTH_SPECIES, //3
265     FIFTH_SPECIES, //4
266     CANTUS_FIRMUS //5
267 };
268
269 const map<int, int> notesPerMeasure = {{FIRST_SPECIES, 1}, {SECOND_SPECIES, 2},
270     {THIRD_SPECIES, 4}, {FOURTH_SPECIES, 2},
271     {FIFTH_SPECIES, 4}, {CANTUS_FIRMUS, 1}};
272
273 enum nVoices{
274     TWO_VOICES, //0
275     THREE_VOICES, //1
276     FOUR_VOICES //2
277 };
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 /*****
307  *
308  * Functions
309  *
310  *****/
311
312 /**
313  * For a given set of intervals between notes that loops and a starting note, returns all the
    possible notes
314  * @param note the starting note
315  * @param intervals the set of intervals between notes. It must make a loop. For example, to
    get all notes from a major
316  * scale from note, use {2, 2, 1, 2, 2, 2, 1}. To get all notes from a minor chord, use {3, 4,
    5}.
317  * @return vector<int> all the notes
318  */
319 vector<int> get_all_notes_from_interval_loop(int n, vector<int> intervals);
320

```

```

321 /**
322  * For a given tonality (root + mode), returns all the possible notes
323  * @param root the root of the tonality (in [0,11])
324  * @param scale the set of tones and semitones that define the scale
325  * @return vector<int> all the possible notes from that tonality
326  */
327 vector<int> get_all_notes_from_scale(int root, vector<int> scale);
328
329 /**
330  * For a given chord (root + mode), returns all the possible notes
331  * @param root the root of the chord
332  * @param quality the set of tones and semitones that define the chord
333  * @return vector<int> all the possible notes from that chord
334  */
335 vector<int> get_all_notes_from_chord(int root, vector<int> quality);
336
337 /**
338  * Get all values for a given note
339  * @param note a note
340  * @return vector<int> a vector containing all the given notes
341  */
342 vector<int> get_all_given_note(int note);
343
344 /**
345  * Transforms an int* into a vector<int>
346  * @param ptr an int* pointer
347  * @param size the size of the array
348  * @return a vector<int> containing the same values as the array
349  */
350 vector<int> int_pointer_to_vector(int* ptr, int size);
351
352 /**
353  * Union algorithm found and adapted from GeeksForGeeks.com
354  */
355 vector<int> vector_union(vector<int> v1, vector<int> v2);
356
357 /**
358  * Intersection algorithm found and adapted from GeeksForGeeks.com
359  */
360 vector<int> vector_intersection(vector<int> v1, vector<int> v2);
361
362 /**
363  * Difference algorithm found and adapted from GeeksForGeeks.com
364  */
365 vector<int> vector_difference(vector<int> v1, int lb, int ub);
366
367 /**
368  * Transforms a vector of integers into a string
369  * @param vector a vector of integers
370  * @return string the string representation of the vector
371  */
372 string int_vector_to_string(vector<int> vector);
373
374 /**
375  * Prints the Search::Statistics object into a readable format
376  * @param stats a Search::Statistics object representing the statistics of a search
377  * @return The string representation of the statistics object
378  */
379 string statistics_to_string(Search::Statistics stats);
380
381 /**
382  * Prints the Search::Statistics object into a csv format (coma separated)
383  * @param stats a Search::Statistics object representing the statistics of a search
384  * @return The string representation of the statistics object
385  */
386 string statistics_to_csv_string(Search::Statistics stats);
387
388 /**
389  * Returns the value of a variable as a string
390  * @param var an integer variable
391  * @param absolute a boolean indicating if the value should be returned as an absolute value (
    default is false)
392  * @return a string representing the value of the variable

```

```

393 */
394 string intVar_to_string(const IntVar &var, bool absolute = false);
395
396 /**
397  * Returns the values of an array of variables as a string. Calls the intVar_to_string
    function
398  * @param vars an array of integer variables
399  * @return a string representing the values of the variables
400  */
401 string intVarArray_to_string(IntVarArray vars);
402
403 /**
404  * Returns the value of a variable as a string
405  * @param var an integer variable
406  * @param absolute a boolean indicating if the value should be returned as an absolute value (
    default is false)
407  * @return a string representing the value of the variable
408  */
409 string boolVar_to_string(const BoolVar &var, bool absolute = false);
410
411 /**
412  * Returns the values of an array of variables as a string
413  * @param vars an array of integer variables
414  * @return a string representing the values of the variables
415  */
416 string boolVarArray_to_string(BoolVarArray vars);
417
418 /**
419  * Returns the values of an array of variables as a string
420  * @param vars an array of integer variables
421  * @return a string representing the values of the variables, ignoring unassigned variables
422  */
423 string cleanIntVarArray_to_string(IntVarArray vars);
424
425 /**
426  * Returns the values of an IntVarArgs as a string
427  * @param args an IntVarArgs
428  * @return a string representing the values
429  */
430 string intVarArgs_to_string(IntVarArgs args);
431
432 /**
433  * Returns the name of a note based on its MIDI value
434  * @param note an integer
435  */
436 string midi_to_letter(int note);
437
438 /**
439  * Returns the name of a mode based on its integer value
440  * @param mode an integer
441  * @return a string representing the name of the mode
442  */
443 string mode_int_to_name(int mode);
444
445 /**
446  * returns a string with the time
447  * @return a string with the time
448  */
449 string time();
450
451 /**
452  * Write a text into a log file
453  * @param message the text to write
454  */
455 void write_to_log_file(const char *message, const string& filename);
456
457 void writeToLogFile(const char* message);
458
459
460 #endif

```

C.3.7 Voice.hpp

```
1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #ifndef FUXCP_BASE_VOICE_HPP
6 #define FUXCP_BASE_VOICE_HPP
7
8 #include "Utilities.hpp"
9
10 #include "gecode/kernel.hh"
11 #include "gecode/int.hh"
12 #include "gecode/search.hh"
13 #include "gecode/minimodel.hh"
14 #include "gecode/set.hh"
15
16 using namespace Gecode;
17 using namespace Gecode::Search;
18 using namespace std;
19
20 /// abstract class!!! Should not be instantiated
21 /// This class represents a Voice, so it creates all the variables associated to that part and
22 /// posts the constraints that are species independent
23 class Voice{
24     protected:
25         int nMeasures;
26         int size;
27
28         int lowerBound;
29         int upperBound;
30
31         IntVarArray notes;
32         IntVarArray h_intervals; // with respect to lowest stratum. TODO set to 0 if isLowest
33         IntVarArray m_intervals_brut;
34         IntVarArray motions;
35
36     public:
37         Voice(Home home, int nMes, int lb, int ub);
38
39         Voice(Home home, Voice& s); // clone constructor
40
41         int getNMeasures() { return nMeasures; }
42         int getSize() { return size; }
43
44         int getLowerBound() { return lowerBound; }
45         int getUpperBound() { return upperBound; }
46
47         IntVarArray getNotes() { return notes; }
48         IntVarArray getMelodicIntervals();
49
50         /// must be implemented in the child classes, returns the variables to branch on
51         /// virtual IntVarArray getBranchingNotes();
52
53         virtual Voice* clone(Home home);
54
55         virtual string to_string() const;
56
57         virtual IntVarArgs getFirstNotes();
58
59         IntVarArray getHInterval();
60
61         IntVarArgs getSecondHInterval();
62 };
63
64 #endif //FUXCP_BASE_VOICE_HPP
```

C.4 Counterpoint source files

C.4.1 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,
16     vector<int> g_costs, vector<int> s_costs,
17     vector<int> imp, int nV){
18     nMeasures = cf.size();
19     lowest = new Stratum(*this, nMeasures, 0, 127);
20     cantusFirmus = new CantusFirmus(*this, nMeasures, cf, lowest, v_type, m_costs, g_costs,
21     s_costs, nV);
22     importance = imp;
23     n_unique_costs = 0;
24     importanceNames = {"borrow", "fifth", "octave", "succ", "variety", "triad", "direct", "
25     motion", "penult", "cambiata", "triad3", "m2", "syncopation", "melodic"};
26
27     //creating the map with the names of the costs and their importance
28
29     prefs = {};
30
31     setPreferenceMap(importanceNames);
32
33     orderedFactors = IntVarArray(*this, 14, 0, 1000); //orderedFactors will contain the
34     costs in order of their importance
35     for(int i = 0; i < 14; i++){
36         vector<string> tmp = {};
37         costLevels.push_back(tmp); //costLevels will contain all the
38         costs at a specific level of importance (1 to 14)
39     }
40
41     for(const auto& entry : prefs){
42         int val = entry.second-1;
43         costLevels[val].push_back(entry.first); //initialize the costLevels
44     }
45
46     globalCost = IntVar(*this, 0, 2000000); //contains the global cost
47
48     writeToLogFile("counterpointproblem constructor");
49 }
50
51 // COPY CONSTRUCTOR
52 CounterpointProblem::CounterpointProblem(CounterpointProblem& s) : IntLexMinimizeSpace(s){
53     if (s.cantusFirmus) {
54         cantusFirmus = s.cantusFirmus->clone(*this);
55     } else {
56         cantusFirmus = nullptr;
57     }
58     if (s.lowest) {
59         lowest = s.lowest->clone(*this);
60     } else {
61         lowest = nullptr;
62     }
63     if (s.counterpoint_1) {
64         counterpoint_1 = s.counterpoint_1->clone(*this);
65     } else {
66         counterpoint_1 = nullptr;
67     }
68 }
```

```

64     if (s.counterpoint_2) {
65         counterpoint_2 = s.counterpoint_2->clone(*this);
66     } else {
67         counterpoint_2 = nullptr;
68     }
69     if (s.counterpoint_3) {
70         counterpoint_3 = s.counterpoint_3->clone(*this);
71     } else {
72         counterpoint_3 = nullptr;
73     }
74     if(s.upper_1){
75         upper_1 = s.upper_1->clone(*this);
76     } else {
77         upper_1 = nullptr;
78     }
79     if(s.upper_2){
80         upper_2 = s.upper_2->clone(*this);
81     } else {
82         upper_2 = nullptr;
83     }
84     if(s.upper_3){
85         upper_3 = s.upper_3->clone(*this);
86     } else {
87         upper_3 = nullptr;
88     }
89     nMeasures = s.nMeasures;
90     importance = s.importance;
91     prefs = s.prefs;
92     sorted_voices = s.sorted_voices;
93     unitedCostNames = s.unitedCostNames;
94     costLevels = s.costLevels;
95     n_unique_costs = s.n_unique_costs;
96     importanceNames = s.importanceNames;
97     successiveCostArray.update(*this, s.successiveCostArray);
98     triadCostArray.update(*this, s.triadCostArray);
99     unitedCosts.update(*this, s.unitedCosts);
100    solutionArray.update(*this, s.solutionArray);
101    sortedCosts.update(*this, s.sortedCosts);
102    orderedFactors.update(*this, s.orderedFactors);
103    finalCosts.update(*this, s.finalCosts);
104    for(int i = 0; i < sorted_voices.size(); i++){
105        sorted_voices[i].update(*this, s.sorted_voices[i]);
106    }
107    globalCost.update(*this, s.globalCost);
108 }
109
110 IntLexMinimizeSpace* CounterpointProblem::copy(){
111     return new CounterpointProblem(*this);
112 }
113
114 void CounterpointProblem::constrain(const IntLexMinimizeSpace& _b){
115
116     const CounterpointProblem &b = dynamic_cast<const CounterpointProblem &>(_b);
117 }
118
119
120 IntVarArgs CounterpointProblem::cost() const{
121
122     return IntVarArgs(finalCosts);
123 }
124
125
126 string CounterpointProblem::to_string() const {
127     string text = "Counterpoint problem : \n";
128     text += cantusFirmus->to_string();
129     text += "\n";
130     text += "All costs : \n";
131     text += intVarArray_to_string(unitedCosts);
132     text += "\nLowest : \n";
133     text += lowest->to_string();
134     text += "\n";
135     text += "Upper : \n";
136     text += upper_1->to_string();

```

```

137     text += "\n";
138     text += "Final Costs : \n";
139     text += intVarArray_to_string(finalCosts);
140     text += "\n";
141     return text;
142 }
143
144 Home CounterpointProblem::getHome(){
145     return *this;
146 }
147
148 void CounterpointProblem::setPreferenceMap(vector<string> importance_names){
149     for(int i = 0; i < importance.size(); i++){
150         prefs.insert({importance_names[i], importance[i]});
151     }
152 }
153
154 void CounterpointProblem::orderCosts(){
155     for(int i = 0; i < 14; i++){
156         if(!costLevels[i].empty()){
157             int sm_size = 0;
158             for(int k = 0; k < costLevels[i].size(); k++){
159                 //goes through every cost at the cost level
160                 for(int t = 0; t < unitedCostNames.size(); t++){
161                     if(unitedCostNames[t]==costLevels[i][k]){
162                         //adjusts the amount of costs at this level
163                         sm_size++;
164                     }
165                 }
166             }
167             //now cst_sm_size contains the size of all the cost array on this level combined
168             IntVarArgs sm(sm_size); //sum of all the costs on this level
169             int idx = 0;
170             for(int k = 0; k < costLevels[i].size(); k++){
171                 //goes through every cost at the cost level
172                 for(int t = 0; t < unitedCostNames.size(); t++){
173                     if(unitedCostNames[t]==costLevels[i][k]){
174                         //add the cost to the intvarargs
175                         sm[idx] = unitedCosts[t];
176                         idx++;
177                     }
178                 }
179             }
180             //if there is at least one cost at this level
181             if(idx>0){
182                 //then sum all the intvarargs of this level together and add them to the
orderedFactors
183                 rel(*this, orderedFactors[n_unique_costs], IRT_EQ, expr(*this, sum(sm)));
184                 //increase the amount of unique costs
185                 n_unique_costs++;
186             }
187         }
188     }
189
190     finalCosts = IntVarArray(*this, n_unique_costs, 0, 1000000);
191     for(int i = 0; i < n_unique_costs; i++){
192         //adds the cost/costs to the finalCosts list, which will be the one that will be
minimized
193         //(this eliminates all the <not assigned> values of the intvararray for costs which
are not set)
194         rel(*this, finalCosts[i], IRT_EQ, orderedFactors[i]);
195     }
196     //globalCost is the sum of all the finalCosts
197     rel(*this, globalCost, IRT_EQ, expr(*this, sum(finalCosts)));
198 }
199
200 void CounterpointProblem::setStrata(){
201     //decided how many voices we have
202     int nVoices = 2;
203     if(upper_2!=nullptr){
204         nVoices ++;
205     }
206     if(upper_3!=nullptr){

```

```

207     nVoices++;
208 }
209
210 int size = counterpoint_1->getNMeasures();
211 //sortedVoices : will contain in order at index 0 the lowest note and then the next
    highest at index 1 etc.
212 sorted_voices = {};
213 for(int i = 0; i < size; i++){
214     //initialize the voices array which will contain the current note of a part
215     IntVarArray voices = IntVarArray(*this, nVoices, 0, 127);
216     //at index 0, we have the cantusFirmus
217     rel(*this, voices[0], IRT_EQ, cantusFirmus->getNotes()[i]);
218     //adds the correct note to the voices array at the index. special rules for fourth and
    fifth species apply
219     if(counterpoint_1->getSpecies()==FOURTH_SPECIES && i!=size-1){
220         rel(*this, voices[1], IRT_EQ, counterpoint_1->getNotes()[i*4+2]);
221     } else if(counterpoint_1->getSpecies()==FIFTH_SPECIES && i==0) {
222         rel(*this, voices[1], IRT_EQ, counterpoint_1->getNotes()[i*4+2]);
223     } else{
224         rel(*this, voices[1], IRT_EQ, counterpoint_1->getFirstNotes()[i]);
225     }
226     //same for 3 voices
227     if(nVoices>=3){
228         if(counterpoint_2->getSpecies()==FOURTH_SPECIES && i!=size-1){
229             rel(*this, voices[2], IRT_EQ, counterpoint_2->getNotes()[i*4+2]);
230         } else if(counterpoint_2->getSpecies()==FIFTH_SPECIES && i==0) {
231             rel(*this, voices[2], IRT_EQ, counterpoint_2->getNotes()[i*4+2]);
232         }else{
233             rel(*this, voices[2], IRT_EQ, counterpoint_2->getFirstNotes()[i]);
234         }
235     }
236     //same for 4 voices
237     if(nVoices>=4){
238         if(counterpoint_3->getSpecies()==FOURTH_SPECIES && i!=size-1){
239             rel(*this, voices[3], IRT_EQ, counterpoint_3->getNotes()[i*4+2]);
240         } else if(counterpoint_3->getSpecies()==FIFTH_SPECIES && i==0) {
241             rel(*this, voices[3], IRT_EQ, counterpoint_3->getNotes()[i*4+2]);
242         }else{
243             rel(*this, voices[3], IRT_EQ, counterpoint_3->getFirstNotes()[i]);
244         }
245     }
246
247     //sorting the voices. Order is necessary to get the correct index of the voices array
    to put into the sorted_voices array
248     IntVarArray order = IntVarArray(*this, nVoices, 0, nVoices-1);
249     sorted_voices.push_back(IntVarArray(*this, nVoices, 0, 127));
250     sorted(*this, voices, sorted_voices[i], order);
251
252     //set lowest and upper strata notes
253     lowest->setNote(*this, i*4, sorted_voices[i][0]);
254
255     upper_1->setNote(*this, i*4, sorted_voices[i][1]);
256     if(nVoices>=3){
257         upper_2->setNote(*this, i*4, sorted_voices[i][2]);
258     }
259     if(nVoices>=4){
260         upper_3->setNote(*this, i*4, sorted_voices[i][3]);
261     }
262
263     //if the lowest note is the same as the cantusFirmus note, then the cantusFirmus is
    the lowest stratum
264     rel(*this, lowest->getFirstNotes()[i], IRT_NQ, cantusFirmus->getNotes()[i], Reify(
    cantusFirmus->getIsNotLowest()[i]));
265
266     if(nVoices>=2){
267         //sets the isNotLowest boolean for the first counterpoint
268         if(counterpoint_1->getSpecies()==FOURTH_SPECIES && i!=size-1){
269             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[i]==1)&&(lowest->
    getFirstNotes()[i]==counterpoint_1->getNotes()[i*4+2])), IRT_NQ, 1, Reify(counterpoint_1
    ->getIsNotLowest()[i]));
270         } else if(counterpoint_1->getSpecies()==FIFTH_SPECIES && i==0){
271             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[i]==1)&&(lowest->
    getFirstNotes()[i]==counterpoint_1->getNotes()[i*4+2])), IRT_NQ, 1, Reify(counterpoint_1

```

```

272     ->getIsNotLowest()[i]));
273     } else{
274         rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[i]==1)&&(lowest->
275         getFirstNotes()[i]==counterpoint_1->getFirstNotes()[i])), IRT_NQ, 1, Reify(counterpoint_1
276         ->getIsNotLowest()[i]));
277     }
278
279     }
280     if(nVoices==3){
281         //if we have 3 voices and both the counterpoint and the cantusFirmus are the same
282         //both must be 1), then the counterpoint2 is the lowest
283         rel(*this, expr(*this, counterpoint_1->getIsNotLowest()[i]!=cantusFirmus->
284         getIsNotLowest()[i]), IRT_EQ, counterpoint_2->getIsNotLowest()[i]));
285     }
286     if(nVoices==4){
287         //set the counterpoint2 lowest boolean
288         if(counterpoint_2->getSpecies()==FOURTH_SPECIES && i!=size-1){
289             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[i]==1)&&(counterpoint_1
290             ->getIsNotLowest()[i]==1)&&(lowest->getFirstNotes()[i]==counterpoint_2->getNotes()[i*4
291             +2])), IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[i]));
292         } else if(counterpoint_2->getSpecies()==FIFTH_SPECIES && i==0){
293             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[i]==1)&&(counterpoint_1
294             ->getIsNotLowest()[i]==1)&&(lowest->getFirstNotes()[i]==counterpoint_2->getNotes()[i*4
295             +2])), IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[i]));
296         } else {
297             rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[i]==1)&&(counterpoint_1
298             ->getIsNotLowest()[i]==1)&&(lowest->getFirstNotes()[i]==counterpoint_2->getFirstNotes()[i
299             ])), IRT_NQ, 1, Reify(counterpoint_2->getIsNotLowest()[i]));
300         }
301         //if we have 4 voices and the counterpoint1, counterpoint2 and the cantusFirmus
302         //are the same (all must be 1), then the counterpoint3 is the lowest
303         rel(*this, expr(*this, (cantusFirmus->getIsNotLowest()[i]==1) && (counterpoint_1->
304         getIsNotLowest()[i]==1) && (counterpoint_2->getIsNotLowest()[i]==1)), IRT_NQ,
305         counterpoint_3->getIsNotLowest()[i]);
306
307         //the following are the same but for the isHighest array to define the highest
308         //stratum (needed for a 4 voice specific constraint)
309         rel(*this, upper_3->getFirstNotes()[i], IRT_EQ, cantusFirmus->getNotes()[i], Reify(
310         cantusFirmus->getIsHighest()[i]));
311
312         if(counterpoint_1->getSpecies()==FOURTH_SPECIES && i!=size-1){
313             rel(*this, expr(*this, (cantusFirmus->getIsHighest()[i]==0)&&(upper_3->
314             getFirstNotes()[i]==counterpoint_1->getNotes()[i*4+2])), IRT_NQ, 0, Reify(counterpoint_1->
315             getIsHighest()[i]));
316         } else if(counterpoint_1->getSpecies()==FIFTH_SPECIES && i==0){
317             rel(*this, expr(*this, (cantusFirmus->getIsHighest()[i]==0)&&(upper_3->
318             getFirstNotes()[i]==counterpoint_1->getNotes()[i*4+2])), IRT_NQ, 0, Reify(counterpoint_1->
319             getIsHighest()[i]));
320         } else {
321             rel(*this, expr(*this, (cantusFirmus->getIsHighest()[i]==0)&&(upper_3->
322             getFirstNotes()[i]==counterpoint_1->getFirstNotes()[i])), IRT_NQ, 0, Reify(counterpoint_1
323             ->getIsHighest()[i]));
324         }
325
326         if(counterpoint_2->getSpecies()==FOURTH_SPECIES && i!=size-1){
327             rel(*this, expr(*this, (cantusFirmus->getIsHighest()[i]==0)&&(counterpoint_1->
328             getIsHighest()[i]==0)&&(upper_3->getFirstNotes()[i]==counterpoint_2->getNotes()[i*4+2])),
329             IRT_NQ, 0, Reify(counterpoint_2->getIsHighest()[i]));
330         } else if(counterpoint_2->getSpecies()==FIFTH_SPECIES && i==0){
331             rel(*this, expr(*this, (cantusFirmus->getIsHighest()[i]==0)&&(counterpoint_1->
332             getIsHighest()[i]==0)&&(upper_3->getFirstNotes()[i]==counterpoint_2->getNotes()[i*4+2])),
333             IRT_NQ, 0, Reify(counterpoint_2->getIsHighest()[i]));
334         } else {
335             rel(*this, expr(*this, (cantusFirmus->getIsHighest()[i]==0)&&(counterpoint_1->
336             getIsHighest()[i]==0)&&(upper_3->getFirstNotes()[i]==counterpoint_2->getFirstNotes()[i])),
337             IRT_NQ, 0, Reify(counterpoint_2->getIsHighest()[i]));
338         }
339
340         rel(*this, expr(*this, (cantusFirmus->getIsHighest()[i]==0) && (counterpoint_1->
341         getIsHighest()[i]==0) && (counterpoint_2->getIsHighest()[i]==0)), IRT_EQ, counterpoint_3->

```

```

316     getIsHighest()[i]);
317     }
318     if(i > 0){
319
320         //the following lines define the melodic interval of the lowest stratum
321
322         vector<IntVarArray> corresponding_m_intervals;
323
324         corresponding_m_intervals.push_back(cantusFirmus->getMelodicIntervals());
325         for(int j = 0; j < nVoices-1; j++){
326             Part* curr_cp;
327
328             if(j==0) curr_cp=counterpoint_1;
329             else if(j==1) curr_cp=counterpoint_2;
330             else curr_cp = counterpoint_3;
331
332             if(curr_cp->getSpecies()==FIRST_SPECIES){
333                 corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
getMelodicIntervals().slice(0, 4, curr_cp->getMelodicIntervals().size())));
334             } else if(curr_cp->getSpecies()==SECOND_SPECIES){
335                 corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
getMelodicIntervals().slice(2, 4, curr_cp->getMelodicIntervals().size())));
336             } else if(curr_cp->getSpecies()==THIRD_SPECIES){
337                 corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
getMelodicIntervals().slice(3, 4, curr_cp->getMelodicIntervals().size())));
338             } else if(curr_cp->getSpecies()==FOURTH_SPECIES){
339                 corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
getMelodicIntervals().slice(2, 4, curr_cp->getMelodicIntervals().size())));
340             } else if(curr_cp->getSpecies()==FIFTH_SPECIES){
341                 corresponding_m_intervals.push_back(IntVarArray(*this, curr_cp->
getMelodicIntervals().slice(2, 4, curr_cp->getMelodicIntervals().size())));
342             }
343         }
344
345         rel(*this, (cantusFirmus->getIsNotLowest()[i]==0) >> (lowest->getMelodicIntervals
()[i-1]==corresponding_m_intervals[0][i-1]));
346         rel(*this, (counterpoint_1->getIsNotLowest()[i]==0) >> (lowest->
getMelodicIntervals()[i-1]==corresponding_m_intervals[1][i-1]));
347         if(nVoices>=3){
348             rel(*this, (counterpoint_2->getIsNotLowest()[i]==0) >> (lowest->
getMelodicIntervals()[i-1]==corresponding_m_intervals[2][i-1]));
349         }
350         if(nVoices>=4){
351             rel(*this, (counterpoint_3->getIsNotLowest()[i]==0) >> (lowest->
getMelodicIntervals()[i-1]==corresponding_m_intervals[3][i-1]));
352         }
353     }
354 }
355 }
356 }
357
358 IntVarArray CounterpointProblem::getSolutionArray(){
359     return solutionArray;
360 }
361
362 /**
363  * Returns the size of the problem
364  * @return an integer representing the size of the vars array          RETURNS THE SIZE OF
THE SOLUTION_ARRAY
365  */
366 int CounterpointProblem::getSize(){
367     string message = "getSize function called. size = " + std::to_string(solutionArray.size())
+ "\n";
368     writeToLogFile(message.c_str());
369     return this->solutionArray.size(); /// have to use this-> ?
370 }
371
372 /**
373  * Returns the values taken by the variables vars in a solution
374  * @todo Modify this to return the solution for your problem. This function uses @param size
to generate an array of integers
375  * @return an array of integers representing the values of the variables in a solution

```

```

376 */
377 int* CounterpointProblem::return_solution(){
378     string message = "return_solution method. Solution : [";
379     int* solution = new int[solutionArray.size()];
380     for(int i = 0; i < solutionArray.size(); i++){
381         solution[i] = solutionArray[i].val();
382         message += std::to_string(solution[i]) + " ";
383     }
384     message += "]\n";
385     writeToLogFile(message.c_str());
386     return solution;
387 }
388
389
390 int* CounterpointProblem::get_species_array_5sp(int ctp_index){
391
392     FifthSpeciesCounterpoint* fifth_sp_ctp;
393     if (ctp_index == 0) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint*>(counterpoint_1)
394     ;
395     else if (ctp_index == 1) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint*>(
396         counterpoint_2);
397     else if (ctp_index == 2) fifth_sp_ctp = dynamic_cast<FifthSpeciesCounterpoint*>(
398         counterpoint_3);
399     else{
400         writeToLogFile("invalid value of ctp_index given as argument to get_species_array_5sp"
401             );
402         return nullptr;
403     }
404
405     if (!fifth_sp_ctp){
406         cout << "type cast of part to fifth species failed " << endl;
407         writeToLogFile("type cast of part to fifth species failed");
408         return nullptr;
409     }
410
411     IntVarArray speciesArray = fifth_sp_ctp->getSpeciesArray();
412
413     string message = "return_species_array method. Species array : [";
414     int* solution = new int[speciesArray.size()];
415     for(int i = 0; i < speciesArray.size(); i++){
416
417         switch (speciesArray[i].val())
418         {
419             case FIRST_SPECIES:
420                 solution[i] = 1;
421                 break;
422
423             case SECOND_SPECIES:
424                 solution[i] = 2;
425                 break;
426
427             case THIRD_SPECIES:
428                 solution[i] = 3;
429                 break;
430
431             case FOURTH_SPECIES:
432                 solution[i] = 4;
433                 break;
434
435             case -1:
436                 solution[i] = 0;
437                 break;
438
439             default:
440                 cout << "invalid value in species array" << endl;
441                 writeToLogFile("invalid value in species array");
442                 return nullptr;
443         }
444         message += std::to_string(solution[i]) + " ";
445     }
446     message += "]\n";
447     writeToLogFile(message.c_str());
448     return solution;

```

```

445 }
446
447
448 int* CounterpointProblem::get_extended_cp_domain(int ctp_index){
449     vector<int> ext_cp_dom;
450     if(ctp_index == 0) ext_cp_dom = counterpoint_1->getExtendedDomain();
451     else if(ctp_index == 1) ext_cp_dom = counterpoint_2->getExtendedDomain();
452     else if(ctp_index == 2) ext_cp_dom = counterpoint_3->getExtendedDomain();
453     else{
454         writeToLogFile("invalid value of ctp_index given as argument to get_extended_cp_domain
455             ");
456         return nullptr;
457     }
458     writeToLogFile(int_vector_to_string(ext_cp_dom).c_str());
459
460     int* ext_cp_dom_int_ptr = new int[ext_cp_dom.size()];
461     for(int i = 0; i < ext_cp_dom.size(); i++){
462         ext_cp_dom_int_ptr[i] = ext_cp_dom[i];
463         // message += std::to_string(solution[i]) + " ";
464     }
465
466     return ext_cp_dom_int_ptr;
467 }
468
469 int CounterpointProblem::get_ext_cp_domain_size(int ctp_index){
470     int cpDomSize;
471     if(ctp_index == 0) cpDomSize = counterpoint_1->getExtendedDomain().size();
472     else if(ctp_index == 1) cpDomSize = counterpoint_2->getExtendedDomain().size();
473     else if(ctp_index == 2) cpDomSize = counterpoint_3->getExtendedDomain().size();
474     else{
475         writeToLogFile("invalid value of ctp_index given as argument to get_extended_cp_domain
476             ");
477         return -1;
478     }
479
480     string message = "getextcpdomainsize function called. size = " + std::to_string(cpDomSize)
481         + "\n";
482     writeToLogFile(message.c_str());
483     return cpDomSize;
484 }
485
486 Stratum* CounterpointProblem::getLowest(){
487     return lowest;
488 }
489
490 /*****
491  * Search engine methods *
492  *****/
493 //Gecode::Search::TimeStop global_timeout(50000);
494
495 /**
496  * Creates a search engine for the given problem
497  * Should only be used when using OM, otherwise you can create the solver etc in the main file
498  * @todo Modify this function to add search options etc
499  * @param pb an instance of the Problem class representing a given problem
500  * @param type the type of search engine to create (see enumeration in headers/gecode_problem.
501  *     hpp)
502  * @return a search engine for the given problem
503  */
504 Search::Base<CounterpointProblem>* make_solver(CounterpointProblem* pb, int type){
505     string message = "make_solver function called. type of solver :\n" + to_string(type) + "\n
506 ";
507     writeToLogFile(message.c_str());
508
509     Gecode::Search::Options opts;
510     /**@todo add here any options you want*/
511     //opts.stop = &global_timeout;
512     opts.threads = 1;
513
514     if (type == bab_solver)
515         return new BAB<CounterpointProblem>(pb, opts);
516     else // default case

```

```

513         return new DFS<CounterpointProblem>(pb, opts);
514     }
515
516     /**
517     * Returns the next solution space for the problem
518     * Should only be used when using OM
519     * @param solver a solver for the problem
520     * @return an instance of the Problem class representing the next solution to the problem
521     */
522     CounterpointProblem* get_next_solution_space(Search::Base<CounterpointProblem>* solver){
523         string message = "get_next_solution_space function called.\n";
524         // RESET TIMEOUT OBJECT HERE
525         //global_timeout.reset();
526         CounterpointProblem* sol_space = solver->next();
527         if (sol_space == nullptr){
528             message += "solution_space was null. \n";
529             writeToLogFile(message.c_str());
530             return NULL;
531         }
532         message += sol_space->to_string();
533         writeToLogFile(message.c_str());
534         return sol_space;
535     }

```

C.4.2 FourVoiceCounterpoint.cpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #include "../headers/CounterpointProblems/FourVoiceCounterpoint.hpp"
6 #include "../headers/CounterpointUtils.hpp"
7
8 FourVoiceCounterpoint::FourVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int>
9     v_type, vector<int> m_costs, vector<int> g_costs,
10     vector<int> s_costs, vector<int> imp, int bm):
11     CounterpointProblem(cf, -1, m_costs, g_costs, s_costs, imp, FOUR_VOICES)
12 {
13     species = sp;
14
15     //initialize upper strata
16
17     upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES,
18         FOUR_VOICES);
19     upper_2 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES,
20         FOUR_VOICES);
21     upper_3 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES,
22         FOUR_VOICES);
23
24     //create counterpoints
25
26     counterpoint_1 = create_counterpoint(*this, species[0], nMeasures, cf, (6 * v_type[0] - 6)
27         + cf[0], (6 * v_type[0] + 12) + cf[0], lowest,
28         cantusFirmus, v_type[0], m_costs, g_costs, s_costs, bm, FOUR_VOICES);
29     counterpoint_2 = create_counterpoint(*this, species[1], nMeasures, cf, (6 * v_type[1] - 6)
30         + cf[0], (6 * v_type[1] + 12) + cf[0], lowest,
31         cantusFirmus, v_type[1], m_costs, g_costs, s_costs, bm, FOUR_VOICES);
32     counterpoint_3 = create_counterpoint(*this, species[2], nMeasures, cf, (6 * v_type[2] - 6)
33         + cf[0], (6 * v_type[2] + 12) + cf[0], lowest,
34         cantusFirmus, v_type[2], m_costs, g_costs, s_costs, bm, FOUR_VOICES);
35
36     //create strata
37
38     setStrata();
39
40     //creating variables
41
42     vector<Part*> parts = {cantusFirmus, counterpoint_1, counterpoint_2, counterpoint_3};
43     int scc_cz = 3*((cantusFirmus->getSize()/4)-1);
44     bool containsThirdSpecies = 0;
45
46 }

```

```

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   //G9 last chord must have the same fundamental as the cf (used throughout the composition)
44   G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
45
46   for(int p = 1; p < parts.size(); p++){
47       // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
48       G6_noChromaticMelodies(*this, parts[p], sp[p-1]);
49   }
50
51   //H8 : harmonic triads are preferred, adapted for 4 voices
52   H8_4v_preferHarmonicTriad(*this, triadCostArray, upper_1, upper_2, upper_3);
53
54   //M4 variety cost (notes should be as diverse as possible)
55   M2_1_varietyCost(*this, parts);
56
57   //two fifth species counterpoints should be as different as possible
58   twoFifthSpeciesDiversity_3v(*this, counterpoint_1, counterpoint_3);
59
60   //P4 avoid successive perfect consonances
61   P4_successiveCost(*this, parts, scc_cz, successiveCostArray, species);
62
63   //P6 : no move in same direction
64   if(counterpoint_1->getSpecies() != FORTH_SPECIES && counterpoint_2->getSpecies() !=
    FORTH_SPECIES && counterpoint_3->getSpecies() != FORTH_SPECIES){
65       P6_noMoveInSameDirection(*this, parts);
66   }
67
68   //P7 : no successive ascending sixths
69   P7_noSuccessiveAscendingSixths(*this, parts);
70
71   //2.M2, have to write it here since it has a weird interaction with the third species
72   M2_2_3v_melodicIntervalsNotExceedMinorSixth(*this, parts, containsThirdSpecies);
73
74   //no minor second interval between upper
75   noMinorSecondBetweenUpper(*this, parts);
76
77   solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size() +
    counterpoint_2->getBranchingNotes().size() +
78     counterpoint_3->getBranchingNotes().size(), 0, 127);
79
80   unitedCosts = IntVarArray(*this, 14, 0, 10000000);
81   unitedCostNames = {};
82
83   uniteCounterpoints();
84   uniteCosts();
85
86   orderCosts();
87
88   //Branching strategies
89
90   branch(*this, lowest->getNotes().slice(0, 4/notePerMeasure.at(FIRST_SPECIES), lowest->
    getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN());
91
92   if(species[0] == FIFTH_SPECIES){
93       branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U)
    );
94   }
95   if(species[1] == FIFTH_SPECIES){
96       branch(*this, counterpoint_2->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U)
    );
97   }
98   if(species[2] == FIFTH_SPECIES){
99       branch(*this, counterpoint_3->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U)
    );
100   }
101
102   if(species[0] == FIFTH_SPECIES){

```

```

103     branch(*this, counterpoint_1->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
104           INT_VAL_SPLIT_MIN());
105 }
106 if(species[1]==FIFTH_SPECIES){
107     branch(*this, counterpoint_2->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
108           INT_VAL_SPLIT_MIN());
109 }
110 if(species[2]==FIFTH_SPECIES){
111     branch(*this, counterpoint_3->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
112           INT_VAL_SPLIT_MIN());
113 }
114 if(species[0]==FIFTH_SPECIES){
115     branch(*this, counterpoint_1->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
116           INT_VAL_SPLIT_MIN());
117 }
118 if(species[1]==FIFTH_SPECIES){
119     branch(*this, counterpoint_2->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
120           INT_VAL_SPLIT_MIN());
121 }
122 if(species[2]==FIFTH_SPECIES){
123     branch(*this, counterpoint_3->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
124           INT_VAL_SPLIT_MIN());
125 }
126 if(species[0]==FOURTH_SPECIES){
127     branch(*this, counterpoint_1->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
128           INT_VAL_MIN());
129 }
130 if(species[1]==FOURTH_SPECIES){
131     branch(*this, counterpoint_2->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
132           INT_VAL_MIN());
133 }
134 if(species[2]==FOURTH_SPECIES){
135     branch(*this, counterpoint_3->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
136           INT_VAL_MIN());
137 }
138 branch(*this, solutionArray, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
139 cout << "HERE" << endl;
140 }
141
142 // COPY CONSTRUCTOR
143 FourVoiceCounterpoint::FourVoiceCounterpoint(FourVoiceCounterpoint& s) : CounterpointProblem(s)
144 {
145     species = s.species;
146 }
147
148 IntLexMinimizeSpace* FourVoiceCounterpoint::copy(){
149     return new FourVoiceCounterpoint(*this);
150 }
151
152 string FourVoiceCounterpoint::to_string() const {
153     string text = CounterpointProblem::to_string();
154     text += "Counterpoint 1 : \n";
155     text += counterpoint_1->to_string();
156     text += "\n";
157     text += "Counterpoint 2 : \n";
158     text += counterpoint_2->to_string();
159     text += "\n";
160     text += "Counterpoint 3 : \n";
161     text += counterpoint_3->to_string();
162     text += "\n";
163     text += "Solution array : \n";
164     text += intVarArray_to_string(solutionArray);
165     text += "\n";
166     return text;
167 }
168
169 void FourVoiceCounterpoint::uniteCounterpoints(){
170     int idx = 0;
171     for(int i = 0; i < counterpoint_1->getBranchingNotes().size(); i++){

```

```

166     rel(*this, solutionArray[idx], IRT_EQ, counterpoint_1->getBranchingNotes()[i]);
167     idx++;
168 }
169 for(int i = 0; i < counterpoint_2->getBranchingNotes().size(); i++){
170     rel(*this, solutionArray[idx], IRT_EQ, counterpoint_2->getBranchingNotes()[i]);
171     idx++;
172 }
173 for(int i = 0; i < counterpoint_3->getBranchingNotes().size(); i++){
174     rel(*this, solutionArray[idx], IRT_EQ, counterpoint_3->getBranchingNotes()[i]);
175     idx++;
176 }
177 }
178
179 void FourVoiceCounterpoint::uniteCosts(){
180     int cp1_idx = 0;
181     int cp2_idx = 0;
182     int cp3_idx = 0;
183     for(int i = 0; i < 14; i++){
184         string name = importanceNames[i];
185         if(name=="succ"){
186             unitedCostNames.push_back(name);
187             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(successiveCostArray)
188             )));
189         } else if(name=="triad"){
190             unitedCostNames.push_back(name);
191             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(triadCostArray))));
192         } else {
193             //decides if the cost is present for any counterpoint
194
195             bool cp1_contains = 0;
196             bool cp2_contains = 0;
197             bool cp3_contains = 0;
198             int sz = 0;
199             //determining in how many counterpoints a cost appears
200             for(int t = 0; t < counterpoint_1->getCostNames().size(); t++){
201                 if(name==counterpoint_1->getCostNames()[t]){
202                     cp1_contains=1;
203                     sz++;
204                 }
205             }
206             for(int t = 0; t < counterpoint_2->getCostNames().size(); t++){
207                 if(name==counterpoint_2->getCostNames()[t]){
208                     cp2_contains=1;
209                     sz++;
210                 }
211             }
212             for(int t = 0; t < counterpoint_3->getCostNames().size(); t++){
213                 if(name==counterpoint_3->getCostNames()[t]){
214                     cp3_contains=1;
215                     sz++;
216                 }
217             }
218             //if it is not present in any counterpoint -> leave it empty
219             if(!cp1_contains && !cp2_contains && !cp3_contains){
220                 unitedCostNames.push_back("NOT ADDED");
221             } else { //else -> add the costs together for that entry
222                 unitedCostNames.push_back(name);
223                 //adds the cost to the IntVarArgs
224                 IntVarArgs x(sz);
225                 int idx=0;
226                 if(cp1_contains){
227                     x[idx] = counterpoint_1->getCosts()[cp1_idx];
228                     idx++;
229                     cp1_idx++;
230                 }
231                 if(cp2_contains){
232                     x[idx] = counterpoint_2->getCosts()[cp2_idx];
233                     idx++;
234                     cp2_idx++;
235                 }
236                 if(cp3_contains){
237                     x[idx] = counterpoint_3->getCosts()[cp3_idx];

```

```

238         idx++;
239         cp3_idx++;
240     }
241     rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(x)));
242 }
243 }
244 }
245 }

```

C.4.3 ThreeVoiceCounterpoint.cpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #include "../headers/CounterpointProblems/ThreeVoiceCounterpoint.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 sp the species of the counterpoint. it takes values from the enum "species" in
12  * headers/Utilities.hpp
13  * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
14  * @param lb the lowest note possible for the counterpoint in MIDI
15  * @param ub the highest note possible for the counterpoint in MIDI
16  */
17 ThreeVoiceCounterpoint::ThreeVoiceCounterpoint(vector<int> cf, vector<Species> sp, vector<int>
18     v_type, vector<int> m_costs, vector<int> g_costs,
19     vector<int> s_costs, vector<int> imp, int bm) :
20     CounterpointProblem(cf, -1, m_costs, g_costs, s_costs, imp, THREE_VOICES){
21     species = sp;
22
23     //initialize upper strata
24
25     upper_1 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES);
26     upper_2 = new Stratum(*this, nMeasures, 0, 127, lowest->getNotes(), THREE_VOICES);
27
28     //create counterpoints
29
30     counterpoint_1 = create_counterpoint(*this, species[0], nMeasures, cf, (6 * v_type[0] - 6)
31         + cf[0], (6 * v_type[0] + 12) + cf[0], lowest,
32         cantusFirmus, v_type[0], m_costs, g_costs, s_costs, bm, THREE_VOICES);
33     counterpoint_2 = create_counterpoint(*this, species[1], nMeasures, cf, (6 * v_type[1] - 6)
34         + cf[0], (6 * v_type[1] + 12) + cf[0], lowest,
35         cantusFirmus, v_type[1], m_costs, g_costs, s_costs, bm, THREE_VOICES);
36
37     //create strata
38
39     setStrata();
40
41     //creating variables
42
43     vector<Part*> parts = {cantusFirmus, counterpoint_1, counterpoint_2};
44     int scc_cz = ((cantusFirmus->getSize())/4)-1;
45     bool containsThirdSpecies = 0;
46
47     triadCostArray = IntVarArray(*this, counterpoint_1->getFirstHInterval().size(), IntSet({0,
48         counterpoint_1->getTriadCost()}));
49     successiveCostArray = IntVarArray(*this, scc_cz, IntSet({0, counterpoint_1->getSuccCost()
50         }));
51
52     //G9 last chord must have the same fundamental as the cf (used throughout the composition)
53     G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
54
55     for(int p = 1; p < parts.size(); p++){
56         // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
57         G6_noChromaticMelodies(*this, parts[p], sp[p-1]);
58     }
59
60     //H5 for three voices

```

```

55     H5_1_differentNotes(*this, parts);
56
57     //H8 : the triad should be used as much as possible
58     H8_3v_preferHarmonicTriad(*this, counterpoint_1, triadCostArray, upper_1, upper_2);
59
60     //M4 variety cost (notes should be as diverse as possible)
61     M2_1_varietyCost(*this, parts);
62
63     //two fifth species counterpoints should be as different as possible
64     twoFifthSpeciesDiversity_3v(*this, counterpoint_1, counterpoint_2);
65
66     //P4 avoid successive perfect consonances
67     P4_successiveCost(*this, parts, scc_cz, successiveCostArray, species);
68
69     //P6 : no move in same direction
70     if(counterpoint_1->getSpecies()!=FOURTH_SPECIES&&counterpoint_2->getSpecies()!=
        FOURTH_SPECIES){ //doesn't apply to 4th species
71         P6_noMoveInSameDirection(*this, parts);
72     }
73
74     //P7 : no successive ascending sixths
75     P7_noSuccessiveAscendingSixths(*this, parts);
76
77     //2.M2, have to write it here since it has a weird interaction with the third species
78     M2_2_3v_melodicIntervalsNotExceedMinorSixth(*this, parts, containsThirdSpecies);
79
80     solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size() +
        counterpoint_2->getBranchingNotes().size(), 0, 127);
81
82     unitedCosts = IntVarArray(*this, 14, 0, 10000000);
83     unitedCostNames = {};
84
85     uniteCounterpoints();
86     uniteCosts();
87
88     for(int i = 0; i < unitedCostNames.size(); i++){
89         cout << unitedCostNames[i] << endl;
90     }
91
92     orderCosts();
93
94     //Branching strategies
95
96     branch(*this, lowest->getNotes().slice(0, 4/notesPerMeasure.at(FIRST_SPECIES), lowest->
        getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN());
97
98     if(species[0]==FIFTH_SPECIES){
99         branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U)
        );
100     }
101     if(species[1]==FIFTH_SPECIES){
102         branch(*this, counterpoint_2->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U)
        );
103     }
104
105     if(species[0]==FIFTH_SPECIES){
106         branch(*this, counterpoint_1->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
        INT_VAL_SPLIT_MIN());
107     }
108     if(species[1]==FIFTH_SPECIES){
109         branch(*this, counterpoint_2->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
        INT_VAL_SPLIT_MIN());
110     }
111
112     if(species[0]==FOURTH_SPECIES || species[0]==FIFTH_SPECIES){
113         branch(*this, counterpoint_1->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
        INT_VAL_MIN());
114     }
115     if(species[1]==FOURTH_SPECIES || species[1]==FIFTH_SPECIES){
116         branch(*this, counterpoint_2->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
        INT_VAL_MIN());
117     }
118

```

```

119     branch(*this, solutionArray, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
120
121     writeToLogFile(("solution array size : " + std::to_string(solutionArray.size())).c_str());
122 }
123
124 // COPY CONSTRUCTOR
125 ThreeVoiceCounterpoint::ThreeVoiceCounterpoint(ThreeVoiceCounterpoint& s) :
126     CounterpointProblem(s){
127     species = s.species;
128 }
129
130 IntLexMinimizeSpace* ThreeVoiceCounterpoint::copy(){
131     return new ThreeVoiceCounterpoint(*this);
132 }
133
134
135 string ThreeVoiceCounterpoint::to_string() const {
136     string text = CounterpointProblem::to_string();
137     text += "Counterpoint 1 : \n";
138     text += counterpoint_1->to_string();
139     text += "\n";
140     text += "Counterpoint 2 : \n";
141     text += counterpoint_2->to_string();
142     text += "\n";
143     text += "Successive cost array : \n";
144     text += intVarArray_to_string(successiveCostArray);
145     text += "\n";
146     text += "Solution array : \n";
147     text += intVarArray_to_string(solutionArray);
148     text += "\n";
149     return text;
150 }
151
152 void ThreeVoiceCounterpoint::uniteCounterpoints(){
153     //this function takes all the notes that are being branched on and adds them one after the
154     //other to the solution array
155     int idx = 0;
156     for(int i = 0; i < counterpoint_1->getBranchingNotes().size(); i++){
157         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_1->getBranchingNotes()[i]);
158         idx++;
159     }
160     for(int i = 0; i < counterpoint_2->getBranchingNotes().size(); i++){
161         rel(*this, solutionArray[idx], IRT_EQ, counterpoint_2->getBranchingNotes()[i]);
162         idx++;
163     }
164 }
165
166 void ThreeVoiceCounterpoint::uniteCosts(){
167     //this function takes the costs that are present for each species and some 3v specific
168     //costs and adds them together
169     int cp1_idx = 0;
170     int cp2_idx = 0;
171     for(int i = 0; i < 14; i++){
172         //goes through every possible cost
173         string name = importanceNames[i];
174         //check seperately for the successive cost
175         if(name=="succ"){
176             unitedCostNames.push_back(name);
177             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(successiveCostArray)
178             )));
179         } else if(name=="triad"){ //check seperately for the triad cost
180             unitedCostNames.push_back(name);
181             rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(IntVarArgs(triadCostArray))));
182         } else {
183             //decides if the cost is present for any counterpoint
184
185             bool cp1_contains = 0;
186             bool cp2_contains = 0;
187             int sz = 0;
188             for(int t = 0; t < counterpoint_1->getCostNames().size(); t++){
189                 if(name==counterpoint_1->getCostNames()[t]){

```

```

188         cp1_contains=1;
189         sz++;
190     }
191 }
192 for(int t = 0; t < counterpoint_2->getCostNames().size(); t++){
193     if(name==counterpoint_2->getCostNames()[t]){
194         cp2_contains=1;
195         sz++;
196     }
197 }
198 //if the cost is present in no counterpoint -> add a specific name to the array
containing all present cost names
199 if(!cp1_contains && !cp2_contains){
200     unitedCostNames.push_back("NOT ADDED");
201 } else {
202     //if it is in either counterpoint -> add the cost
203     unitedCostNames.push_back(name);
204     //adds the cost to the IntVarArgs
205     IntVarArgs x(sz);
206     int idx=0;
207     if(cp1_contains){
208         x[idx] = counterpoint_1->getCosts()[cp1_idx];
209         idx++;
210         cp1_idx++;
211     }
212     if(cp2_contains){
213         x[idx] = counterpoint_2->getCosts()[cp2_idx];
214         idx++;
215         cp2_idx++;
216     }
217     //sum the cost together if it are present in both counterpoints
218     rel(*this, unitedCosts[i], IRT_EQ, expr(*this, sum(x)));
219 }
220 }
221 }
222 }

```

C.4.4 TwoVoiceCounterpoint.cpp

```

1 //
2 // Created by Damien Sprockeels on 13/06/2024.
3 // Extended and developed by Luc Cleenewerk and Diego de Patoul up to August 2024.
4 //
5
6 #include "../headers/CounterpointProblems/TwoVoiceCounterpoint.hpp"
7 #include "../headers/CounterpointUtils.hpp"
8
9 /**
10  * Constructor of the class.
11  * @param cf a vector<int> representing the cantus firmus.
12  * @param sp the species of the counterpoint. it takes values from the enum "species" in
headers/Utilities.hpp
13  * @param k the key of the score. it takes values from the notes in headers/Utilities.hpp
14  * @param lb the lowest note possible for the counterpoint in MIDI
15  * @param ub the highest note possible for the counterpoint in MIDI
16  */
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
23     counterpoint_1 = create_counterpoint(*this, species, nMeasures, cf, (6 * v_type - 6) + cf
[0], (6 * v_type + 12) + cf[0], lowest, cantusFirmus,
24         v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
25
26     // G6 : no chromatic melodies (works for 1st, 2nd and 3rd species)
27     G6_noChromaticMelodies(*this, counterpoint_1, species);
28
29     //G9

```

```

30 G9_lastChordSameAsFundamental(*this, lowest, cantusFirmus);
31
32 /// H2 from Thibault: The first harmonic interval must be a perfect consonance
33 if(species!=FIFTH_SPECIES){
34     //we check for fifth species since it always starts with a break, so it doesn't make
35     //sense to apply the constraint in this case
36     H2_1_startWithPerfectConsonance(*this, counterpoint_1);
37 }
38
39 /// H3 from Thibault: The last harmonic interval must be a perfect consonance
40 H3_1_endWithPerfectConsonance(*this, counterpoint_1);
41
42 // H5 from Thibault : The cp and the cf cannot play the same note
43 H5_1_cpAndCfDifferentNotes(*this, counterpoint_1, cantusFirmus);
44
45 setStrata();
46
47 unitedCosts = IntVarArray(*this, counterpoint_1->getCosts().size(), 0, 1000000);
48
49 for(int i = 0; i < unitedCosts.size(); i++){
50     rel(*this, unitedCosts[i], IRT_EQ, counterpoint_1->getCosts()[i]);
51 }
52
53 unitedCostNames = counterpoint_1->getCostNames();
54
55 orderCosts();
56
57 solutionArray = IntVarArray(*this, counterpoint_1->getBranchingNotes().size(), 0, 127);
58 rel(*this, solutionArray, IRT_EQ, counterpoint_1->getBranchingNotes());
59
60 branch(*this, lowest->getNotes().slice(0, 4/notesPerMeasure.at(FIRST_SPECIES), lowest->
61 getNotes().size()), INT_VAR_DEGREE_MAX(), INT_VAL_SPLIT_MIN());
62 if(species==FIFTH_SPECIES){
63     branch(*this, counterpoint_1->getSpeciesArray(), INT_VAR_DEGREE_MAX(), INT_VAL_RND(3U)
64 );
65 }
66 if(species==FIFTH_SPECIES){
67     branch(*this, counterpoint_1->getCambiataCostArray(), INT_VAR_DEGREE_MAX(),
68     INT_VAL_MIN());
69 }
70 if(species==FOURTH_SPECIES || species==FIFTH_SPECIES){
71     branch(*this, counterpoint_1->getSyncopeCostArray(), INT_VAR_DEGREE_MAX(),
72     INT_VAL_MIN());
73 }
74 branch(*this, solutionArray, INT_VAR_SIZE_MIN(), INT_VAL_MIN());
75
76 }
77
78 // COPY CONSTRUCTOR
79 TwoVoiceCounterpoint::TwoVoiceCounterpoint(TwoVoiceCounterpoint& s) : CounterpointProblem(s){
80     species = s.species;
81 }
82
83 IntLexMinimizeSpace* TwoVoiceCounterpoint::copy(){
84     return new TwoVoiceCounterpoint(*this);
85 }
86
87
88 string TwoVoiceCounterpoint::to_string() const {
89     string text = "";
90     text += CounterpointProblem::to_string();
91     text += "Lowest : \n";
92     text += cantusFirmus->to_string();
93     text += "\n";
94     text += "Counterpoint 1 : \n";
95     text += counterpoint_1->to_string();
96     text += "\n";
97     text += "Solution Array : \n";
98     text += intVarArray_to_string(solutionArray);
99     text += "\n";
100    text += counterpoint_1->to_string();
101    return text;
102 }

```

C.5 Voice source files

C.5.1 CantusFirmus.cpp

```
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 #include "../headers/Parts/CantusFirmus.hpp"
7
8 CantusFirmus::CantusFirmus(Home home, int size, vector<int> cf, Stratum* low, int v_type,
9     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int nV) :
10     Part(home, size, CANTUS_FIRMUS, cf, 0, 127, v_type, m_costs, g_costs, s_costs, nV, -1){
11     cf_vector = cf;
12     notes = IntVarArray(home, size, lowerBound, upperBound);
13     /// Melodic intervals for the first species notes
14     m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
15     h_intervals = IntVarArray(home, size, -PERFECT_OCTAVE, PERFECT_OCTAVE);
16     ///link melodic intervals
17     for(int i = 0; i < m_intervals_brut.size(); i++){
18         rel(home, m_intervals_brut[i], IRT_EQ, expr(home, notes[i+1] - notes[i]));
19     }
20     for(int i = 0; i < size; i++){
21         rel(home, h_intervals[i], IRT_EQ, expr(home, (notes[i]-low->getFirstNotes()[i])%12));
22     }
23     for(int i = 0; i < size; i++){
24         rel(home, notes[i], IRT_EQ, cf_vector[i]);
25     }
26     motions = IntVarArray(home, notes.size()-1, IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION,
27         PARALLEL_MOTION});
28     //create motions
29
30     for(int i = 0; i < motions.size(); i++){
31         ///direct motions help creation
32
33         BoolVar both_up = expr(home, (m_intervals_brut[i]>0)&&(low->getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
34         BoolVar both_stay = expr(home, (m_intervals_brut[i]==0)&&(low->getMelodicIntervals()[i]==0)); //if both parts are staying
35         BoolVar both_down = expr(home, (m_intervals_brut[i]<0)&&(low->getMelodicIntervals()[i]<0)); //if both parts are going down
36         //oblique motions help creation
37         BoolVar cf_stays_1 = expr(home, (m_intervals_brut[i]>0)&&(low->getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
38         BoolVar cf_stays_2 = expr(home, (m_intervals_brut[i]<0)&&(low->getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
39         BoolVar cp_stays_1 = expr(home, (m_intervals_brut[i]==0)&&(low->getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
40         BoolVar cp_stays_2 = expr(home, (m_intervals_brut[i]==0)&&(low->getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
41         //contrary motions help creation
42         BoolVar cpd_cfu = expr(home, (m_intervals_brut[i]<0)&&(low->getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
43         BoolVar cpu_cfd = expr(home, (m_intervals_brut[i]>0)&&(low->getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
44
45         //direct constraints
46         rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >> (motions[i]==PARALLEL_MOTION));
47         //oblique constraints
48         rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]==1)) >> (motions[i]==OBLIQUE_MOTION));
49         //contrary constraints
50         rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (motions[i]==CONTRARY_MOTION));
51         //bass constraints
52         rel(home, (isNotLowest[i]==0) >> (motions[i]==-1));
53     }
54
55     dom(home, h_intervals, IntSet({UNISSON, MINOR_THIRD, MAJOR_THIRD, PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
```

```

54     -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -
    PERFECT_OCTAVE}));
55
56     if(nV==TWO_VOICES){
57         /// H2 from Thibault: The first harmonic interval must be a perfect consonance
58         dom(home, h_intervals[0], IntSet(IntArgs(PERFECT_CONSONANCES)));
59
60         /// H3 from Thibault: The last harmonic interval must be a perfect consonance
61         dom(home, h_intervals[h_intervals.size()-1], IntSet(IntArgs(PERFECT_CONSONANCES)));
62
63         ///H7,H8 cf version
64         rel(home, h_intervals[h_intervals.size()-2], IRT_EQ, MINOR_THIRD, Reify(isNotLowest[
        isNotLowest.size()-2], RM_IMP));
65
66         ///P1 from Thibault : Perfect consonances cannot be reached by direct motion
67         for(int j = 0; j < motions.size(); j++){
68
69             rel(home, ((h_intervals[j+1]==UNISSON || h_intervals[j+1]==PERFECT_FIFTH)&&
        isNotLowest[j+1]==1) >>
70                 (motions[j]!=PARALLEL_MOTION));
71
72         }
73     } else {
74         ///H7,H8 cf version, 3v adapted
75         rel(home, expr(home, abs(h_intervals[h_intervals.size()-2]))==UNISSON||expr(home, abs(
        h_intervals[h_intervals.size()-2]))==MINOR_THIRD
76             ||expr(home, abs(h_intervals[h_intervals.size()-2]))==PERFECT_FIFTH||expr(home,
        abs(h_intervals[h_intervals.size()-2]))==MAJOR_SIXTH);
77     }
78
79     //No battuta
80     P3_0_noBattuta(home, this);
81 }
82
83 string CantusFirmus::to_string() const {
84     string text = "Cantus Firmus notes : ";
85     text += intVarArray_to_string(notes);
86     text += "\n";
87     text += "H intervals brut : ";
88     text += intVarArray_to_string(h_intervals);
89     text += "\n";
90     text += "Motions : ";
91     text += intVarArray_to_string(motions);
92     text += "\n";
93     text += "is Lowest : ";
94     text += boolVarArray_to_string(isNotLowest);
95     text += "\n";
96     text += "is Highest : ";
97     text += boolVarArray_to_string(isHighest);
98     text += "\n";
99     return text;
100 }
101
102
103 // clone constructor
104 CantusFirmus::CantusFirmus(Home home, CantusFirmus &s) : Part(home, s){
105
106 }
107
108 CantusFirmus* CantusFirmus::clone(Home home){
109     return new CantusFirmus(home, *this);
110 }
111
112 IntVarArray CantusFirmus::getFirstHInterval(){
113     return h_intervals;
114 }
115
116 IntVarArray CantusFirmus::getMotions(){
117     return motions;
118 }
119
120 IntVarArray CantusFirmus::getFirstMInterval(){
121     return m_intervals_brut;

```

```

122 }
123
124 IntVarArgs CantusFirmus::getFirstNotes(){
125     return notes.slice(0, 4/4, notes.size());
126 }
127
128 int CantusFirmus::getHIntervalSize(){
129     return h_intervals.size();
130 }

```

C.5.2 FifthSpeciesCounterpoint.cpp

```

1 //
2 // 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
11 , int ub, Species mSpecies, Stratum* low, CantusFirmus* c,
12 int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV)
13 :
14 Part(home, nMes, mSpecies, cf, lb, ub, v_type, m_costs, g_costs, s_costs, nV, bm)
15 {
16     for(int i = lowerBound; i <= upperBound; i++){
17         cp_range.push_back(i);
18     }
19     cout << lowerBound << endl;
20     cout << upperBound << endl;
21     /*
22     if borrowMode is enabled, the extended domain is extended to make the inclusion of
23     borrowed notes possible. We can see from Fux's examples
24     that he does like to borrow notes, so the borrow cost should just do the job and still
25     allow borrowed notes, not outright forbid them
26     */
27     if(borrowMode==1){
28         extended_domain = vector_union(cp_range, vector_union(scale, borrowed_scale));
29     } else {
30         extended_domain = vector_intersection(cp_range, vector_union(scale, borrowed_scale));
31     }
32
33     off_domain = vector_difference(vector_intersection(cp_range, scale), lowerBound,
34 upperBound);
35
36     solutionLength = notes.size();
37     speciesArray = IntVarArray(home, solutionLength, IntSet({-1, THIRD_SPECIES, FOURTH_SPECIES
38 }));
39     isNthSpeciesArray = BoolVarArray(home, notes.size()*5, 0, 1);
40     isConstrainedArray = BoolVarArray(home, solutionLength, 0, 1);
41
42     /**
43      * CREATE THE SPECIES ARRAY
44      */
45     createSpeciesArrays(home);
46
47     fifthSpeciesNotesCp = IntVarArray(home, notes.size(), IntSet(IntArgs(vector_intersection(
48 cp_range, extended_domain))));
49     if(borrowMode==1){
50         fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(
51 vector_intersection(cp_range, chromatic_scale))));
52     }
53
54     fifthSpeciesHIntervals = IntVarArray(home, h_intervals.size(), -PERFECT_OCTAVE,
55 PERFECT_OCTAVE);
56
57

```

```

50 firstSpeciesHarmonicIntervals = IntVarArray(home, fifthSpeciesHIntervals.slice(0, 4,
51 fifthSpeciesHIntervals.size()));
52
53 for(int i = 0; i < fifthSpeciesHIntervals.size(); i++){
54     rel(home, (fifthSpeciesHIntervals[i])==(fifthSpeciesNotesCp[i]-low->getNotes()[floor(
55 i/4)*4])%12));
56 }
57 //rel(home, (fifthSpeciesHIntervals[0])==(fifthSpeciesNotesCp[2]-low->getNotes()[0])%12)
58 ;
59
60 fifthSpeciesSuccMIntervals = IntVarArray(home, m_intervals_brut.size(), -PERFECT_OCTAVE,
61 PERFECT_OCTAVE);
62
63 firstSpeciesMelodicIntervals = IntVarArray(home, nMeasures-1, -PERFECT_OCTAVE,
64 PERFECT_OCTAVE);
65
66 for(int i = 0; i < fifthSpeciesSuccMIntervals.size(); i++){
67     rel(home, fifthSpeciesSuccMIntervals[i] == (fifthSpeciesNotesCp[i+1]-
68 fifthSpeciesNotesCp[i]));
69 }
70
71 for(int i = 0; i < firstSpeciesMelodicIntervals.size(); i++){
72     rel(home, firstSpeciesMelodicIntervals[i] == (fifthSpeciesNotesCp[(i+1)*4]-
73 fifthSpeciesNotesCp[i*4]));
74 }
75
76 fifthSpeciesMIntervals = IntVarArray(home, m_intervals_brut.size(), -16, 16);
77 for(int i = 0; i < fifthSpeciesMIntervals.size(); i+=4){
78     rel(home, fifthSpeciesMIntervals[i+2] == (fifthSpeciesNotesCp[i+4]-fifthSpeciesNotesCp
79 [i+2]));
80     rel(home, fifthSpeciesMIntervals[i+3] == (fifthSpeciesNotesCp[i+4]-fifthSpeciesNotesCp
81 [i+3]));
82 }
83
84 fifthSpeciesMTAIntervals = IntVarArray(home, nMeasures-1, -16, 16);
85 for(int i = 0; i < fifthSpeciesMTAIntervals.size(); i++){
86     rel(home, fifthSpeciesMTAIntervals[i] == (fifthSpeciesNotesCp[(i*4)+2]-
87 fifthSpeciesNotesCp[(i*4)]));
88 }
89
90 m2Len = ((nMeasures-1)*4)-1;
91 fifthSpeciesM2Intervals = IntVarArray(home, m2Len, -16, 16);
92 for(int i = 0; i < m2Len; i++){
93     rel(home, fifthSpeciesM2Intervals[i] == (fifthSpeciesNotesCp[i+2]-fifthSpeciesNotesCp[
94 i]));
95 }
96
97 fifthSpeciesMallIntervals = IntVarArray(home, m_intervals_brut.size(), -PERFECT_OCTAVE,
98 PERFECT_OCTAVE);
99 for(int i = 0; i < fifthSpeciesMallIntervals.size(); i++){
100     rel(home, fifthSpeciesMallIntervals[i] == (fifthSpeciesNotesCp[i+1]-
101 fifthSpeciesNotesCp[i]));
102 }
103
104 fifthSpeciesMotions = IntVarArray(home, nMeasures-1, IntSet({-1, CONTRARY_MOTION,
105 OBLIQUE_MOTION, PARALLEL_MOTION}));
106 fifthSpeciesMotionsCosts = IntVarArray(home, nMeasures-1, IntSet({0, contraryCost,
107 obliqueCost, directCost}));
108
109 //create motions
110 for(int i = 0; i < fifthSpeciesMotions.size(); i++){
111     //direct motions help creation
112
113     BoolVar both_up = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low->
114 getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
115     BoolVar both_stay = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(low->
116 getMelodicIntervals()[i]==0)); //if both parts are staying
117     BoolVar both_down = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low->
118 getMelodicIntervals()[i]<0)); //if both parts are going down
119     //oblique motions help creation
120     BoolVar cf_stays_1 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low->
121 getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up

```

```

103     BoolVar cf_stays_2 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low->
getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
104     BoolVar cp_stays_1 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(low->
getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
105     BoolVar cp_stays_2 = expr(home, (fifthSpeciesMIntervals[(i*4)+3]==0)&&(low->
getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
106     //contrary motions help creation
107     BoolVar cpd_cfu = expr(home, (fifthSpeciesMIntervals[(i*4)+3]<0)&&(low->
getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
108     BoolVar cpu_cfd = expr(home, (fifthSpeciesMIntervals[(i*4)+3]>0)&&(low->
getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
109
110     //direct constraints
111     rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
fifthSpeciesMotions[i]==PARALLEL_MOTION));
112     rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
fifthSpeciesMotionsCosts[i]==directCost));
113     //oblique constraints
114     rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->
isNotLowest[i]==1)) >> (fifthSpeciesMotions[i]==OBLIQUE_MOTION));
115     rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->
isNotLowest[i]==1)) >> (fifthSpeciesMotionsCosts[i]==obliqueCost));
116     //contrary constraints
117     rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (fifthSpeciesMotions[
i]==CONTRARY_MOTION));
118     rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
fifthSpeciesMotionsCosts[i]==contraryCost));
119     //bass constraints
120     rel(home, (this->isNotLowest[i]==0) >> (fifthSpeciesMotions[i]==-1));
121     rel(home, (this->isNotLowest[i]==0) >> (fifthSpeciesMotionsCosts[i]==0));
122
123 }
124
125 is5QNSArray = BoolVarArray(home, nMeasures-1, 0, 1);
126
127 for(int i = 0; i < is5QNSArray.size()*4; i+=4){
128
129     BoolVar b1 = BoolVar(home, 0, 1);
130     BoolVar b2 = BoolVar(home, 0, 1);
131     BoolVar b3 = BoolVar(home, 0, 1);
132     BoolVar b4 = BoolVar(home, 0, 1);
133     BoolVar bb1 = BoolVar(home, 0, 1);
134     BoolVar bb2 = BoolVar(home, 0, 1);
135     BoolVar bb3 = BoolVar(home, 0, 1);
136     BoolVar bb4 = BoolVar(home, 0, 1);
137     BoolVar band1 = BoolVar(home, 0, 1);
138     BoolVar band2 = BoolVar(home, 0, 1);
139     BoolVar band3 = BoolVar(home, 0, 1);
140     BoolVar beq1 = BoolVar(home, 0, 1);
141     BoolVar beq2 = BoolVar(home, 0, 1);
142     BoolVar beq3 = BoolVar(home, 0, 1);
143
144     rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i])), IRT_LQ, 2, Reify(b1));
145     rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+1])), IRT_LQ, 2, Reify(b2));
146     rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+2])), IRT_LQ, 2, Reify(b3));
147     rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i+3])), IRT_LQ, 2, Reify(b4));
148
149     rel(home, fifthSpeciesSuccMIntervals[i], IRT_GQ, 0, Reify(bb1));
150     rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_GQ, 0, Reify(bb2));
151     rel(home, fifthSpeciesSuccMIntervals[i+2], IRT_GQ, 0, Reify(bb3));
152     rel(home, fifthSpeciesSuccMIntervals[i+3], IRT_GQ, 0, Reify(bb4));
153
154     rel(home, b1, BOT_AND, b2, band1);
155     rel(home, b3, BOT_AND, b4, band2);
156     rel(home, band1, BOT_AND, band2, band3);
157
158     rel(home, bb1, BOT_EQV, bb2, beq1);
159     rel(home, bb3, BOT_EQV, bb4, beq2);
160     rel(home, beq1, BOT_EQV, beq2, beq3);
161     rel(home, band3, BOT_AND, beq3, is5QNSArray[i/4]);
162 }
163
164 isMostlyThirdArray = BoolVarArray(home, nMeasures-1, 0, 1);

```

```

165     for(int i = 0; i < isMostlyThirdArray.size(); i++){
166         BoolVar b23 = BoolVar(home, 0, 1);
167         /**
168          * EXPLANATION :
169          * -for each measure there are 20 entries in the isNthSpeciesArray (5 possibilities
170          * for each note, and a measure has 4 notes)
171          * -therefor, we multiply i by 20 to always land at the start of a measure (0 is the
172          * first note of the first measure, 20 the first of the second
173          * measure etc etc)
174          * -since each note has 5 entries to check for which type of species it is, to land on
175          * the first entry of the second note, we do 20+5
176          * -now to check if the second note is of the third species, we do 20+5+3 (second
177          * measure, second note, check for third species)
178          */
179         rel(home, isNthSpeciesArray[(i*20)+8], BOT_AND, isNthSpeciesArray[(i*20)+13], b23);
180         rel(home, b23, BOT_AND, isNthSpeciesArray[(i*20)+18], isMostlyThirdArray[i]);
181     }
182
183     isConsonance = BoolVarArray(home, notes.size(), 0, 1);
184     isThirdSpeciesArray = BoolVarArray(home, notes.size(), 0, 1);
185     isFourthSpeciesArray = BoolVarArray(home, notes.size(), 0, 1);
186
187     //create isConsonance array
188     for(int i = 0; i < isConsonance.size(); i++){
189         rel(home, expr(home, (h_intervals[i]==UNISSON)|| (h_intervals[i]==MINOR_THIRD)|| (
190             h_intervals[i]==MAJOR_THIRD)|| (h_intervals[i]==PERFECT_FIFTH)||
191             (h_intervals[i]==MINOR_SIXTH)|| (h_intervals[i]==MAJOR_SIXTH)|| (h_intervals[i]==
192             PERFECT_OCTAVE)|| (h_intervals[i]==-MINOR_THIRD)||
193             (h_intervals[i]==-MAJOR_THIRD)|| (h_intervals[i]==-PERFECT_FIFTH)|| (h_intervals[i]
194             )==MINOR_SIXTH)|| (h_intervals[i]==-MAJOR_SIXTH)||
195             (h_intervals[i]==-PERFECT_OCTAVE)), IRT_EQ, isConsonance[i]));
196     }
197
198     //create isThird and isFourth arrays
199     for(int i = 0; i < notes.size(); i++){
200         rel(home, isThirdSpeciesArray[i], IRT_EQ, isNthSpeciesArray[(i*5)+3]);
201         rel(home, isFourthSpeciesArray[i], IRT_EQ, isNthSpeciesArray[(i*5)+4]);
202     }
203
204     isDiminution = BoolVarArray(home, nMeasures-1, 0, 1);
205     for(int i = 0; i < isDiminution.size()*4; i+=4){
206
207         BoolVar btt3 = BoolVar(home, 0, 1);
208         BoolVar btt4 = BoolVar(home, 0, 1);
209         BoolVar bta2nd = BoolVar(home, 0, 1);
210         BoolVar btt3rd = BoolVar(home, 0, 1);
211         BoolVar bat2nd = BoolVar(home, 0, 1);
212         BoolVar band = BoolVar(home, 0, 1);
213         IntVar addition = expr(home, abs(fifthSpeciesM2Intervals[i+1]));
214         rel(home, addition, IRT_EQ, 3, Reify(btt3));
215         rel(home, addition, IRT_EQ, 4, Reify(btt4));
216         rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LQ, 2, Reify(bta2nd));
217         rel(home, fifthSpeciesSuccMIntervals[i+2], IRT_LQ, 2, Reify(bat2nd));
218         rel(home, btt3, BOT_OR, btt4, btt3rd);
219         rel(home, bta2nd, BOT_AND, btt3rd, band);
220         rel(home, band, BOT_AND, bat2nd, isDiminution[i/4]);
221     }
222
223     //set cambiata array
224     isNotCambiata = BoolVarArray(home, nMeasures-1, 0, 1);
225     for(int i = 0; i < isNotCambiata.size(); i++){
226         rel(home, ((fifthSpeciesHIntervals[(i*4)+1]==UNISSON || fifthSpeciesHIntervals[(i*4)
227             +1]==PERFECT_FIFTH)&&(fifthSpeciesHIntervals[(i*4)+2]==UNISSON || fifthSpeciesHIntervals[(
228             i*4)+2]==PERFECT_FIFTH)
229             &&(abs(fifthSpeciesSuccMIntervals[(i*4)+1])<=2)) >> (isNotCambiata[i]==1));
230         rel(home, ((fifthSpeciesHIntervals[(i*4)+1]!=UNISSON && fifthSpeciesHIntervals[(i*4)
231             +1]!=PERFECT_FIFTH)|| (fifthSpeciesHIntervals[(i*4)+2]!=UNISSON && fifthSpeciesHIntervals[(
232             i*4)+2]!=PERFECT_FIFTH)
233             || (abs(fifthSpeciesSuccMIntervals[(i*4)+1])>2)) >> (isNotCambiata[i]==0));
234     }
235
236

```

```

227 is_off = BoolVarArray(home, notes.size(), 0, 1);
228 initializeIsOffArray(home, this);
229
230 isNoSyncopeArray = BoolVarArray(home, nMeasures-1, 0, 1);
231 for(int i = 0; i < isNoSyncopeArray.size(); i++){
232     rel(home, fifthSpeciesMIntervals[(i*2)+2], IRT_NQ, 0, Reify(isNoSyncopeArray[i]));
233 }
234 rel(home, fifthSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(FIFTH_SPECIES),(
    notes.size())));
235 rel(home, fifthSpeciesHIntervals, IRT_EQ, h_intervals.slice(0,4/notesPerMeasure.at(
    FIFTH_SPECIES),h_intervals.size()));
236 rel(home, fifthSpeciesSuccMIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure.
    at(FIFTH_SPECIES),m_intervals_brut.size()));
237
238 /** =====
239 *                                HARMONIC CONSTRAINS
240 * =====
241 */
242
243 //Only one possible value for non constrained variables
244 for(int i = 0; i < fifthSpeciesNotesCp.size()-1; i++){
245     //rel(home, fifthSpeciesNotesCp[i], IRT_EQ, fifthSpeciesNotesCp[i+1], Reify(
    isNthSpeciesArray[(i*5)], RM_IMP));
246 }
247
248 //Third note of penult measure must be below the fourth one
249 rel(home, fifthSpeciesSuccMIntervals[fifthSpeciesSuccMIntervals.size()-3], IRT_GR,
    MINOR_SECOND,
250     Reify(isThirdSpeciesArray[isThirdSpeciesArray.size()-2], RM_IMP));
251 //Second and third note distant by more than one semi tone from fourth note
252 rel(home, expr(home, abs(fifthSpeciesM2Intervals[fifthSpeciesM2Intervals.size()-2])),
    IRT_NQ, 1,
253     Reify(isThirdSpeciesArray[isThirdSpeciesArray.size()-2], RM_IMP));
254
255 //is penult cons to cf
256 BoolVar isPenultConsToCf = BoolVar(home, 0, 1);
257 vector<int> consonances = {0,3,4,7,8,9,-3,-4,-7,-8,-9};
258 IntVarArray res = IntVarArray(home, consonances.size(), 0, 1);
259 IntVar sm = IntVar(home, 0, consonances.size());
260 for(int l = 0; l < consonances.size(); l++){
261     BoolVar b1 = BoolVar(home, 0, 1);
262     rel(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-5], IRT_EQ, consonances[l],
    Reify(b1));
263     ite(home, b1, IntVar(home, 1, 1), IntVar(home, 0, 0), res[l]);
264 }
265 IntVarArgs x(res.size());
266 for(int t = 0; t < consonances.size(); t++){
267     x[t] = res[t];
268 }
269 rel(home, sm, IRT_EQ, expr(home, sum(x)));
270 rel(home, sm, IRT_GR, 0, Reify(isPenultConsToCf));
271
272 rel(home, isFourthSpeciesArray[isFourthSpeciesArray.size()-5], BOT_AND, isPenultConsToCf,
    0);
273
274 //every thesis note should be consonant
275 for(int i = 0; i < isConsonance.size(); i+=4){ //checks every thesis note
276     rel(home, isNthSpeciesArray[(i*5)+1], BOT_IMP, isConsonance[i], 1); //first species
    check
277     rel(home, isNthSpeciesArray[(i*5)+2], BOT_IMP, isConsonance[i], 1); //second species
    check
278     rel(home, isThirdSpeciesArray[i], BOT_IMP, isConsonance[i], 1); //third species
    check
279     if(i!=isConsonance.size()-1){
280         //rel(home, isFourthSpeciesArray[i+2], BOT_IMP, isConsonance[i+2], 1); //fourth
        species check
281     }
282     if(i!=0 && i!=isConsonance.size()-1){
283         //rel(home, expr(home, isFourthSpeciesArray[i]==1 && isNoSyncopeArray[i]==0),
        BOT_IMP, isConsonance[i],1);
284     }
285 }
286

```

```

287     for(int i = 4; i < isConsonance.size()-1; i+=4){ //start at the first note of the second
        measure
288         rel(home, expr(home, isFourthSpeciesArray[i]==1 && isNoSyncopeArray[i/4]==1), BOT_IMP,
            isConsonance[i], 1);
289     }
290
291     //five consecutive notes by joint degree (3rd species)
292     for(int i = 0; i < is5QNArray.size(); i++){
293         BoolVar b = BoolVar(home, 0, 1);
294         rel(home, is5QNArray[i], BOT_AND, isMostlyThirdArray[i], b);
295         rel(home, b, BOT_IMP, isConsonance[i*4+2], 1);
296     }
297
298     //any dissonant note implies it is surrounded by consonant notes (3rd species)
299     for(int i = 0; i < isDiminution.size(); i++){
300         BoolVar band1 = BoolVar(home, 0, 1);
301         rel(home, expr(home, isConsonance[(i*4)+2] || isThirdSpeciesArray[(i*4)+2]), BOT_OR,
            isDiminution[i], 1);
302     }
303
304     /** =====
305      *                               MELODIC CONSTRAINTS
306      * =====
307     */
308
309     //no melodic interval between MAJOR_SIXTH and MAJOR_SEVENTH
310     for(int i = 0; i < fifthSpeciesMallIntervals.size(); i++){
311         rel(home, expr(home, abs(fifthSpeciesMallIntervals[i])), IRT_NQ, MAJOR_SIXTH, Reify(
            expr(home, isConstrainedArray[i]==1 && isConstrainedArray[i+1]==1), RM_IMP));
312         rel(home, expr(home, abs(fifthSpeciesMallIntervals[i])), IRT_NQ, MINOR_SEVENTH, Reify(
            expr(home, isConstrainedArray[i]==1 && isConstrainedArray[i+1]==1), RM_IMP));
313         rel(home, expr(home, abs(fifthSpeciesMallIntervals[i])), IRT_NQ, MAJOR_SEVENTH, Reify(
            expr(home, isConstrainedArray[i]==1 && isConstrainedArray[i+1]==1), RM_IMP));
314     }
315
316     //no unisson between two consecutive notes
317     for(int i = 0; i < nMeasures-1; i++){
318         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));
319         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))
            );
320     }
321
322     //no more than minor sixth interval between arsis and thesis notes
323     for(int i = 0; i < nMeasures-1; i++){
324         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ, MAJOR_SIXTH, Reify(
            expr(home, isConstrainedArray[i+1]==1), RM_IMP));
325         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ, MINOR_SEVENTH, Reify(
            expr(home, isConstrainedArray[i+1]==1), RM_IMP));
326         rel(home, expr(home, abs(fifthSpeciesMTAIntervals[i])), IRT_NQ, MAJOR_SEVENTH, Reify(
            expr(home, isConstrainedArray[i+1]==1), RM_IMP));
327     }
328
329     //no same syncopation
330     for(int i = 1; i < nMeasures-1; i++){
331         rel(home, fifthSpeciesNotesCp[(i*4)], IRT_NQ, fifthSpeciesNotesCp[(i*4)+2],
332             Reify(expr(home, isFourthSpeciesArray[(i*4)]==1 && isConstrainedArray[(i*4)+2]),
            RM_IMP));
333
334         rel(home, (isFourthSpeciesArray[(i*4)]==1 && isConstrainedArray[(i*4)+2]==1) >> (
            fifthSpeciesNotesCp[(i*4)]!=fifthSpeciesNotesCp[(i*4)+2]));
335     }
336
337     /** =====
338      *                               MOTION CONSTRAINTS
339      * =====
340     */
341
342     //no battuta kind of motion
343     for(int i = 0; i < fifthSpeciesMotions.size(); i++){
344         BoolVar isCM = BoolVar(home, 0, 1);
345         BoolVar isOct = BoolVar(home, 0, 1);

```

```

346 BoolVar isCPDown = BoolVar(home, 0, 1);
347 BoolVar band1 = BoolVar(home, 0, 1);
348 BoolVar band2 = BoolVar(home, 0, 1);
349 BoolVar band3 = BoolVar(home, 0, 1);
350
351 rel(home, fifthSpeciesMotions[i], IRT_EQ, CONTRARY_MOTION, Reify(isCM));
352 rel(home, expr(home, (fifthSpeciesHIntervals[((i+1)*4)]==UNISSON ||
fifthSpeciesHIntervals[((i+1)*4)]==PERFECT_OCTAVE ||
353 fifthSpeciesHIntervals[((i+1)*4)]==PERFECT_OCTAVE)) >> isOct);
354 rel(home, fifthSpeciesMIntervals[(i*4)+3], IRT_LE, -4, Reify(isCPDown));
355
356 rel(home, isCM, BOT_AND, isOct, band1);
357 rel(home, band1, BOT_AND, isCPDown, band2);
358 rel(home, band2, BOT_AND, expr(home, c->getNotes()[i] <= fifthSpeciesNotesCp[(i*4)+3])
, band3);
359 rel(home, isThirdSpeciesArray[(i*4)+3], BOT_AND, band3, 0);
360 }
361
362 //dissonant notes must be followed by the consonant note below
363 for(int i = 0; i < fifthSpeciesMTAIntervals.size(); i++){
364 //isFourthArray first note
365 //is Consonance array first note
366 //mTa intervals, general
367 BoolVar bNot = BoolVar(home, 0, 1);
368 BoolVar isCst = BoolVar(home, 0, 1);
369 BoolVar bAnd = BoolVar(home, 0, 1);
370
371 rel(home, isConsonance[i*4], BOT_EQV, BoolVar(home, 0, 0), bNot);
372 rel(home, bNot, BOT_AND, isFourthSpeciesArray[i*4], bAnd);
373 rel(home, fifthSpeciesMTAIntervals[i], IRT_LE, 0, Reify(bAnd, RM_IMP));
374 rel(home, fifthSpeciesMTAIntervals[i], IRT_GQ, -2, Reify(bAnd, RM_IMP));
375 }
376
377 //no second dissonant note if the cantusFirmus is at the bass
378 for(int i = 0; i < nMeasures-1; i++){
379 BoolVar bUni = BoolVar(home, 0, 1);
380 BoolVar bAnd = BoolVar(home, 0, 1);
381 BoolVar isCst = BoolVar(home, 0, 1);
382 BoolVar bAndCst = BoolVar(home, 0, 1);
383
384 rel(home, isFourthSpeciesArray[((i+1)*4)], BOT_EQV, isCst, 1);
385 rel(home, fifthSpeciesHIntervals[(i*4)+2], IRT_EQ, 0, Reify(bUni));
386 rel(home, isNotLowest[i], BOT_AND, bUni, bAnd);
387 rel(home, bAnd, BOT_AND, isCst, bAndCst);
388 rel(home, fifthSpeciesHIntervals[((i+1)*4)], IRT_NQ, 1, Reify(bAndCst, RM_IMP));
389 rel(home, fifthSpeciesHIntervals[((i+1)*4)], IRT_NQ, 2, Reify(bAndCst, RM_IMP));
390 }
391
392 //marcel's rule
393 for(int i = 0; i < fifthSpeciesSuccMIntervals.size()-1; i++){
394 BoolVar bSkip = BoolVar(home, 0, 1);
395 BoolVar bMbUp = BoolVar(home, 0, 1);
396 BoolVar bMbDown = BoolVar(home, 0, 1);
397 BoolVar bContrary = BoolVar(home, 0, 1);
398
399 rel(home, expr(home, abs(fifthSpeciesSuccMIntervals[i])), IRT_GR, 2, Reify(bSkip));
400 rel(home, fifthSpeciesSuccMIntervals[i], IRT_GR, 0, Reify(bMbUp));
401 rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LE, 1, Reify(bMbDown));
402 rel(home, bMbUp, BOT_EQV, bMbDown, bContrary);
403 rel(home, fifthSpeciesSuccMIntervals[i+1], IRT_LQ, 2, Reify(bSkip, RM_IMP));
404 rel(home, bSkip, BOT_IMP, bContrary, 1);
405 }
406
407 /** =====
408 * COST CONSTRAINTS
409 * =====
410 */
411
412 //Imperfect consonances are preferred
413 fifthCostArray = IntVarArray(home, notes.size(), IntSet({0, fifthCost}));
414 octaveCostArray = IntVarArray(home, notes.size(), IntSet({0, octaveCost}));
415
416 //set fifth cost

```

```

417     for(int i = 0; i < notes.size(); i++){
418         BoolVar b = BoolVar(home, 0, 1);
419         BoolVar band = BoolVar(home, 0, 1);
420
421         rel(home, fifthSpeciesHIntervals[i], IRT_EQ, 7, Reify(b));
422         rel(home, b, BOT_AND, isConstrainedArray[i], band);
423         ite(home, band, IntVar(home, fifthCost, fifthCost), IntVar(home, 0, 0), fifthCostArray
424         [i]);
425     }
426
427     //set octave cost
428     for(int i = 0; i < notes.size(); i++){
429         BoolVar b = BoolVar(home, 0, 1);
430         BoolVar band = BoolVar(home, 0, 1);
431         BoolVar band2 = BoolVar(home, 0, 1);
432
433         rel(home, fifthSpeciesHIntervals[i], IRT_EQ, 7, Reify(b));
434         rel(home, isNotLowest[floor(i/4)], BOT_AND, isConstrainedArray[i], band2);
435         rel(home, b, BOT_AND, band2, band);
436         ite(home, band, IntVar(home, octaveCost, octaveCost), IntVar(home, 0, 0),
437         octaveCostArray[i]);
438     }
439
440     //create off_cost array
441     offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
442     //set the cost for borrowing this note (G4 constraint, modified)
443     for(int i = 0; i < is_off.size(); i++){
444         rel(home, (is_off[i]==0) >> (offCostArray[i]==0));
445         rel(home, (is_off[i]==1 && isConstrainedArray[i]==1) >> (offCostArray[i]==borrowCost))
446         ;
447         rel(home, (is_off[i]==1 && isConstrainedArray[i]==0) >> (offCostArray[i]==0));
448     }
449
450     melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({0, secondCost,
451     thirdCost, fourthCost, tritoneCost, fifthCost,
452     sixthCost, seventhCost, octaveCost}));
453
454     //G7 (modified)
455     for(int i = 0; i < m_intervals_brut.size(); i+=4/notesPerMeasure.at(FIFTH_SPECIES)){
456         rel(home, (isConstrainedArray[i]==0 || isConstrainedArray[i+1]==0) >> (
457         melodicDegreeCost[i]==0));
458         rel(home, (abs(m_intervals_brut[i])<MINOR_THIRD && isConstrainedArray[i]==1 &&
459         isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==secondCost));
460         rel(home, ((abs(m_intervals_brut[i])==MINOR_THIRD || abs(m_intervals_brut[i])==
461         MAJOR_THIRD)&& isConstrainedArray[i]==1&& isConstrainedArray[i+1]==1) >> (
462         melodicDegreeCost[i]==thirdCost));
463         rel(home, (abs(m_intervals_brut[i])==PERFECT_FOURTH && isConstrainedArray[i]==1&&
464         isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==fourthCost));
465         rel(home, (abs(m_intervals_brut[i])==TRITONE && isConstrainedArray[i]==1&&
466         isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==tritoneCost));
467         rel(home, (abs(m_intervals_brut[i])==PERFECT_FIFTH && isConstrainedArray[i]==1&&
468         isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==fifthCost));
469         rel(home, ((abs(m_intervals_brut[i])==MINOR_SIXTH || abs(m_intervals_brut[i])==
470         MAJOR_SIXTH)&& isConstrainedArray[i]==1&& isConstrainedArray[i+1]==1) >> (
471         melodicDegreeCost[i]==sixthCost));
472         rel(home, ((abs(m_intervals_brut[i])==MINOR_SEVENTH || abs(m_intervals_brut[i])==
473         MAJOR_SEVENTH)&& isConstrainedArray[i]==1&& isConstrainedArray[i+1]==1) >> (
474         melodicDegreeCost[i]==seventhCost));
475         rel(home, (abs(m_intervals_brut[i])==PERFECT_OCTAVE && isConstrainedArray[i]==1&&
476         isConstrainedArray[i+1]==1) >> (melodicDegreeCost[i]==octaveCost));
477     }
478
479     cambiataCostArray = IntVarArray(home, nMeasures-1, IntSet({0, cambiataCost}));
480
481     for(int i = 0; i < cambiataCostArray.size(); i++){
482         BoolVar band = BoolVar(home, 0, 1);
483         rel(home, isNotCambiata[i], BOT_AND, isMostlyThirdArray[i], band);
484         ite(home, band, IntVar(home, cambiataCost, cambiataCost), IntVar(home, 0, 0),
485         cambiataCostArray[i]);
486     }
487
488     m2ZeroCostArray = IntVarArray(home, m2Len, IntSet({0, m2ZeroCost}));

```

```

473     for(int i = 0; i < fifthSpeciesM2Intervals.size(); i++){
474         BoolVar b = BoolVar(home, 0, 1);
475         BoolVar band = BoolVar(home, 0, 1);
476         BoolVar band2 = BoolVar(home, 0, 1);
477
478         rel(home, fifthSpeciesM2Intervals[i], IRT_EQ, 0, Reify(b));
479         rel(home, isConstrainedArray[i], BOT_AND, isConstrainedArray[i+2], band2);
480         rel(home, b, BOT_AND, band2, band);
481         ite(home, band, IntVar(home, m2ZeroCost, m2ZeroCost), IntVar(home, 0, 0),
482             m2ZeroCostArray[i]);
483     }
484
485     snycopeCostArray = IntVarArray(home, (fifthSpeciesMIntervals.size())/4, IntSet({0,
486         syncopationCost}));
487     for(int i = 0; i < snycopeCostArray.size(); i++){
488         BoolVar b = BoolVar(home, 0, 1);
489         BoolVar band = BoolVar(home, 0, 1);
490
491         rel(home, fifthSpeciesMIntervals[(i*4)+2], IRT_NQ, 0, Reify(b));
492         rel(home, b, BOT_AND, isFourthSpeciesArray[(i*4)+2], band);
493         ite(home, band, IntVar(home, syncopationCost, syncopationCost), IntVar(home, 0, 0),
494             snycopeCostArray[i]);
495     }
496 }
497
498 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
499     , int ub, Stratum* low, CantusFirmus* c, int v_type,
500     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
501     FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type, m_costs,
502         g_costs, s_costs, bm, nV)
503 {
504     //following two rel are H2_1 but modified, so we check in TwoVoiceProblem that that
505     //constrained isn't activated if fifth species
506     //if the first note is constrained, then it must be a perfect consonance (works)
507     rel(home, (isConstrainedArray[0]==1) >> (fifthSpeciesHIntervals[0]==UNISSON ||
508         fifthSpeciesHIntervals[0]==PERFECT_FIFTH));
509     //else if the first note is not constrained, then the third note must be a perfect
510     //consonance (works)
511     rel(home, (isConstrainedArray[0]==0) >> (fifthSpeciesHIntervals[2]==UNISSON ||
512         fifthSpeciesHIntervals[2]==PERFECT_FIFTH));
513
514     //If the cantusFirmus is in the upper part, then no hamonic seventh
515     for(int j = 1; j < c->getIsNotLowest().size(); j++){
516         rel(home, (getIsNotLowest()[j]==0 && isFourthSpeciesArray[j*4]) >> (
517             firstSpeciesHarmonicIntervals[j]!=MINOR_SEVENTH && firstSpeciesHarmonicIntervals[j]!=-
518             MINOR_SEVENTH));
519         rel(home, (getIsNotLowest()[j]==0 && isFourthSpeciesArray[j*4]) >> (
520             firstSpeciesHarmonicIntervals[j]!=MAJOR_SEVENTH && firstSpeciesHarmonicIntervals[j]!=-
521             MAJOR_SEVENTH));
522     }
523
524     //H3_1 can be activated as-is
525
526     //penultimate note (only applies if it is 3rd species)
527     BoolVar bandThen_1 = BoolVar(home, 0, 1);
528     BoolVar testNot_1 = BoolVar(home, 0, 1);
529     BoolVar bandElse_1 = BoolVar(home, 0, 1);
530     BoolVar test_1 = expr(home, c->getNotes()[c->getNotes().size()-2] <= fifthSpeciesNotesCp[
531         fifthSpeciesNotesCp.size()-2]);
532
533     rel(home, test_1, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()-2], bandThen_1);
534     rel(home, test_1, BOT_EQV, BoolVar(home, 0, 0), testNot_1);
535     rel(home, testNot_1, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()-2],
536         bandElse_1);
537
538     rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2]-c->getNotes()[c->
539         getNotes().size()-2]), IRT_EQ, MAJOR_SIXTH, Reify(bandThen_1, RM_IMP));
540     rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-2]-c->getNotes()[c->
541         getNotes().size()-2]), IRT_EQ, MINOR_THIRD, Reify(bandElse_1, RM_IMP));
542
543     //penultimate note (only applies if it is 4th species)
544     BoolVar bandThen = BoolVar(home, 0, 1);
545     BoolVar testNot = BoolVar(home, 0, 1);

```

```

529 BoolVar bandElse = BoolVar(home, 0, 1);
530 BoolVar test = expr(home, c->getNotes()[c->getNotes().size()-2] <= fifthSpeciesNotesCp[
    fifthSpeciesNotesCp.size()-3]);
531
532 rel(home, test, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()-2], bandThen);
533 rel(home, test, BOT_EQV, BoolVar(home, 0, 0), testNot);
534 rel(home, testNot, BOT_AND, isThirdSpeciesArray[isThirdSpeciesArray.size()-2], bandElse);
535 rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-3]-c->getNotes()[c->
    getNotes().size()-2]), IRT_EQ, MAJOR_SIXTH, Reify(bandThen, RM_IMP));
536 rel(home, expr(home, fifthSpeciesNotesCp[fifthSpeciesNotesCp.size()-3]-c->getNotes()[c->
    getNotes().size()-2]), IRT_EQ, MINOR_THIRD, Reify(bandElse, RM_IMP));
537
538 //no direct motion to reach a perfect consonance
539 for(int i = 0; i < fifthSpeciesMotions.size(); i++){
540     BoolVar isThirdSpeciesAndPCons = BoolVar(home, 0, 1);
541     BoolVar isPConsAndNotLowest = BoolVar(home, 0, 1);
542
543     rel(home, isThirdSpeciesArray[i*4], BOT_AND, expr(home, fifthSpeciesHIntervals[(i*4)
        +3]==UNISSON || fifthSpeciesHIntervals[(i*4)+3]==PERFECT_FIFTH ||
        fifthSpeciesHIntervals[(i*4)+3]==-PERFECT_FIFTH), isThirdSpeciesAndPCons);
544     rel(home, isNotLowest[i], BOT_AND, isThirdSpeciesAndPCons, isPConsAndNotLowest);
545
546     rel(home, fifthSpeciesMotions[i], IRT_NQ, PARALLEL_MOTION, Reify(isPConsAndNotLowest,
        RM_IMP));
547
548 }
549
550 costs = IntVarArray(home, 8, 0, 1000000);
551 cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "m2", "
    syncopation"};
552
553 //set cost[0] to be fifth cost
554 add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), fifthCostArray.size()))), costs);
555 //set cost[1] to be octave cost
556 add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), octaveCostArray.size()))), costs);
557 //set cost[2] to be off cost
558 add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), offCostArray.size()))), costs);
559 //set cost[3] to be melodic cost
560 add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    FIFTH_SPECIES), melodicDegreeCost.size()))), costs);
561 //set cost[4] to be motion cost
562 add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
563 //need to set cost[5] to be cambiata cost
564 add_cost(home, 5, cambiataCostArray, costs);
565 //need to set cost[6] to be m2Zero cost
566 add_cost(home, 6, m2ZeroCostArray, costs);
567 //need to set cost[7] to be syncopation cost
568 add_cost(home, 7, snycopeCostArray, costs);
569 }
570
571 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):
572 FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV2)
573 {
574     directCostArray = IntVarArray(home, fifthSpeciesMotions.size()-1, IntSet({0, directMoveCost
        }));
575     varietyCostArray = IntVarArray(home, 3*(fifthSpeciesHIntervals.size()-2), IntSet({0,
        varietyCost}));
576
577 //3.H6 : harmonic triad should be used on the second or third beat
578 thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
579 for(int i = 0; i < thirdHTriadArray.size(); i++){
580     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)&&
581         (fifthSpeciesHIntervals[(i*4)+2]!=UNISSON&&fifthSpeciesHIntervals[(i*4)+2]!=
        MINOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=MAJOR_THIRD&&fifthSpeciesHIntervals[(i*4)
        +2]!=PERFECT_FIFTH)) >>

```

```

584         (thirdHTriadArray[i]==triad3rdCost));
585         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)||
586         (fifthSpeciesHIntervals[(i*4)+2]==UNISSON||fifthSpeciesHIntervals[(i*4)+2]==
MINOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==MAJOR_THIRD||fifthSpeciesHIntervals[(i*4)
+2]==PERFECT_FIFTH)) >>
587         (thirdHTriadArray[i]==0));
588     }
589
590     //1.P1 3 voices version
591     for(int j = 0; j < fifthSpeciesMotions.size()-1; j++){
592         //set a cost when it is reached through direct motion, it is 0 when not
593         rel(home, (fifthSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||
firstSpeciesHarmonicIntervals[j+1]==7))>>
594         (directCostArray[j]==directMoveCost));
595         rel(home, (fifthSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&
firstSpeciesHarmonicIntervals[j+1]!=7))>>
596         (directCostArray[j]==0));
597     }
598
599     costs = IntVarArray(home, 11, 0, 1000000);
600     cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "m2", "
syncopation", "direct", "variety", "triad3"};
601
602     //set cost[0] to be fifth cost
603     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(1, 4/notesPerMeasure.at(
FIFTH_SPECIES), fifthCostArray.size()))), costs);
604     //set cost[1] to be octave cost
605     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(1, 4/notesPerMeasure.at(
FIFTH_SPECIES), octaveCostArray.size()))), costs);
606     //set cost[2] to be off cost
607     add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
FIFTH_SPECIES), offCostArray.size()))), costs);
608     //set cost[3] to be melodic cost
609     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
FIFTH_SPECIES), melodicDegreeCost.size()))), costs);
610     //set cost[4] to be motion cost
611     add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
612     //need to set cost[5] to be cambiata cost
613     add_cost(home, 5, cambiataCostArray, costs);
614     //need to set cost[6] to be m2Zero cost
615     add_cost(home, 6, m2ZeroCostArray, costs);
616     //need to set cost[7] to be syncopation cost
617     add_cost(home, 7, snycopeCostArray, costs);
618     //need to set cost[8] to be syncopation cost
619     add_cost(home, 8, directCostArray, costs);
620     //need to set cost[8] to be syncopation cost
621     add_cost(home, 9, varietyCostArray, costs);
622     //set cost[10] to be triad h third cost
623     add_cost(home, 10, thirdHTriadArray, costs);
624 }
625
626 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
, int ub, Stratum* low, CantusFirmus* c, int v_type,
627 vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2,
int nV3):
628 FifthSpeciesCounterpoint(home, nMes, cf, lb, ub, FIFTH_SPECIES, low, c, v_type, m_costs,
g_costs, s_costs, bm, nV3)
629 {
630     directCostArray = IntVarArray(home, fifthSpeciesMotions.size()-1, IntSet({0, directMoveCost
}));
631     varietyCostArray = IntVarArray(home, 3*(fifthSpeciesHIntervals.size()-2), IntSet({0,
varietyCost}));
632
633     //3.H6 : harmonic triad should be used on the second or third beat
634     thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
635     for(int i = 0; i < thirdHTriadArray.size(); i++){
636         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)&&
637         (fifthSpeciesHIntervals[(i*4)+2]!=UNISSON&&fifthSpeciesHIntervals[(i*4)+2]!=
MINOR_THIRD&&fifthSpeciesHIntervals[(i*4)+2]!=MAJOR_THIRD&&fifthSpeciesHIntervals[(i*4)

```

```

+2]!=PERFECT_FIFTH)) >>
638     (thirdHTriadArray[i]==triad3rdCost));
639     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)||
640     (fifthSpeciesHIntervals[(i*4)+2]==UNISSON||fifthSpeciesHIntervals[(i*4)+2]==
MINOR_THIRD||fifthSpeciesHIntervals[(i*4)+2]==MAJOR_THIRD||fifthSpeciesHIntervals[(i*4)
+2]==PERFECT_FIFTH)) >>
641     (thirdHTriadArray[i]==0));
642 }
643
644 //1.P1 3 voices version
645 for(int j = 0; j < fifthSpeciesMotions.size()-1; j++){
646     //set a cost when it is reached through direct motion, it is 0 when not
647     rel(home, (fifthSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||
firstSpeciesHarmonicIntervals[j+1]==7))>>
648     (directCostArray[j]==directMoveCost));
649     rel(home, (fifthSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&
firstSpeciesHarmonicIntervals[j+1]!=7))>>
650     (directCostArray[j]==0));
651 }
652
653 costs = IntVarArray(home, 11, 0, 1000000);
654 cost_names = {"fifth", "octave", "borrow", "melodic", "motion", "cambiata", "m2", "
syncopation", "direct", "variety", "triad3"};
655
656 //set cost[0] to be fifth cost
657 add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(1, 4/notesPerMeasure.at(
FIFTH_SPECIES), fifthCostArray.size()))), costs);
658 //set cost[1] to be octave cost
659 add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(1, 4/notesPerMeasure.at(
FIFTH_SPECIES), octaveCostArray.size()))), costs);
660 //set cost[2] to be off cost
661 add_cost(home, 2, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
FIFTH_SPECIES), offCostArray.size()))), costs);
662 //set cost[3] to be melodic cost
663 add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
FIFTH_SPECIES), melodicDegreeCost.size()))), costs);
664 //set cost[4] to be motion cost
665 add_cost(home, 4, fifthSpeciesMotionsCosts, costs);
666 //need to set cost[5] to be cambiata cost
667 add_cost(home, 5, cambiataCostArray, costs);
668 //need to set cost[6] to be m2Zero cost
669 add_cost(home, 6, m2ZeroCostArray, costs);
670 //need to set cost[7] to be syncopation cost
671 add_cost(home, 7, snycopeCostArray, costs);
672 //need to set cost[8] to be syncopation cost
673 add_cost(home, 8, directCostArray, costs);
674 //need to set cost[8] to be syncopation cost
675 add_cost(home, 9, varietyCostArray, costs);
676 //set cost[10] to be triad h third cost
677 add_cost(home, 10, thirdHTriadArray, costs);
678 }
679
680 /**
681  * This function returns a string with the characteristics of the counterpoint. It calls the
to_string() method from
682  * the Part class and adds 1st species specific characteristics.
683  * @return a string representation of the current instance of the FirstSpeciesCounterpoint
class.
684  */
685 string FifthSpeciesCounterpoint::to_string() const {
686     string text = Part::to_string() + "\n Fifth species :\n";
687     text += "Fifth species notes : " + intVarArray_to_string(fifthSpeciesNotesCp) + "\n";
688     text += "M Intervals array : " + intVarArray_to_string(fifthSpeciesSuccMIntervals) + "\n";
689     text += "isMostlyThird : " + boolVarArray_to_string(isMostlyThirdArray) + "\n";
690     text += "not Lowest array : " + boolVarArray_to_string(isNotLowest) + "\n";
691     text += "Species array : " + intVarArray_to_string(speciesArray) + "\n";
692     return text;
693 }
694
695 // clone constructor

```

```

696 FifthSpeciesCounterpoint::FifthSpeciesCounterpoint(Home home, FifthSpeciesCounterpoint &s) :
    Part(home, s){
697     solutionLength = s.solutionLength;
698     m2Len = s.m2Len;
699     fifthSpeciesNotesCp.update(home, s.fifthSpeciesNotesCp);
700     fifthSpeciesHIntervals.update(home, s.fifthSpeciesHIntervals);
701     firstHInterval.update(home, s.firstHInterval);
702     isNthSpeciesArray.update(home, s.isNthSpeciesArray);
703     isConstrainedArray.update(home, s.isConstrainedArray);
704     fifthSpeciesSuccMIntervals.update(home, s.fifthSpeciesSuccMIntervals);
705     fifthSpeciesMIntervals.update(home, s.fifthSpeciesMIntervals);
706     fifthSpeciesMTAIntervals.update(home, s.fifthSpeciesMTAIntervals);
707     fifthSpeciesM2Intervals.update(home, s.fifthSpeciesM2Intervals);
708     fifthSpeciesMAllIntervals.update(home, s.fifthSpeciesMAllIntervals);
709     fifthSpeciesMotions.update(home, s.fifthSpeciesMotions);
710     fifthSpeciesMotionsCosts.update(home, s.fifthSpeciesMotionsCosts);
711     isMostlyThirdArray.update(home, s.isMostlyThirdArray);
712     isFourthSpeciesArray.update(home, s.isFourthSpeciesArray);
713     isThirdSpeciesArray.update(home, s.isThirdSpeciesArray);
714     isNotCambiata.update(home, s.isNotCambiata);
715     m2ZeroCostArray.update(home, s.m2ZeroCostArray);
716     thirdHTriadArray.update(home, s.thirdHTriadArray);
717 }
718
719 FifthSpeciesCounterpoint* FifthSpeciesCounterpoint::clone(Home home){
720     return new FifthSpeciesCounterpoint(home, *this);
721 }
722
723 IntVarArray FifthSpeciesCounterpoint::getBranchingNotes(){
724     return fifthSpeciesNotesCp;
725 }
726
727 IntVarArray FifthSpeciesCounterpoint::getFirstHInterval(){
728     return firstSpeciesHarmonicIntervals;
729 }
730
731 IntVarArray FifthSpeciesCounterpoint::getMotions(){
732     return fifthSpeciesMotions;
733 }
734
735 IntVarArray FifthSpeciesCounterpoint::getFirstMInterval(){
736     return firstSpeciesMelodicIntervals;
737 }
738
739 int FifthSpeciesCounterpoint::getHIntervalSize(){
740     return fifthSpeciesHIntervals.size();
741 }
742
743 void FifthSpeciesCounterpoint::createSpeciesArrays(Home home){
744     IntVarArray countThird = IntVarArray(home, solutionLength, 0, 1);
745     IntVarArray countFourth = IntVarArray(home, solutionLength, 0, 1);
746     int nThirdInt = floor((prefSlider-1)*0.66);
747     int nFourthInt = floor(prefSlider*0.5);
748     IntVar sumThird = IntVar(home, nThirdInt, solutionLength);
749     IntVar sumFourth = IntVar(home, nFourthInt, solutionLength);
750
751     //Count 3rd and 4th species
752     for(int i = 0; i < solutionLength; i++){
753         BoolVar b = BoolVar(home, 0, 1);
754         rel(home, speciesArray[i], IRT_EQ, THIRD_SPECIES, Reify(b));
755         ite(home, b, IntVar(home, 1, 1), IntVar(home, 0, 0), countThird[i]);
756     }
757
758     for(int i = 0; i < solutionLength; i++){
759         BoolVar b = BoolVar(home, 0, 1);
760         rel(home, speciesArray[i], IRT_EQ, FOURTH_SPECIES, Reify(b));
761         ite(home, b, IntVar(home, 1, 1), IntVar(home, 0, 0), countFourth[i]);
762     }
763
764     //sum the counts
765     //rel(home, sumThird, IRT_EQ, expr(home, sum(countThird)));
766     //rel(home, sumFourth, IRT_EQ, expr(home, sum(countFourth)));
767

```

```

768 //no 4th species in second and fourth positions in the species array
769 for(int i = 1; i < solutionLength; i+=2){
770     rel(home, speciesArray[i], IRT_NQ, FOURTH_SPECIES);
771 }
772
773 //4th species in the third position is followed by no species (no note / constraint) and
774 //then a 4th species
775 for(int i = 0; i < solutionLength-1; i+=4){
776     BoolVar b34 = BoolVar(home, 0, 1);
777     BoolVar b14 = BoolVar(home, 0, 1);
778
779     rel(home, speciesArray[i+2], IRT_EQ, FOURTH_SPECIES, Reify(b34));
780     //the following line works to allow quarter notes after fourth species, but behaves
781     //weird
782     rel(home, expr(home, speciesArray[i+4]==FOURTH_SPECIES), BOT_OR, expr(home,
783     speciesArray[i+4]==THIRD_SPECIES), b14);
784     rel(home, speciesArray[i+4], IRT_EQ, FOURTH_SPECIES, Reify(b14));
785     rel(home, speciesArray[i+3], IRT_EQ, -1, Reify(b34, RM_IMP));
786     rel(home, b34, BOT_EQV, b14, 1);
787 }
788
789 //only 3rd and 4th species
790 for(int i = 0; i < speciesArray.size(); i++){
791     rel(home, speciesArray[i], IRT_NQ, FIRST_SPECIES);
792     rel(home, speciesArray[i], IRT_NQ, SECOND_SPECIES);
793 }
794
795 //first and penultimate measures are 4th species
796 rel(home, speciesArray[0], IRT_EQ, -1);
797 rel(home, speciesArray[1], IRT_EQ, -1);
798 rel(home, speciesArray[2], IRT_EQ, FOURTH_SPECIES);
799
800 rel(home, speciesArray[solutionLength-5], IRT_EQ, FOURTH_SPECIES);
801 rel(home, speciesArray[solutionLength-4], IRT_EQ, -1);
802 rel(home, speciesArray[solutionLength-3], IRT_EQ, FOURTH_SPECIES);
803
804 //no silence after third species notes
805 for(int i = 0; i < solutionLength-1; i++){
806     BoolVar b = BoolVar(home, 0, 1);
807     rel(home, speciesArray[i], IRT_EQ, THIRD_SPECIES, Reify(b));
808     rel(home, speciesArray[i+1], IRT_NQ, -1, Reify(b, RM_IMP));
809 }
810
811 //no silence after fourth species notes
812 for(int i = 0; i < solutionLength-5; i++){
813     BoolVar b = BoolVar(home, 0, 1);
814     rel(home, speciesArray[i], IRT_EQ, FOURTH_SPECIES, Reify(b));
815     rel(home, speciesArray[i+4], IRT_NQ, -1, Reify(b, RM_IMP));
816 }
817
818 //maximum two consecutive measures without a fourth species
819 for(int i = 2; i < solutionLength-13; i+=4){
820     BoolVar b1n4 = BoolVar(home, 0, 1);
821     BoolVar b2n4 = BoolVar(home, 0, 1);
822     BoolVar band = BoolVar(home, 0, 1);
823
824     rel(home, speciesArray[i+4], IRT_NQ, FOURTH_SPECIES, Reify(b1n4));
825     rel(home, speciesArray[i+8], IRT_NQ, FOURTH_SPECIES, Reify(b2n4));
826     rel(home, b1n4, BOT_AND, b2n4, band);
827     rel(home, speciesArray[i+12], IRT_EQ, FOURTH_SPECIES, Reify(band, RM_IMP));
828 }
829
830 /**
831  * CREATE NTH SPECIES ARRAY
832  */
833 cout << speciesArray.size() << endl;
834 cout << isNthSpeciesArray.size() << endl;
835 for(int i = 0; i < isNthSpeciesArray.size(); i+=5){
836     rel(home, speciesArray[floor(i/5)], IRT_EQ, -1, Reify(isNthSpeciesArray[i]));
837     rel(home, speciesArray[floor(i/5)], IRT_EQ, FIRST_SPECIES, Reify(isNthSpeciesArray[i
838     +1]));
839     rel(home, speciesArray[floor(i/5)], IRT_EQ, SECOND_SPECIES, Reify(isNthSpeciesArray[i
840     +2]));

```

```

836     rel(home, speciesArray[floor(i/5)], IRT_EQ, THIRD_SPECIES, Reify(isNthSpeciesArray[i
+3]));
837     rel(home, speciesArray[floor(i/5)], IRT_EQ, FOURTH_SPECIES, Reify(isNthSpeciesArray[i
+4]));
838 }
839
840 /**
841  * CREATE IS CONSTRAINED ARRAY
842  */
843 for(int i = 0; i < isConstrainedArray.size(); i++){
844     rel(home, isNthSpeciesArray[i*5], IRT_EQ, 0, Reify(isConstrainedArray[i]));
845 }
846 }

```

C.5.3 FirstSpeciesCounterpoint.cpp

```

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 #include "../headers/Parts/FirstSpeciesCounterpoint.hpp"
7
8 /**
9  * GENERAL CONSTRUCTOR
10 */
11
12 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
, int ub, Species mSpecies, Stratum* low, CantusFirmus* c,
13 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 for the first species notes
43     firstSpeciesHarmonicIntervals = IntVarArray(home, nMeasures* notesPerMeasure.at(
FIRST_SPECIES), -PERFECT_OCTAVE, PERFECT_OCTAVE);
44

```

```

45   for(int i = 0; i < firstSpeciesHarmonicIntervals.size(); i++){
46       rel(home, (firstSpeciesHarmonicIntervals[i]==((firstSpeciesNotesCp[i]-low->getNotes()
47       [i*4])%12)));
48   }
49   rel(home, firstSpeciesHarmonicIntervals, IRT_EQ, h_intervals.slice(0,4/notesPerMeasure.at(
50   FIRST_SPECIES),h_intervals.size()));
51
52   /// Melodic intervals for the first species notes
53   firstSpeciesMelodicIntervals = IntVarArray(home, nMeasures* notesPerMeasure.at(
54   FIRST_SPECIES) -1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
55   for(int i = 0; i < firstSpeciesMelodicIntervals.size(); i++)
56       rel(home, firstSpeciesMelodicIntervals[i], IRT_EQ, expr(home, firstSpeciesNotesCp[i+1] -
57       firstSpeciesNotesCp[i]));
58
59   //create the is off array which determines if a notes is borrowed or not
60   is_off = BoolVarArray(home, notes.size(), 0, 1);
61   initializeIsOffArray(home, this);
62
63   //create off_cost array
64   offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
65
66   //create the melodic degree cost array
67   melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({secondCost,
68   thirdCost, fourthCost, tritoneCost, fifthCost,
69   sixthCost, seventhCost, octaveCost}));
70
71   //create motions arrays
72   firstSpeciesMotions = IntVarArray(home, nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1,
73   IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION});
74   firstSpeciesMotionCosts = IntVarArray(home, firstSpeciesMotions.size(), IntSet{0,
75   directCost, obliqueCost, contraryCost});
76
77   //create motions
78
79   for(int i = 0; i < firstSpeciesMotions.size(); i++){
80
81       //direct motions help creation
82       BoolVar both_up = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low->
83       getMelodicIntervals()[i]>0)); //if both parts are going in the same direction
84       BoolVar both_stay = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(low->
85       getMelodicIntervals()[i]==0)); //if both parts are staying
86       BoolVar both_down = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low->
87       getMelodicIntervals()[i]<0)); //if both parts are going down
88       //oblique motions help creation
89       BoolVar cf_stays_1 = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low->
90       getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
91       BoolVar cf_stays_2 = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low->
92       getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
93       BoolVar cp_stays_1 = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(low->
94       getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
95       BoolVar cp_stays_2 = expr(home, (firstSpeciesMelodicIntervals[i]==0)&&(low->
96       getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
97       //contrary motions help creation
98       BoolVar cpd_cfu = expr(home, (firstSpeciesMelodicIntervals[i]<0)&&(low->
99       getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
100      BoolVar cpu_cfd = expr(home, (firstSpeciesMelodicIntervals[i]>0)&&(low->
101      getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
102
103      //direct constraints
104      rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
105      firstSpeciesMotions[i]==PARALLEL_MOTION));
106      rel(home, ((both_up || both_stay || both_down) && (this->isNotLowest[i]==1)) >> (
107      firstSpeciesMotionCosts[i]==directCost));
108      //oblique constraints
109      rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->
110      isNotLowest[i]==1)) >> (firstSpeciesMotions[i]==OBLIQUE_MOTION));
111      rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (this->
112      isNotLowest[i]==1)) >> (firstSpeciesMotionCosts[i]==obliqueCost));
113      //contrary constraints
114      rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (firstSpeciesMotions[
115      i]==CONTRARY_MOTION));

```

```

96     rel(home, ((cpd_cfu || cpu_cfd) && (this->isNotLowest[i]==1)) >> (
    firstSpeciesMotionCosts[i]==contraryCost));
97     //bass constraints
98     rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotions[i]==-1));
99     rel(home, (this->isNotLowest[i]==0) >> (firstSpeciesMotionCosts[i]==0));
100
101 }
102
103 // create pefectConsArray
104 fifthCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_fifthCost}));
105 octaveCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_octaveCost}));
106
107 isConsonance = BoolVarArray(home, h_intervals.size(), 0, 1);
108 //this loop checks that every harmonic interval is a consonance or not
109 for(int i = 0; i < isConsonance.size(); i++){
110     rel(home, expr(home, (h_intervals[i]==UNISSON)|| (h_intervals[i]==MINOR_THIRD)|| (
    h_intervals[i]==MAJOR_THIRD)|| (h_intervals[i]==PERFECT_FIFTH)||
111     (h_intervals[i]==MINOR_SIXTH)|| (h_intervals[i]==MAJOR_SIXTH)|| (h_intervals[i]==
    PERFECT_OCTAVE)), IRT_EQ, isConsonance[i]));
112 }
113 /// General rules
114
115 //G4 :
116 G4_counterpointMustBeInTheSameKey(home, this);
117
118 // G7 : melodic intervals should be small (works for 1st, 2nd and 3rd species)
119 G7_melodicIntervalsShouldBeSmall(home, this, motherSpecies);
120
121 /// Harmonic rules
122 /// H1 from Thibault: All harmonic intervals must be consonances
123 H1_1_harmonicIntervalsAreConsonances(home, this);
124
125 //H2 and H3 are found in the TwoVoiceCounterpoint class, since these rules are 2 voice
    specific
126
127 // H4 : applied as rule G9 in the Two, Three and FourVoice Counterpointproblem
128
129 // H6 from Thibault : Imperfect consonances are preferred
130 H6_1_preferImperfectConsonances(home, this);
131 }
132
133 /**
134  * 2 VOICES CONSTRUCTOR
135  */
136
137 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
    , int ub, Stratum* low, CantusFirmus* c, int v_type
138 , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV) :
139     FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c, v_type,
    m_costs, g_costs, s_costs, bm, nV) ///call the general constructor
140 {
141     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure
    .at(FIRST_SPECIES),m_intervals_brut.size()));
142
143     //H7,H8 from Thibault : penultimate note major sixth or minor third
144     H7_1_2v_penultimateSixthOrThird(home, this);
145
146     //1.M2 from Thibault: Melodic intervals cannot exceed a minor sixth
147     M1_1_2v_melodicIntervalsNotExceedMinorSixth(home, this);
148
149     // Motion rules
150     //1.P1 from Thibault : Perfect consonances cannot be reached by direct motion
151     P1_1_2v_noDirectMotionFromPerfectConsonance(home, this);
152
153     //P2 from Thibault : already done when creating motions array
154
155     //P3 from Thibault : no battuta
156     P3_1_noBattuta(home, this);
157
158     costs = IntVarArray(home, 5, 0, 1000000);
159     cost_names = {"fifth", "octave", "motion", "melodic", "borrow"};
160
161     //set cost[0] to be fifth cost

```

```

162     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
163         FIRST_SPECIES), fifthCostArray.size()))), costs);
164     //set cost[1] to be octave cost
165     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
166         FIRST_SPECIES), octaveCostArray.size()))), costs);
167     //set cost[2] to be motion cost
168     add_cost(home, 2, firstSpeciesMotionCosts, costs);
169     //set cost[3] to be melodic cost
170     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
171         FIRST_SPECIES), melodicDegreeCost.size()))), costs);
172     //set cost[4] to be off cost
173     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
174         FIRST_SPECIES), offCostArray.size()))), costs);
175 }
176
177 /**
178  * 3 VOICES CONSTRUCTOR
179  */
180
181 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
182     , int ub, Stratum* low, CantusFirmus* c, int v_type,
183     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2) :
184     FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c, v_type, m_costs,
185     g_costs, s_costs, bm, nV2) ///call the general constructor
186 {
187     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure
188         .at(FIRST_SPECIES),m_intervals_brut.size()));
189
190     varietyCostArray = IntVarArray(home, 3*(firstSpeciesHarmonicIntervals.size()-2), IntSet
191     ({0, varietyCost}));
192     directCostArray = IntVarArray(home, firstSpeciesMotions.size()-1,IntSet({0, directMoveCost
193     }));
194
195     //1.H7
196     H7_1_3v_penultimateSixthOrThird(home, this);
197
198     //1.M2 from Thibault: Melodic intervals cannot exceed a minor sixth (also include octave?)
199     M1_1_3v_melodicIntervalsNotExceedMinorSixth(home, this);
200
201     //1.P1
202     P1_1_3v_noDirectMotionFromPerfectConsonance(home, this);
203
204     //1.P3 from Thibault : no battuta
205     P3_1_noBattuta(home, this);
206
207     costs = IntVarArray(home, 7, 0, 1000000);
208     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct"};
209     //need to set cost[0] to be off cost
210     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
211         FIRST_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(
214         FIRST_SPECIES), fifthCostArray.size()))), costs);
215     //set cost[2] to be octave cost
216     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
217         FIRST_SPECIES), octaveCostArray.size()))), costs);
218     //need to set cost[3] to be variety cost
219     add_cost(home, 3, varietyCostArray, costs);
220     //set cost[4] to be motion cost
221     add_cost(home, 4, firstSpeciesMotionCosts, costs);
222     //set cost[5] to be melodic cost
223     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
224         FIRST_SPECIES), melodicDegreeCost.size()))), costs);
225     //need to set cost[6] to be direct cost
226     add_cost(home, 6, directCostArray, costs);
227 }
228
229 /**
230  * 4 VOICES CONSTRUCTOR
231  */
232
233 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int lb
234     , int ub, Stratum* low, CantusFirmus* c, int v_type,

```

```

221     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2,
222         int nV3):
223     FirstSpeciesCounterpoint(home, nMes, cf, lb, ub, FIRST_SPECIES, low, c, v_type, m_costs,
224         g_costs, s_costs, bm, nV3)
225 {
226     rel(home, firstSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure
227         .at(FIRST_SPECIES),m_intervals_brut.size()));
228
229     varietyCostArray = IntVarArray(home, 3*(firstSpeciesHarmonicIntervals.size()-2), IntSet
230         ({0, varietyCost}));
231     directCostArray = IntVarArray(home, firstSpeciesMotions.size()-1,IntSet({0, 2,
232         directMoveCost}));
233
234     /// M2 from Thibault: Melodic intervals cannot exceed a minor sixth (also include octave?)
235     M1_1_3v_melodicIntervalsNotExceedMinorSixth(home, this);
236
237     //P1 4 voices version
238     P1_1_4v_noDirectMotionFromPerfectConsonance(home, this);
239
240     //P3 from Thibault : no battuta
241     P3_1_noBattuta(home, this);
242
243     costs = IntVarArray(home, 7, 0, 1000000);
244     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct"};
245     //need to set cost[0] to be off cost
246     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
247         FIRST_SPECIES), offCostArray.size()))), costs);
248     //set cost[1] to be fifth cost
249     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
250         FIRST_SPECIES), fifthCostArray.size()))), costs);
251     //set cost[2] to be octave cost
252     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
253         FIRST_SPECIES), octaveCostArray.size()))), costs);
254     //need to set cost[3] to be variety cost
255     add_cost(home, 3, varietyCostArray, costs);
256     //set cost[4] to be motion cost
257     add_cost(home, 4, firstSpeciesMotionCosts, costs);
258     //set cost[5] to be melodic cost
259     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
260         FIRST_SPECIES), melodicDegreeCost.size()))), costs);
261     //need to set cost[6] to be direct cost
262     add_cost(home, 6, directCostArray, costs);
263 }
264
265 /**
266  * This function returns a string with the characteristics of the counterpoint. It calls the
267  * to_string() method from
268  * the Part class and adds 1st species specific characteristics.
269  * @return a string representation of the current instance of the FirstSpeciesCounterpoint
270  * class.
271 */
272 string FirstSpeciesCounterpoint::to_string() const {
273     string text = Part::to_string() + "\nFirst species :\n";
274     text += "First species first notes: " + intVarArray_to_string(firstSpeciesNotesCp) + "\n";
275     text += "First species isLowest array : " + boolVarArray_to_string(isNotLowest) + "\n";
276     text += "First species motions : " + intVarArray_to_string(firstSpeciesMotions) + "\n";
277     text += "First species h intervals : " + intVarArray_to_string(
278         firstSpeciesHarmonicIntervals) + "\n";
279     return text;
280 }
281
282 // clone constructor
283 FirstSpeciesCounterpoint::FirstSpeciesCounterpoint(Home home, FirstSpeciesCounterpoint &s) :
284     Part(home, s){
285     motherSpecies = s.motherSpecies;
286     //cantus = s.cantus;
287 }
288
289 FirstSpeciesCounterpoint* FirstSpeciesCounterpoint::clone(Home home){
290     return new FirstSpeciesCounterpoint(home, *this);
291 }
292
293 IntVarArray FirstSpeciesCounterpoint::getBranchingNotes(){

```

```

281     return firstSpeciesNotesCp;
282 }
283
284 IntVarArray FirstSpeciesCounterpoint::getFirstHInterval(){
285     return firstSpeciesHarmonicIntervals;
286 }
287
288 IntVarArray FirstSpeciesCounterpoint::getMotions(){
289     return firstSpeciesMotions;
290 }
291
292 IntVarArray FirstSpeciesCounterpoint::getFirstMInterval(){
293     return firstSpeciesMelodicIntervals;
294 }
295
296 int FirstSpeciesCounterpoint::getHIntervalSize(){
297     return firstSpeciesHarmonicIntervals.size();
298 }

```

C.5.4 FourthSpeciesCounterpoint.cpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #include "../headers/Parts/FourthSpeciesCounterpoint.hpp"
6
7 /**
8  * GENERAL CONSTRUCTOR
9  */
10 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(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
19     /*
20     if borrowMode is enabled, the extended domain is extended to make the inclusion of
    borrowed notes possible. We can see from Fux's examples
21     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
22     */
23     if(borrowMode==1){
24         extended_domain = vector_union(cp_range, vector_union(scale, borrowed_scale));
25     } else {
26         extended_domain = vector_intersection(cp_range, vector_union(scale, borrowed_scale));
27     }
28
29     off_domain = vector_difference(vector_intersection(cp_range, scale), lowerBound,
    upperBound);
30
31     fourthSpeciesNotesCp = IntVarArray(home, ((nMeasures*notesPerMeasure.at(FOURTH_SPECIES))
    -1)-1, IntSet(IntArgs(vector_intersection(cp_range, extended_domain))));
32     if(borrowMode==1){
33         fourthSpeciesNotesCp[fourthSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(
    vector_intersection(cp_range, chromatic_scale))));
34     }
35
36     sol = IntVarArray(home, fourthSpeciesNotesCp.slice(0,1,fourthSpeciesNotesCp.size()));
37
38     fourthSpeciesHIntervals = IntVarArray(home, ((nMeasures*notesPerMeasure.at(FOURTH_SPECIES)
    )-1)-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
39
40     for(int i = 0; i < fourthSpeciesHIntervals.size(); i++){
41         rel(home, (fourthSpeciesHIntervals[i])==(fourthSpeciesNotesCp[i]-low->getNotes()[
    floor((i+1)/2)*4])%12));

```

```

42     }
43
44
45     firstHInterval = IntVarArray(home, fourthSpeciesHIntervals.slice(1, 2,
46         fourthSpeciesHIntervals.size()));
47
48     //m-intervals : arsis and next thesis
49     //m-succ-intervals : thesis and arsis in same measure
50     //m2-intervals : between n and n+2 for whole counterpoint
51     //m-all-intervals : between all notes
52
53     fourthSpeciesMelodicIntervals = IntVarArray(home, fourthSpeciesNotesCp.size()-1, -
54         PERFECT_OCTAVE, PERFECT_OCTAVE);
55     for(int i = 0; i < fourthSpeciesMelodicIntervals.size(); i++){
56         rel(home, fourthSpeciesMelodicIntervals[i] == expr(home, fourthSpeciesNotesCp[i+1] -
57             fourthSpeciesNotesCp[i]));
58     }
59
60     m2IntervalsArray = IntVarArray(home, fourthSpeciesNotesCp.size()-2, -PERFECT_OCTAVE,
61         PERFECT_OCTAVE);
62     for(int i = 0; i < fourthSpeciesNotesCp.size()-2; i++){
63         rel(home, m2IntervalsArray[i], IRT_EQ, expr(home, fourthSpeciesNotesCp[i+2]-
64             fourthSpeciesNotesCp[i]));
65     }
66
67     is_off = BoolVarArray(home, notes.size(), 0, 1);
68     for(int i = 0; i < is_off.size(); i++){
69         IntVarArray res = IntVarArray(home, off_domain.size(), 0, 1);
70         IntVar sm = IntVar(home, 0, off_domain.size());
71         for(int l = 0; l < off_domain.size(); l++){
72             BoolVar b1 = BoolVar(home, 0, 1);
73             rel(home, notes[i], IRT_EQ, off_domain[l], Reify(b1));
74             ite(home, b1, IntVar(home, 1, 1), IntVar(home, 0, 0), res[l]);
75         }
76         IntVarArgs x(res.size());
77         for(int t = 0; t < off_domain.size(); t++){
78             x[t] = res[t];
79         }
80         rel(home, sm, IRT_EQ, expr(home, sum(x)));
81         rel(home, sm, IRT_GR, 0, Reify(is_off[i]));
82     }
83
84     //create off_cost array
85     offCostArray = IntVarArray(home, is_off.size(), IntSet({0, borrowCost}));
86     //set the cost for borrowing this note
87     for(int i = 0; i < is_off.size(); i++){
88         rel(home, (is_off[i]==0) >> (offCostArray[i]==0));
89         rel(home, (is_off[i]==1) >> (offCostArray[i]==borrowCost));
90     }
91
92     melodicDegreeCost = IntVarArray(home, m_intervals_brut.size(), IntSet({secondCost,
93         thirdCost, fourthCost, tritoneCost, fifthCost,
94         sixthCost, seventhCost, octaveCost}));
95
96     // create pefectConsArray
97     fifthCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_fifthCost}));
98     octaveCostArray = IntVarArray(home, h_intervals.size(), IntSet({0, h_octaveCost}));
99
100     // H6 from Thibault : Imperfect consonances are preferred
101     for(int i = 0; i < h_intervals.size(); i++){
102         rel(home, octaveCostArray[i], IRT_EQ, h_octaveCost, Reify(expr(home, h_intervals[i]==
103             UNISSON), RM_PMI));
104         rel(home, octaveCostArray[i], IRT_EQ, 0, Reify(expr(home, h_intervals[i]!=UNISSON),
105             RM_PMI));
106
107         rel(home, fifthCostArray[i], IRT_EQ, h_fifthCost, Reify(expr(home, h_intervals[i]==
108             PERFECT_FIFTH), RM_PMI));
109         rel(home, fifthCostArray[i], IRT_EQ, 0, Reify(expr(home, h_intervals[i]!=PERFECT_FIFTH
110             ), RM_PMI));
111     }
112 }

```

```

105 isConsonance = BoolVarArray(home, h_intervals.size(), 0, 1);
106 for(int i = 0; i < isConsonance.size(); i++){
107     rel(home, expr(home, (abs(h_intervals[i])==UNISSON)||abs(h_intervals[i])==MINOR_THIRD
108     )||(abs(h_intervals[i])==MAJOR_THIRD)||abs(h_intervals[i])==PERFECT_FIFTH)||
109     (abs(h_intervals[i])==MINOR_SIXTH)||abs(h_intervals[i])==MAJOR_SIXTH)||abs(
110     h_intervals[i])==PERFECT_OCTAVE)), IRT_EQ, isConsonance[i]);
111 }
112
113 isNoSyncopeArray = BoolVarArray(home, nMeasures-1, 0, 1);
114 for(int i = 0; i < isNoSyncopeArray.size(); i++){
115     rel(home, fourthSpeciesMelodicIntervals[(i*2)], IRT_NQ, 0, Reify(isNoSyncopeArray[i]))
116     ;
117 }
118
119 // G6 : no chromatic melodies
120
121 for(int i = 0; i < m_intervals_brut.size()-1; i+=4/notesPerMeasure.at(FOURTH_SPECIES)){
122     rel(home, expr(home, m_intervals_brut[i]==1), BOT_AND, expr(home, m_intervals_brut[i
123     +1]==1), 0);
124     rel(home, expr(home, m_intervals_brut[i]==-1), BOT_AND, expr(home, m_intervals_brut[i
125     +1]==-1), 0);
126 }
127
128 // G7 : melodic intervals should be small (works for 1st, 2nd and 3rd species)
129 int idx = 0;
130 for(int i = 0; i < m_intervals_brut.size(); i+=4/notesPerMeasure.at(FOURTH_SPECIES)){
131     rel(home, (abs(m_intervals_brut[i])<MINOR_THIRD) >> (melodicDegreeCost[i]==secondCost)
132     );
133     rel(home, (abs(m_intervals_brut[i])==MINOR_THIRD || abs(m_intervals_brut[i])==
134     MAJOR_THIRD) >> (melodicDegreeCost[i]==thirdCost));
135     rel(home, (abs(m_intervals_brut[i])==PERFECT_FOURTH) >> (melodicDegreeCost[i]==
136     fourthCost));
137     rel(home, (abs(m_intervals_brut[i])==TRITONE) >> (melodicDegreeCost[i]==tritoneCost));
138     rel(home, (abs(m_intervals_brut[i])==PERFECT_FIFTH) >> (melodicDegreeCost[i]==
139     fifthCost));
140     rel(home, (abs(m_intervals_brut[i])==MINOR_SIXTH || abs(m_intervals_brut[i])==
141     MAJOR_SIXTH) >> (melodicDegreeCost[i]==sixthCost));
142     rel(home, (abs(m_intervals_brut[i])==MINOR_SEVENTH || abs(m_intervals_brut[i])==
143     MAJOR_SEVENTH) >> (melodicDegreeCost[i]==seventhCost));
144     rel(home, (abs(m_intervals_brut[i])==PERFECT_OCTAVE) >> (melodicDegreeCost[i]==
145     octaveCost));
146     idx++;
147 }
148
149 m2ZeroArray = IntVarArray(home, ((fourthSpeciesNotesCp.size()-1)/2)-2, IntSet({0,
150 m2ZeroCost}));
151 snycopeCostArray = IntVarArray(home, (fourthSpeciesMelodicIntervals.size())/2, IntSet({0,
152 syncopationCost}));
153
154 //link them
155 rel(home, fourthSpeciesNotesCp, IRT_EQ, notes.slice(2, 4/notesPerMeasure.at(FOURTH_SPECIES
156 ), notes.size()));
157 rel(home, fourthSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(2,4/
158 notesPerMeasure.at(FOURTH_SPECIES),m_intervals_brut.size()));
159 rel(home, fourthSpeciesHIntervals, IRT_EQ, h_intervals.slice(2, 4/notesPerMeasure.at(
160 FOURTH_SPECIES), h_intervals.size()));
161
162 //4.H1 : arsis harmonies must be consonant (could be the reason for 3 / 4 voice bugs)
163 for(int i = 0; i < fourthSpeciesHIntervals.size(); i+=2){
164     dom(home, expr(home, abs(fourthSpeciesHIntervals[i])), IntSet(CONSONANCES));
165 }
166
167 //Melodic intervals cannot be greater than Minor Sixth (except octave)
168 dom(home, fourthSpeciesMelodicIntervals, IntSet({UNISSON, -PERFECT_OCTAVE, -MINOR_SIXTH, -
169 PERFECT_FIFTH, -AUGMENTED_FOURTH, -PERFECT_FOURTH,
170 -MAJOR_THIRD, -MINOR_THIRD, -MAJOR_SECOND, -MINOR_SECOND, PERFECT_OCTAVE, MINOR_SIXTH,
171 PERFECT_FIFTH, AUGMENTED_FOURTH, PERFECT_FOURTH,
172 MAJOR_THIRD, MINOR_THIRD, MAJOR_SECOND, MINOR_SECOND}));
173
174 //no chromatic motion between 3 consecutive notes
175 for(int i = 1; i < fourthSpeciesMelodicIntervals.size()-1; i++){
176     BoolVar b1 = BoolVar(home, 0, 1);
177     BoolVar b2 = BoolVar(home, 0, 1);

```

```

159     BoolVar b3 = BoolVar(home, 0, 1);
160     BoolVar b4 = BoolVar(home, 0, 1);
161
162     rel(home, fourthSpeciesMelodicIntervals[i+1], IRT_EQ, 1, Reify(b1));
163     rel(home, m2IntervalsArray[i], IRT_EQ, 2, Reify(b2));
164     rel(home, b1, BOT_AND, b2, 0);
165
166     rel(home, fourthSpeciesMelodicIntervals[i+1], IRT_EQ, -1, Reify(b3));
167     rel(home, m2IntervalsArray[i], IRT_EQ, -2, Reify(b4));
168     rel(home, b3, BOT_AND, b4, 0);
169 }
170
171 //4.M1 Arsis half notes should be the same as their next halves in thesis
172 for(int i = 0; i < snycopeCostArray.size(); i++){
173     rel(home, (isNoSyncopeArray[i]==1) >> (snycopeCostArray[i]==syncopationCost));
174     rel(home, (isNoSyncopeArray[i]==0) >> (snycopeCostArray[i]==0));
175 }
176
177 //4.M2 notes and two beats further are preferred to be different
178 for(int i = 0; i < m2ZeroArray.size(); i++){
179     rel(home, (fourthSpeciesNotesCp[(i*2)]==fourthSpeciesNotesCp[(i*2)+4]) >> (m2ZeroArray
180 [i]==m2ZeroCost));
181     rel(home, (fourthSpeciesNotesCp[(i*2)]!=fourthSpeciesNotesCp[(i*2)+4]) >> (m2ZeroArray
182 [i]==0));
183 }
184
185 //4.P1
186 for(int i = 1; i < nMeasures-1; i++){
187     //rel(home, (isConsonance[(i*4)]==0) >> ((fourthSpeciesMelodicIntervals[(i*2)+1]==-
188 MINOR_SECOND) ||
189 // (fourthSpeciesMelodicIntervals[(i*2)+1]==-MAJOR_SECOND)));
190 }
191
192 //4.P2
193 for(int i = 0; i < nMeasures-1; i++){
194     BoolVar buni = BoolVar(home, 0, 1);
195     BoolVar band = BoolVar(home, 0, 1);
196     rel(home, fourthSpeciesHIntervals[(i*2)], IRT_EQ, UNISSON, buni);
197     rel(home, expr(home, !c->getIsNotLowest()[i]), BOT_AND, buni, band);
198     rel(home, fourthSpeciesHIntervals[i*2], IRT_NQ, 1, Reify(band, RM_IMP));
199     rel(home, fourthSpeciesHIntervals[i*2], IRT_NQ, 2, Reify(band, RM_IMP));
200 }
201 }
202
203 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int
204 lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
205 vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
206 FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c, v_type, m_costs,
207 g_costs, s_costs, bm, nV)
208 {
209
210 //4.H2 : If the 4th species is the lowest stratum, then no hamonic seventh
211 for(int j = 1; j < getIsNotLowest().size(); j++){
212     rel(home, (getIsNotLowest()[j]==0) >> (fourthSpeciesHIntervals[(j*2)-1]!=MINOR_SEVENTH
213 && fourthSpeciesHIntervals[(j*2)-1]!=-MINOR_SEVENTH));
214     rel(home, (getIsNotLowest()[j]==0) >> (fourthSpeciesHIntervals[(j*2)-1]!=MAJOR_SEVENTH
215 && fourthSpeciesHIntervals[(j*2)-1]!=-MAJOR_SEVENTH));
216 }
217
218 //4.H3 Penult note condition
219 rel(home, (isNotLowest[isNotLowest.size()-2]==1) >> (expr(home, abs(fourthSpeciesHIntervals
220 [fourthSpeciesHIntervals.size()-3]))==MINOR_SEVENTH ||
221     expr(home, abs(fourthSpeciesHIntervals[fourthSpeciesHIntervals.size()-3]))==
222 MAJOR_SEVENTH));
223
224 //Must start with a perfect consonance (since 1.H2 applies to the first note, which is
225 most likely a rest in 4th species)
226 dom(home, fourthSpeciesHIntervals[0], IntSet({UNISSON, PERFECT_FIFTH, -PERFECT_FIFTH}));
227
228 costs = IntVarArray(home, 6, 0, 1000000);
229 cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation"};
230
231

```

```

222 //set cost[0] to be fifth cost
223 add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure.at(
    FOURTH_SPECIES), fifthCostArray.size()))), costs);
224 //set cost[1] to be octave cost
225 add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/notesPerMeasure.at(
    FOURTH_SPECIES), octaveCostArray.size()))), costs);
226 //set cost[2] to be melodic cost
227 add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/notesPerMeasure.at(
    FOURTH_SPECIES), melodicDegreeCost.size()))), costs);
228 //set cost[3] to be off cost
229 add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.at(
    FOURTH_SPECIES), offCostArray.size()))), costs);
230 //need to set cost[4] to be m2Zero cost
231 add_cost(home, 4, m2ZeroArray, costs);
232 //need to set cost[5] to be syncopation cost
233 add_cost(home, 5, snycopeCostArray, costs);
234 }
235
236 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int
    lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
237 vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2):
238 FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV2)
239 {
240     varietyCostArray = IntVarArray(home, 3*(getHIntervalSize()-2), IntSet({0, varietyCost}));
241
242     //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.
243     // for(int j = 1; j < nMeasures-1; j++){
244         //rel(home, (low->getMelodicIntervals()[j]==0)>>(expr(home, abs(
    fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SECOND || expr(home, abs(fourthSpeciesHIntervals
    [(j*2)+1]))==MAJOR_SECOND
245         // || expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==PERFECT_FOURTH || expr(
    home, abs(fourthSpeciesHIntervals[(j*2)+1]))==AUGMENTED_FOURTH ||
246         // expr(home, abs(fourthSpeciesHIntervals[(j*2)+1]))==MINOR_SEVENTH || expr(home,
    abs(fourthSpeciesHIntervals[(j*2)+1]))==MAJOR_SEVENTH));
247     // }
248
249     costs = IntVarArray(home, 7, 0, 1000000);
250     cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation", "variety"};
251
252     //set cost[0] to be fifth cost
253     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), fifthCostArray.size()))), costs);
254     //set cost[1] to be octave cost
255     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), octaveCostArray.size()))), costs);
256     //set cost[2] to be melodic cost
257     add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), melodicDegreeCost.size()))), costs);
258     //set cost[3] to be off cost
259     add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.at(
        FOURTH_SPECIES), offCostArray.size()))), costs);
260     //need to set cost[4] to be m2Zero cost
261     add_cost(home, 4, m2ZeroArray, costs);
262     //need to set cost[5] to be syncopation cost
263     add_cost(home, 5, snycopeCostArray, costs);
264     //need to set cost[6] to be variety cost
265     add_cost(home, 6, varietyCostArray, costs);
266 }
267
268 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, int nMes, vector<int> cf, int
    lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
270 vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2,
    int nV3):
271 FourthSpeciesCounterpoint(home, nMes, cf, lb, ub, FOURTH_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV2)
272 {
273     varietyCostArray = IntVarArray(home, 3*(getHIntervalSize()-2), IntSet({0, varietyCost}));
274
275     //4.P5

```

```

276     for(int j = 0; j < nMeasures-1; j++){
277         rel(home, (low->getMelodicIntervals()[j]==0)==(isConsonance[(j*4)+2]==0));
278         rel(home, (low->getMelodicIntervals()[j]!=0)==(isConsonance[(j*4)+2]==1));
279     }
280
281     costs = IntVarArray(home, 7, 0, 1000000);
282     cost_names = {"fifth", "octave", "melodic", "borrow", "m2", "syncopation", "variety"};
283
284     //set cost[0] to be fifth cost
285     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(2, 4/notesPerMeasure.at(
286         FOURTH_SPECIES), fifthCostArray.size()))), costs);
287     //set cost[1] to be octave cost
288     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(2, 4/notesPerMeasure.at(
289         FOURTH_SPECIES), octaveCostArray.size()))), costs);
290     //set cost[2] to be melodic cost
291     add_cost(home, 2, IntVarArray(home, melodicDegreeCost.slice(2, 4/notesPerMeasure.at(
292         FOURTH_SPECIES), melodicDegreeCost.size()))), costs);
293     //set cost[3] to be off cost
294     add_cost(home, 3, IntVarArray(home, offCostArray.slice(2, 4/notesPerMeasure.at(
295         FOURTH_SPECIES), offCostArray.size()))), costs);
296     //need to set cost[4] to be m2Zero cost
297     add_cost(home, 4, m2ZeroArray, costs);
298     //need to set cost[5] to be syncopation cost
299     add_cost(home, 5, snycopeCostArray, costs);
300     //need to set cost[6] to be variety cost
301     add_cost(home, 6, varietyCostArray, costs);
302 }
303
304 /**
305  * This function returns a string with the characteristics of the counterpoint. It calls the
306  * to_string() method from
307  * the Part class and adds 1st species specific characteristics.
308  * @return a string representation of the current instance of the FirstSpeciesCounterpoint
309  * class.
310  */
311 string FourthSpeciesCounterpoint::to_string() const {
312     string text = Part::to_string() + "\nFourth species :\n";
313     text += "No Syncope array : " + boolVarArray_to_string(isNoSyncopeArray) + "\n";
314     text += "Fourth species notes : " + intVarArray_to_string(fourthSpeciesNotesCp) + "\n";
315     text += "Fourth species h intervals : " + intVarArray_to_string(fourthSpeciesHIntervals) +
316         "\n";
317     text += "Fourth species m intervals : " + intVarArray_to_string(
318         fourthSpeciesMelodicIntervals) + "\n";
319     text += "is Consonance array : " + boolVarArray_to_string(isConsonance) + "\n";
320     return text;
321 }
322
323 // clone constructor
324 FourthSpeciesCounterpoint::FourthSpeciesCounterpoint(Home home, FourthSpeciesCounterpoint &s)
325 : Part(home, s){
326     fourthSpeciesNotesCp.update(home, s.fourthSpeciesNotesCp);
327     fourthSpeciesHIntervals.update(home, s.fourthSpeciesHIntervals);
328     fourthSpeciesMelodicIntervals.update(home, s.fourthSpeciesMelodicIntervals);
329     m2IntervalsArray.update(home, s.m2IntervalsArray);
330     firstHInterval.update(home, s.firstHInterval);
331     m2ZeroArray.update(home, s.m2ZeroArray);
332     sol.update(home, s.sol);
333 }
334
335 FourthSpeciesCounterpoint* FourthSpeciesCounterpoint::clone(Home home){
336     return new FourthSpeciesCounterpoint(home, *this);
337 }
338
339 IntVarArray FourthSpeciesCounterpoint::getBranchingNotes(){
340     return sol;
341 }
342
343 IntVarArray FourthSpeciesCounterpoint::getFirstHInterval(){
344     return firstHInterval;
345 }
346
347 IntVarArray FourthSpeciesCounterpoint::getMotions(){
348     return motions;
349 }

```

```

340 }
341
342 IntVarArray FourthSpeciesCounterpoint::getFirstMInterval(){
343     return m2IntervalsArray;
344 }
345
346 int FourthSpeciesCounterpoint::getHIntervalSize(){
347     return fourthSpeciesHIntervals.size();
348 }

```

C.5.5 Part.cpp

```

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  #include "../headers/Parts/Part.hpp"
7
8  /// This class represents a part, so it creates all the variables associated to that part and
9  /// posts the constraints that are species independent
10 Part::Part(Home home, int nMes, Species sp, vector<int> cf, int lb, int ub, int v_type, vector
11 <int> m_costs, vector<int> g_costs, vector<int> s_costs,
12 int nV, int bm) :
13     Voice(home, nMes, lb, ub){
14         nVoices      = nV;
15         species      = sp;
16         borrowMode   = bm;
17         voice_type   = v_type;
18         //lowest      = low;
19         isNotLowest   = BoolVarArray(home, nMeasures, 0, 1);
20         isHighest     = BoolVarArray(home, nMeasures, 0, 1);
21
22         borrowed_scale = get_all_notes_from_scale(cf[0]%12, BORROWED_SCALE);
23         scale = get_all_notes_from_scale(cf[0]%12, MAJOR_SCALE);
24         chromatic_scale = get_all_notes_from_scale(cf[0]%12, CHROMATIC_SCALE);
25         cp_range = {};
26
27         secondCost = m_costs[0];
28         thirdCost = m_costs[1];
29         fourthCost = m_costs[2];
30         tritoneCost = m_costs[3];
31         fifthCost = m_costs[4];
32         sixthCost = m_costs[5];
33         seventhCost = m_costs[6];
34         octaveCost = m_costs[7];
35
36         borrowCost = g_costs[0];
37         h_fifthCost = g_costs[1];
38         h_octaveCost = g_costs[2];
39         succCost = g_costs[3];
40         varietyCost = g_costs[4];
41         triadCost = g_costs[5];
42         directMoveCost = g_costs[6];
43         penultCost = g_costs[7];
44
45         penultSixthCost = s_costs[0];
46         cambiataCost = s_costs[1];
47         mSkipCost = s_costs[2];
48         triad3rdCost = s_costs[3];
49         m2ZeroCost = s_costs[4];
50         syncopationCost = s_costs[5];
51         prefSlider = s_costs[6];
52
53         directCost = 0;
54         obliqueCost = 1;
55         contraryCost = 2;
56
57 }
58
59 string Part::to_string() const{

```

```

58     string part = "Part characteristics :\n";
59     part += "Part costs : \n";
60     part += intVarArray_to_string(costs);
61     part += "\n";
62     return part;
63 }
64
65
66
67
68 Part::Part(Home home, Part& s) : Voice(home, s) {
69     nVoices = s.nVoices;
70     borrowMode = s.borrowMode;
71
72     borrowed_scale = s.borrowed_scale;
73     scale = s.scale;
74     chromatic_scale = s.chromatic_scale;
75     species = s.species;
76     voice_type = s.voice_type;
77
78     cp_range = s.cp_range;
79
80     extended_domain = s.extended_domain;
81     off_domain = s.off_domain;
82
83     secondCost = s.secondCost;
84     thirdCost = s.thirdCost;
85     fourthCost = s.fourthCost;
86     tritoneCost = s.tritoneCost;
87     fifthCost = s.fifthCost;
88     sixthCost = s.sixthCost;
89     seventhCost = s.seventhCost;
90     octaveCost = s.octaveCost;
91
92     borrowCost = s.borrowCost;
93     h_fifthCost = s.h_fifthCost;
94     h_octaveCost = s.h_octaveCost;
95     succCost = s.succCost;
96     varietyCost = s.varietyCost;
97     triadCost = s.triadCost;
98     directMoveCost = s.directMoveCost;
99     penultCost = s.penultCost;
100
101     penultSixthCost = s.penultSixthCost;
102     cambiataCost = s.cambiataCost;
103     mSkipCost = s.mSkipCost;
104     triad3rdCost = s.triad3rdCost;
105     m2ZeroCost = s.m2ZeroCost;
106     syncopationCost = s.syncopationCost;
107     prefSlider = s.prefSlider;
108
109     directCost = s.directCost;
110     obliqueCost = s.obliqueCost;
111     contraryCost = s.contraryCost;
112
113     cost_names = s.cost_names;
114
115     melodicDegreeCost.update(home, s.melodicDegreeCost);
116     fifthCostArray.update(home, s.fifthCostArray);
117     octaveCostArray.update(home, s.octaveCostArray);
118     is_off.update(home, s.is_off);
119     offCostArray.update(home, s.offCostArray);
120     costs.update(home, s.costs);
121     varietyCostArray.update(home, s.varietyCostArray);
122     directCostArray.update(home, s.directCostArray);
123     isConsonance.update(home, s.isConsonance);
124     isNotLowest.update(home, s.isNotLowest);
125     isHighest.update(home, s.isHighest);
126
127     firstSpeciesHarmonicIntervals.update(home, s.firstSpeciesHarmonicIntervals);
128     firstSpeciesNotesCp.update(home, s.firstSpeciesNotesCp);
129     firstSpeciesMelodicIntervals.update(home, s.firstSpeciesMelodicIntervals);
130     firstSpeciesMotions.update(home, s.firstSpeciesMotions);

```

```

131     firstSpeciesMotionCosts.update(home, s.firstSpeciesMotionCosts);
132
133     isDiminution.update(home, s.isDiminution);
134     penultCostArray.update(home, s.penultCostArray);
135     secondSpeciesMotions.update(home, s.secondSpeciesMotions);
136     secondSpeciesMelodicIntervals.update(home, s.secondSpeciesMelodicIntervals);
137     secondSpeciesRealMotions.update(home, s.secondSpeciesRealMotions);
138
139     is5QNArray.update(home, s.is5QNArray);
140     thirdSpeciesHarmonicIntervals.update(home, s.thirdSpeciesHarmonicIntervals);
141     thirdSpeciesMelodicIntervals.update(home, s.thirdSpeciesMelodicIntervals);
142     cambiataCostArray.update(home, s.cambiataCostArray);
143
144     isNoSyncopeArray.update(home, s.isNoSyncopeArray);
145     snycopeCostArray.update(home, s.snycopeCostArray);
146
147     speciesArray.update(home, s.speciesArray);
148
149 }
150
151 // Virtual clone function
152 Part* Part::clone(Home home) {
153     return new Part(home, *this);
154 }
155
156 IntVarArray Part::getBranchingNotes(){
157     return notes;
158 }
159
160 IntVarArray Part::getPartNotes(){
161     return notes;
162 }
163
164 int Part::getSuccCost(){
165     return succCost;
166 }
167
168 IntVarArray Part::getFirstHInterval(){
169     return h_intervals;
170 }
171
172 IntVarArray Part::getMotions(){
173     return motions;
174 }
175
176 IntVarArray Part::getFirstMInterval(){
177     return m_intervals_brut;
178 }
179
180 IntVarArray Part::getCosts(){
181     return costs;
182 }
183
184 int Part::getVarietyCost(){
185     return varietyCost;
186 }
187
188 IntVar Part::getVarietyArray(int idx){
189     return varietyCostArray[idx];
190 }
191
192 int Part::getHIntervalSize(){
193     return h_intervals.size();
194 }
195
196 int Part::getTriadCost(){
197     return triadCost;
198 }
199
200 int Part::getSecondCost(){
201     return secondCost;
202 }
203 int Part::getThirdCost(){

```

```

204     return thirdCost;
205 }
206 int Part::getFourthCost(){
207     return fourthCost;
208 }
209 int Part::getTritoneCost(){
210     return tritoneCost;
211 }
212 int Part::getFifthCost(){
213     return fifthCost;
214 }
215 int Part::getSixthCost(){
216     return sixthCost;
217 }
218 int Part::getSeventhCost(){
219     return seventhCost;
220 }
221 int Part::getOctaveCost(){
222     return octaveCost;
223 }
224
225 IntVarArray Part::getMelodicDegreeCost(){
226     return melodicDegreeCost;
227 }
228
229 vector<string> Part::getCostNames(){
230     return cost_names;
231 }
232
233 BoolVarArray Part::getConsonance(){
234     return isConsonance;
235 }
236
237 void Part::add_cost(Home home, int idx, IntVarArray to_be_added, IntVarArray costs){
238     int sz = to_be_added.size();
239     IntVarArgs args(sz);
240     for(int i = 0; i < sz; i++){
241         args[i] = to_be_added[i];
242     }
243     rel(home, costs[idx], IRT_EQ, expr(home, sum(args)));
244 }
245
246 BoolVarArray Part::getIsNotLowest(){
247     return isNotLowest;
248 }
249
250 IntVarArray Part::getFirstSpeciesHIntervals(){
251     return firstSpeciesHarmonicIntervals;
252 }
253
254 IntVarArray Part::getFirstSpeciesNotes(){
255     return firstSpeciesNotesCp;
256 }
257
258 IntVarArray Part::getFirstSpeciesMIntervals(){
259     return firstSpeciesMelodicIntervals;
260 }
261
262 IntVarArray Part::getFifthCostArray(){
263     return fifthCostArray;
264 }
265
266 IntVarArray Part::getOctaveCostArray(){
267     return octaveCostArray;
268 }
269
270 int Part::getHFifthCost(){
271     return h_fifthCost;
272 }
273
274 int Part::getHOctaveCost(){
275     return h_octaveCost;
276 }

```

```

277
278 IntVarArray Part::getFirstSpeciesMotions(){
279     return firstSpeciesMotions;
280 }
281
282 IntVarArray Part::getDirectCostArray(){
283     return directCostArray;
284 }
285
286 int Part::getDirectCost(){
287     return directCost;
288 }
289
290 BoolVarArray Part::getIsOffArray(){
291     return is_off;
292 }
293
294 vector<int> Part::getOffDomain(){
295     return off_domain;
296 }
297
298 vector<int> Part::getExtendedDomain(){
299     return extended_domain;
300 }
301
302 IntVarArray Part::getOffCostArray(){
303     return offCostArray;
304 }
305
306 int Part::getBorrowCost(){
307     return borrowCost;
308 }
309
310 BoolVarArray Part::getIsDiminution(){
311     return isDiminution;
312 }
313
314 int Part::getPenultCost(){
315     return penultCost;
316 }
317
318 IntVarArray Part::getPenultCostArray(){
319     return penultCostArray;
320 }
321
322 IntVarArray Part::getSecondSpeciesMotions(){
323     return secondSpeciesMotions;
324 }
325
326 IntVarArray Part::getSecondSpeciesMIntervals(){
327     return secondSpeciesMelodicIntervals;
328 }
329
330 IntVarArray Part::getSecondSpeciesRealMotions(){
331     return secondSpeciesRealMotions;
332 }
333
334 int Part::getDirectMoveCost(){
335     return directMoveCost;
336 }
337
338 BoolVarArray Part::getIs5QNArray(){
339     return is5QNArray;
340 }
341
342 IntVarArray Part::getThirdSpeciesHIntervals(){
343     return thirdSpeciesHarmonicIntervals;
344 }
345
346 IntVarArray Part::getThirdSpeciesMIntervals(){
347     return thirdSpeciesMelodicIntervals;
348 }
349

```

```

350 int Part::getCambiataCost(){
351     return cambiataCost;
352 }
353
354 IntVarArray Part::getCambiataCostArray(){
355     return cambiataCostArray;
356 }
357
358 BoolVarArray Part::getNoSyncope(){
359     return isNoSyncopeArray;
360 }
361
362 IntVarArray Part::getSyncopeCostArray(){
363     return snycopeCostArray;
364 }
365
366 IntVarArray Part::getSpeciesArray(){
367     return speciesArray;
368 }
369
370 BoolVarArray Part::getIsHighest(){
371     return isHighest;
372 }

```

C.5.6 SecondSpeciesCounterpoint.cpp

```

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 #include "../headers/Parts/SecondSpeciesCounterpoint.hpp"
7
8 /**
9  * Second species constructor. Does general constructing. Then calls the general first species
10   constructor
11 */
12 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf, int
13   lb, int ub, int mSpec, Stratum* low, CantusFirmus* c, int v_type
14   , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
15   FirstSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs,
16   g_costs, s_costs, bm, nV) /// super constructor. Applies all rules for the first species
17   to the 1st note of each measure
18 {
19     /// Second species notes in the counterpoint
20     secondSpeciesNotesCp = IntVarArray(home, (nMeasures*notesPerMeasure.at(SECOND_SPECIES))-1,
21     IntSet(IntArgs(vector_intersection(cp_range, extended_domain))));
22     if(borrowMode==1){
23         secondSpeciesNotesCp[secondSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(
24         vector_intersection(cp_range, chromatic_scale))));
25     }
26     rel(home, secondSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(SECOND_SPECIES)
27     ,(notes.size())));
28     /// Harmonic intervals for the second species notes
29     secondSpeciesHarmonicIntervals = IntVarArray(home, (nMeasures*notesPerMeasure.at(
30     SECOND_SPECIES))-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
31     rel(home, secondSpeciesHarmonicIntervals, IRT_EQ, h_intervals.slice(0,4/notesPerMeasure.at
32     (SECOND_SPECIES),(h_intervals.size())));
33     for(int i = 0; i < secondSpeciesHarmonicIntervals.size(); i++){
34         rel(home, (secondSpeciesHarmonicIntervals[i])=((secondSpeciesNotesCp[i]-low->getNotes
35         ())[(floor(i/2)*4)%12]));
36     }
37
38     /// Melodic intervals for the second species notes
39     secondSpeciesMelodicIntervals = IntVarArray(home, secondSpeciesNotesCp.size()-1, -
40     PERFECT_OCTAVE, PERFECT_OCTAVE);
41
42     /// link melodic intervals
43     for(int i = 0; i < secondSpeciesMelodicIntervals.size(); i++)
44         rel(home, secondSpeciesMelodicIntervals[i], IRT_EQ, expr(home, secondSpeciesNotesCp[i
45         +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(), -
    PERFECT_OCTAVE, PERFECT_OCTAVE);
37
38 for(int i = 0; i < secondSpeciesArsisArray.size()-1; i++){
39     rel(home, secondSpeciesArsisArray[i], IRT_EQ, expr(home, secondSpeciesNotesCp[(i*2)
40 +3]-secondSpeciesNotesCp[(i*2)+1]));
41 }
42 rel(home, secondSpeciesArsisArray[secondSpeciesArsisArray.size()-1], IRT_EQ, expr(home,
    secondSpeciesNotesCp[secondSpeciesNotesCp.size()-2]-
43     secondSpeciesNotesCp[secondSpeciesNotesCp.size()-1]));
44
45 secondSpeciesMotions = IntVarArray(home, (nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1)
    , IntSet{-1, CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION});
46 secondSpeciesMotionCosts = IntVarArray(home, (nMeasures* notesPerMeasure.at(FIRST_SPECIES)
    -1), IntSet{0, directCost, obliqueCost, contraryCost});
47
48 //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
49 //m intervals between two arsis notes and then calculates the motion with respect to the
    cantusFirmus
50 for(int i = 0; i < secondSpeciesArsisArray.size(); i++){
51     //direct motions help creation
52     BoolVar both_up = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->getMelodicIntervals
    ())>0)); //if both parts are going in the same direction
53     BoolVar both_stay = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->
    getMelodicIntervals()[i]==0)); //if both parts are staying
54     BoolVar both_down = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->
    getMelodicIntervals()[i]<0)); //if both parts are going down
55     //oblique motions help creation
56     BoolVar cf_stays_1 = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->
    getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
57     BoolVar cf_stays_2 = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->
    getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
58     BoolVar cp_stays_1 = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->
    getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
59     BoolVar cp_stays_2 = expr(home, (secondSpeciesArsisArray[i]==0)&&(low->
    getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
60     //contrary motions help creation
61     BoolVar cpd_cfu = expr(home, (secondSpeciesArsisArray[i]<0)&&(low->getMelodicIntervals
    ())>0)); //if the cf goes up and the cp down
62     BoolVar cpu_cfd = expr(home, (secondSpeciesArsisArray[i]>0)&&(low->getMelodicIntervals
    ())<0)); //if the cf goes down and the cp up
63
64     //direct constraints
65     rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >> (
    secondSpeciesMotions[i]==PARALLEL_MOTION));
66     rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i]==1)) >> (
    secondSpeciesMotionCosts[i]==directCost));
67     //oblique constraints
68     rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]
    )==1)) >> (secondSpeciesMotions[i]==OBLIQUE_MOTION));
69     rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]
    )==1)) >> (secondSpeciesMotionCosts[i]==obliqueCost));
70     //contrary constraints
71     rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (secondSpeciesMotions[i]==
    CONTRARY_MOTION));
72     rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i]==1)) >> (secondSpeciesMotionCosts[i]
    )==contraryCost));
73     //bass constraints
74     rel(home, (isNotLowest[i]==0) >> (secondSpeciesMotions[i]==-1));
75     rel(home, (isNotLowest[i]==0) >> (secondSpeciesMotionCosts[i]==0));
76 }
77
78 //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
79 //a measure. to get this, we iterate in increments of 2 starting from i=0
80 secondSpeciesRealMotions = IntVarArray(home, secondSpeciesMotions.size(), IntSet{-1,
    CONTRARY_MOTION, OBLIQUE_MOTION, PARALLEL_MOTION});

```

```

81 secondSpeciesRealMotionCosts = IntVarArray(home, secondSpeciesRealMotions.size(), IntSet
    {0, directCost, obliqueCost, contraryCost});
82 for(int i = 0; i < secondSpeciesRealMotions.size(); i++){
83     rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==1) >> (
        secondSpeciesRealMotions[i]==secondSpeciesMotions[i]));
84     rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==0) >> (
        secondSpeciesRealMotions[i]==firstSpeciesMotions[i]));
85
86     rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==1) >> (
        secondSpeciesRealMotionCosts[i]==secondSpeciesMotionCosts[i]));
87     rel(home, (expr(home, abs(secondSpeciesMelodicIntervals[i*2])>4)==0) >> (
        secondSpeciesRealMotionCosts[i]==firstSpeciesMotionCosts[i]));
88 }
89
90 //create isDiminution Array
91 isDiminution = BoolVarArray(home, (nMeasures* notesPerMeasure.at(FIRST_SPECIES) -1), 0, 1)
    ;
92
93 for(int i = 0; i < isDiminution.size(); i++){
94
95     BoolVar btt3 = BoolVar(home, 0, 1);
96     BoolVar btt4 = BoolVar(home, 0, 1);
97     BoolVar bta2nd = BoolVar(home, 0, 1);
98     BoolVar btt3rd = BoolVar(home, 0, 1);
99     BoolVar bat2nd = BoolVar(home, 0, 1);
100     BoolVar band = BoolVar(home, 0, 1);
101
102     rel(home, expr(home, abs(firstSpeciesMelodicIntervals[i])), IRT_EQ, 3, Reify(btt3));
103     rel(home, expr(home, abs(firstSpeciesMelodicIntervals[i])), IRT_EQ, 4, Reify(btt4));
104     rel(home, expr(home, abs(secondSpeciesMelodicIntervals[i*2])), IRT_LQ, 2, Reify(bta2nd
    ));
105     rel(home, expr(home, abs(secondSpeciesMelodicIntervals[i*2+1])), IRT_LQ, 2, Reify(
    bat2nd));
106     rel(home, btt3, BOT_OR, btt4, btt3rd);
107     rel(home, bta2nd, BOT_AND, btt3rd, band);
108     rel(home, band, BOT_AND, bat2nd, isDiminution[i]);
109
110 }
111
112 penultCostArray = IntVarArray(home, 1, IntSet({0, penultCost}));
113 /// Constraints
114
115 // 2.H2 : Arsis harmonies cannot be dissonant except if there is a diminution.
116 H2_2_arsisHarmoniesCannotBeDissonant(home, this);
117
118 //2.M1
119 M1_2_octaveLeap(home, this, low);
120
121 //2.P2 : battuta adapted
122 P3_2_noBattuta(home, this);
123
124 }
125
126 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf,int
    lb, int ub, Stratum* low, CantusFirmus* c, int v_type
127 , vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV) :
128 SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV)
129 {
130     costs = IntVarArray(home, 6, 0, 1000000);
131     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "penult"};
132
133     // 2.H3 : penult cost
134     H3_2_penultimateNoteDomain(home, this);
135     //can do better than this?
136     M2_2_2v_twoConsecutiveNotesAreNotTheSame(home, this);
137
138     //2.P1 : adapted no direct motion rule from first species to real motions
139     P1_2_2v_noDirectMotionFromPerfectConsonance(home, this);
140
141     //set cost[0] to be fifth cost
142     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), fifthCostArray.size()))), costs);

```

```

143 //set cost[1] to be octave cost
144 add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
    SECOND_SPECIES), octaveCostArray.size()))), costs);
145 //set cost[2] to be motion cost
146 add_cost(home, 2, secondSpeciesRealMotionCosts, costs);
147 //set cost[3] to be melodic cost
148 add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    SECOND_SPECIES), melodicDegreeCost.size()))), costs);
149 //need to set cost[4] to be off cost
150 add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
    SECOND_SPECIES), offCostArray.size()))), costs);
151 //set cost[5] to be penult sixth cost
152 add_cost(home, 5, penultCostArray, costs);
153
154 }
155
156 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf, int
    lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
157 vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2) :
158 SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV2)
159 {
160     costs = IntVarArray(home, 8, 0, 1000000);
161
162     varietyCostArray = IntVarArray(home, 3*(secondSpeciesHarmonicIntervals.size()-2), IntSet
        ({0, varietyCost}));
163     directCostArray = IntVarArray(home, secondSpeciesRealMotions.size()-1, IntSet({0,
        directMoveCost}));
164
165     // 2.H3 : penult cost
166     H3_2_penultimateNoteDomain(home, this);
167     //P1 3 voices version
168     P1_2_3v_noDirectMotionFromPerfectConsonance(home, this);
169
170     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct", "
        penult"};
171     //need to set cost[0] to be off cost
172     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), offCostArray.size()))), costs);
173     //set cost[1] to be fifth cost
174     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), fifthCostArray.size()))), costs);
175     //set cost[2] to be octave cost
176     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), octaveCostArray.size()))), costs);
177     //need to set cost[3] to be variety cost
178     add_cost(home, 3, varietyCostArray, costs);
179     //set cost[4] to be motion cost
180     add_cost(home, 4, secondSpeciesRealMotionCosts, costs);
181     //set cost[5] to be melodic cost
182     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
        SECOND_SPECIES), melodicDegreeCost.size()))), costs);
183     //need to set cost[6] to be direct cost
184     add_cost(home, 6, directCostArray, costs);
185     //set cost[7] to be penult sixth cost
186     add_cost(home, 7, penultCostArray, costs);
187
188 }
189
190 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, int size, vector<int> cf, int
    lb, int ub, Stratum* low, CantusFirmus* c, int v_type,
191 vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2,
    int nV3) :
192 SecondSpeciesCounterpoint(home, size, cf, lb, ub, SECOND_SPECIES, low, c, v_type, m_costs,
    g_costs, s_costs, bm, nV3)
193 {
194
195     costs = IntVarArray(home, 8, 0, 1000000);
196     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "variety", "direct", "
        penult"};
197
198     varietyCostArray = IntVarArray(home, 3*(secondSpeciesHarmonicIntervals.size()-2), IntSet
        ({0, varietyCost}));

```

```

199     directCostArray = IntVarArray(home, secondSpeciesRealMotions.size()-1, IntSet({0, 2,
200         directMoveCost}));
201
202     // 2.H3 : penult cost
203     H3_2_penultimateNoteDomain(home, this);
204     //P1 4 voices version
205     P1_2_4v_noDirectMotionFromPerfectConsonance(home, this);
206
207     //set cost[0] to be fifth cost
208     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
209         SECOND_SPECIES), fifthCostArray.size()))), costs);
210     //set cost[1] to be octave cost
211     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
212         SECOND_SPECIES), octaveCostArray.size()))), costs);
213     //set cost[2] to be motion cost
214     add_cost(home, 2, secondSpeciesRealMotionCosts, costs);
215     //set cost[3] to be melodic cost
216     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
217         SECOND_SPECIES), melodicDegreeCost.size()))), costs);
218     //need to set cost[4] to be off cost
219     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
220         SECOND_SPECIES), offCostArray.size()))), costs);
221     //need to set cost[5] to be variety cost
222     add_cost(home, 5, varietyCostArray, costs);
223     //need to set cost[6] to be direct cost
224     add_cost(home, 6, directCostArray, costs);
225     //set cost[7] to be penult sixth cost
226     add_cost(home, 7, penultCostArray, costs);
227 }
228
229 string SecondSpeciesCounterpoint::to_string() const {
230     string text = "\nSecond species diminution array : " + boolVarArray_to_string(isDiminution
231         ) + "\n";
232     text += "Second species isLowest array : " + boolVarArray_to_string(isNotLowest) + "\n";
233     text += "First species m intervals : " + intVarArray_to_string(
234         firstSpeciesMelodicIntervals) + "\n";
235     text += "Second species m intervals : " + intVarArray_to_string(
236         secondSpeciesMelodicIntervals) + "\n";
237     text += "Second species h intervals : " + intVarArray_to_string(
238         secondSpeciesHarmonicIntervals) + "\n";
239     text += "Second species is Consonance : " + boolVarArray_to_string(isConsonance) + "\n";
240     text += "Second species notes : " + intVarArray_to_string(notes) + "\n";
241     text += "First species : " + FirstSpeciesCounterpoint::to_string() + "\n";
242     return text;
243 }
244
245 SecondSpeciesCounterpoint::SecondSpeciesCounterpoint(Home home, SecondSpeciesCounterpoint &s):
246     FirstSpeciesCounterpoint(home, s){
247     secondSpeciesNotesCp.update(home, s.secondSpeciesNotesCp);
248     secondSpeciesHarmonicIntervals.update(home, s.secondSpeciesHarmonicIntervals);
249     secondSpeciesMelodicIntervals.update(home, s.secondSpeciesMelodicIntervals);
250     secondSpeciesMotions.update(home, s.secondSpeciesMotions);
251     secondSpeciesMotionCosts.update(home, s.secondSpeciesMotionCosts);
252     secondSpeciesRealMotionCosts.update(home, s.secondSpeciesRealMotionCosts);
253
254     secondSpeciesArsisArray.update(home, s.secondSpeciesArsisArray);
255 }
256
257 SecondSpeciesCounterpoint* SecondSpeciesCounterpoint::clone(Home home){
258     return new SecondSpeciesCounterpoint(home, *this);
259 }
260
261 IntVarArray SecondSpeciesCounterpoint::getFirstHInterval(){
262     return firstSpeciesHarmonicIntervals;
263 }
264
265 IntVarArray SecondSpeciesCounterpoint::getMotions(){
266     return secondSpeciesMotions;
267 }
268
269 IntVarArray SecondSpeciesCounterpoint::getFirstMInterval(){
270     return firstSpeciesMelodicIntervals;
271 }

```

```

262 }
263
264 IntVarArray SecondSpeciesCounterpoint::getBranchingNotes(){
265     return secondSpeciesNotesCp;
266 }
267
268 int SecondSpeciesCounterpoint::getHIntervalSize(){
269     return secondSpeciesHarmonicIntervals.size();
270 }

```

C.5.7 ThirdSpeciesCounterpoint.cpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #include "../headers/Parts/ThirdSpeciesCounterpoint.hpp"
6
7 /**
8  * Third species general constructor
9  */
10 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf, int lb,
11     int ub, int mSpec, Stratum* low, CantusFirmus* c,
12     int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV)
13 :
14     FirstSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs,
15     g_costs, s_costs, bm, nV)
16 {
17     thirdSpeciesNotesCp = IntVarArray(home, notes.size(), IntSet(IntArgs(vector_intersection(
18     cp_range, extended_domain))));
19     if(borrowMode==1){
20         thirdSpeciesNotesCp[thirdSpeciesNotesCp.size()-2] = IntVar(home, IntSet(IntArgs(
21     vector_intersection(cp_range, chromatic_scale))));
22     }
23     rel(home, thirdSpeciesNotesCp, IRT_EQ, notes.slice(0,4/notesPerMeasure.at(THIRD_SPECIES),(
24     notes.size())));
25
26     thirdSpeciesHarmonicIntervals = IntVarArray(home, h_intervals.size(), -PERFECT_OCTAVE,
27     PERFECT_OCTAVE);
28     rel(home, thirdSpeciesHarmonicIntervals, IRT_EQ, h_intervals);
29     for(int i = 0; i < thirdSpeciesHarmonicIntervals.size(); i++){
30         rel(home, (thirdSpeciesHarmonicIntervals[i])=((thirdSpeciesNotesCp[i]-low->getNotes()
31     [floor(i/4)*4])%12));
32     }
33
34     thirdSpeciesMelodicIntervals = IntVarArray(home, m_intervals_brut.size(), -PERFECT_OCTAVE,
35     PERFECT_OCTAVE);
36
37     for(int i = 0; i < thirdSpeciesMelodicIntervals.size(); i++)
38         rel(home, thirdSpeciesMelodicIntervals[i], IRT_EQ, expr(home, thirdSpeciesNotesCp[i+1]
39     - thirdSpeciesNotesCp[i]));
40
41     rel(home, thirdSpeciesMelodicIntervals, IRT_EQ, m_intervals_brut.slice(0,4/notesPerMeasure
42     .at(THIRD_SPECIES),m_intervals_brut.size()));
43
44     for(int i = 0; i < thirdSpeciesMelodicIntervals.size(); i++){
45         rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i])), IRT_GR, 0);
46     }
47
48     thirdSpeciesMotions = IntVarArray(home, notes.size()/4, IntSet({-1, CONTRARY_MOTION,
49     OBLIQUE_MOTION, PARALLEL_MOTION}));
50     thirdSpeciesMotionCosts = IntVarArray(home, notes.size()/4, IntSet{0, directCost,
51     obliqueCost, contraryCost});
52
53     for(int i = 0; i < thirdSpeciesMotions.size(); i++){
54         //direct motions help creation
55         BoolVar both_up = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(low->
56     getMelodicIntervals()[i]>0)); //if both parts are going in the same direction

```

```

45     BoolVar both_stay = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->
getMelodicIntervals()[i]==0)); //if both parts are staying
46     BoolVar both_down = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(low->
getMelodicIntervals()[i]<0)); //if both parts are going down
47     //oblique motions help creation
48     BoolVar cf_stays_1 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(low->
getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes up
49     BoolVar cf_stays_2 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(low->
getMelodicIntervals()[i]==0)); //if the lowest part stays and one goes down
50     BoolVar cp_stays_1 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->
getMelodicIntervals()[i]>0)); //if the lowest part goes up and one stays
51     BoolVar cp_stays_2 = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]==0)&&(low->
getMelodicIntervals()[i]<0)); //if the lowest part goes down and one stays
52     //contrary motions help creation
53     BoolVar cpd_cfu = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]<0)&&(low->
getMelodicIntervals()[i]>0)); //if the cf goes up and the cp down
54     BoolVar cpu_cfd = expr(home, (thirdSpeciesMelodicIntervals[(i*4)+3]>0)&&(low->
getMelodicIntervals()[i]<0)); //if the cf goes down and the cp up
55
56     //direct constraints
57     rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i])) >> (
thirdSpeciesMotions[i]==PARALLEL_MOTION));
58     rel(home, ((both_up || both_stay || both_down) && (isNotLowest[i])) >> (
thirdSpeciesMotionCosts[i]==directCost));
59     //oblique constraints
60     rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]))
>> (thirdSpeciesMotions[i]==OBLIQUE_MOTION));
61     rel(home, ((cf_stays_1 || cf_stays_2 || cp_stays_1 || cp_stays_2) && (isNotLowest[i]))
>> (thirdSpeciesMotionCosts[i]==obliqueCost));
62     //contrary constraints
63     rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i])) >> (thirdSpeciesMotions[i]==
CONTRARY_MOTION));
64     rel(home, ((cpd_cfu || cpu_cfd) && (isNotLowest[i])) >> (thirdSpeciesMotionCosts[i]==
contraryCost));
65     //bass constraints
66     rel(home, (!isNotLowest[i]) >> (thirdSpeciesMotions[i]==-1));
67     rel(home, (!isNotLowest[i]) >> (thirdSpeciesMotionCosts[i]==0));
68 }
69
70 thirdSpeciesAbsMelodic = IntVarArray(home, thirdSpeciesMelodicIntervals.size(), UNISSON,
PERFECT_OCTAVE);
71 for(int i = 0; i < thirdSpeciesAbsMelodic.size(); i++){
72     rel(home, thirdSpeciesAbsMelodic[i] == abs(thirdSpeciesMelodicIntervals[i]));
73 }
74
75 m2IntervalsArray = IntVarArray(home, thirdSpeciesNotesCp.size()-2, -PERFECT_OCTAVE,
PERFECT_OCTAVE);
76 for(int i = 0; i < thirdSpeciesNotesCp.size()-2; i++){
77     //size-2 of notes (butlast butlast) ; from 2 onwards in notes (rest rest)
78     rel(home, m2IntervalsArray[i], IRT_EQ, expr(home, thirdSpeciesNotesCp[i+2]-
thirdSpeciesNotesCp[i]));
79 }
80
81 is5QNArray = BoolVarArray(home, nMeasures-1, 0, 1);
82 for(int i = 0; i < is5QNArray.size()*4; i+=4){
83
84     BoolVar b1 = BoolVar(home, 0, 1);
85     BoolVar b2 = BoolVar(home, 0, 1);
86     BoolVar b3 = BoolVar(home, 0, 1);
87     BoolVar b4 = BoolVar(home, 0, 1);
88     BoolVar bb1 = BoolVar(home, 0, 1);
89     BoolVar bb2 = BoolVar(home, 0, 1);
90     BoolVar bb3 = BoolVar(home, 0, 1);
91     BoolVar bb4 = BoolVar(home, 0, 1);
92     BoolVar band1 = BoolVar(home, 0, 1);
93     BoolVar band2 = BoolVar(home, 0, 1);
94     BoolVar band3 = BoolVar(home, 0, 1);
95     BoolVar beq1 = BoolVar(home, 0, 1);
96     BoolVar beq2 = BoolVar(home, 0, 1);
97     BoolVar beq3 = BoolVar(home, 0, 1);
98
99     rel(home, thirdSpeciesAbsMelodic[i], IRT_LQ, 2, Reify(b1));
100    rel(home, thirdSpeciesAbsMelodic[i+1], IRT_LQ, 2, Reify(b2));

```

```

101     rel(home, thirdSpeciesAbsMelodic[i+2], IRT_LQ, 2, Reify(b3));
102     rel(home, thirdSpeciesAbsMelodic[i+3], IRT_LQ, 2, Reify(b4));
103
104     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 ;
121 for(int i = 0; i < isDiminution.size()*4; i+=4){
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 );
133     rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i+2])), IRT_LQ, 2, Reify(bat2nd)
134 );
135     rel(home, btt3, BOT_OR, btt4, btt3rd);
136     rel(home, bta2nd, BOT_AND, btt3rd, band);
137     rel(home, band, BOT_AND, bat2nd, isDiminution[i/4]);
138
139 }
140
141 cambiataCostArray = IntVarArray(home, nMeasures-1, IntSet({0, cambiataCost}));
142
143 m2ZeroArray = IntVarArray(home, thirdSpeciesMelodicIntervals.size()-2, IntSet({0,
144 m2ZeroCost}));
145
146 //third note of the penultimate measure must be below the fourth one
147 rel(home, thirdSpeciesMelodicIntervals[thirdSpeciesMelodicIntervals.size()-2], IRT_GR, 1);
148 //second one must also be more distant than a semi tone from the last note of the
149 penultimate measure
150 rel(home, expr(home, thirdSpeciesMelodicIntervals[thirdSpeciesMelodicIntervals.size()-3]+
151 thirdSpeciesMelodicIntervals[thirdSpeciesMelodicIntervals.size()-2])
152 , IRT_NQ, 1);
153
154 //3.H1 : five consecutive notes by joint degree implies that the first and the third note
155 are consonants
156 H1_3_fiveConsecutiveNotesByJointDegree(home, this);
157
158 //3.H2 : any dissonant note implies that it is surrounded by consonant notes
159 H2_3_disonanceImpliesDiminution(home, this);
160
161 //3.H3 : cambiata cost
162 H3_3_cambiataCost(home, this);
163
164 //no melodic interval between 9 and 11
165 for(int i = 0; i < nMeasures-1; i++){
166     rel(home, abs(thirdSpeciesMelodicIntervals[(i*4)+3])!=MAJOR_SIXTH && abs(
167 thirdSpeciesMelodicIntervals[(i*4)+3])!=MINOR_SEVENTH && abs(thirdSpeciesMelodicIntervals
168 [(i*4)+3])!=MAJOR_SEVENTH);
169 }
170
171 //3.M1 : each note and its two beats further peer are preferred to be different
172 //i + i+1 + i+2

```

```

165     for(int i = 0; i < m2ZeroArray.size(); i++){
166         rel(home, ((thirdSpeciesMelodicIntervals[i]+thirdSpeciesMelodicIntervals[i+1]+
167         thirdSpeciesMelodicIntervals[i+2])==0) >> (m2ZeroArray[i]==m2ZeroCost));
168     }
169
170     //marcel's rule
171     /*for(int i = 0; i < thirdSpeciesMelodicIntervals.size()-1; i++){
172         BoolVar bSkip = BoolVar(home, 0, 1);
173         BoolVar bMbUp = BoolVar(home, 0, 1);
174         BoolVar bMbDown = BoolVar(home, 0, 1);
175         BoolVar bContrary = BoolVar(home, 0, 1);
176
177         rel(home, expr(home, abs(thirdSpeciesMelodicIntervals[i])), IRT_GR, 2, Reify(bSkip));
178         rel(home, thirdSpeciesMelodicIntervals[i], IRT_GR, 0, Reify(bMbUp));
179         rel(home, thirdSpeciesMelodicIntervals[i+1], IRT_LE, 1, Reify(bMbDown));
180         rel(home, bMbUp, BOT_EQV, bMbDown, bContrary);
181         rel(home, thirdSpeciesMelodicIntervals[i+1], IRT_LQ, 2, Reify(bSkip, RM_IMP));
182         rel(home, bSkip, BOT_IMP, bContrary, 1);
183     }*/
184
185     //no battuta adapted for third species
186     for(int j = 0; j < thirdSpeciesMotions.size(); j++){
187         rel(home, expr(home, thirdSpeciesMotions[j]==CONTRARY_MOTION &&
188         firstSpeciesHarmonicIntervals[j+1]==0 && thirdSpeciesMelodicIntervals[(j*4)+3]<-4),
189         BOT_AND, isNotLowest[j], 0);
190     }
191 }
192
193 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
194 int ub, Stratum* low, CantusFirmus* c, int v_type,
195 vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV):
196 ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs,
197 g_costs, s_costs, bm, nV)
198 {
199     //3.H4 : in the penultimate measure, if the cantusFirmus is in the upper part, then the
200     //interval of the first note should be a minor third
201     rel(home, (getIsNotLowest()[getIsNotLowest().size()-2]==0) >>
202     (expr(home, abs(thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-5])
203     )==MINOR_THIRD));
204
205     costs = IntVarArray(home, 7, 0, 10000);
206     cost_names = {"fifth", "octave", "motion", "melodic", "borrow", "cambiata", "m2"};
207
208     //set cost[0] to be fifth cost
209     add_cost(home, 0, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
210     THIRD_SPECIES), fifthCostArray.size()))), costs);
211     //set cost[1] to be octave cost
212     add_cost(home, 1, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
213     THIRD_SPECIES), octaveCostArray.size()))), costs);
214     //set cost[2] to be motion cost
215     add_cost(home, 2, thirdSpeciesMotionCosts, costs);
216     //set cost[3] to be melodic cost
217     add_cost(home, 3, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
218     THIRD_SPECIES), melodicDegreeCost.size()))), costs);
219     //need to set cost[4] to be off cost
220     add_cost(home, 4, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
221     THIRD_SPECIES), offCostArray.size()))), costs);
222     //need to set cost[5] to be cambiata cost
223     add_cost(home, 5, cambiataCostArray, costs);
224     //need to set cost[5] to be cambiata cost
225     add_cost(home, 6, m2ZeroArray, costs);
226 }
227
228 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf,int lb,
229 int ub, Stratum* low, CantusFirmus* c, int v_type,
230 vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2):
231 ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs,
232 g_costs, s_costs, bm, nV2)
233 {
234     varietyCostArray = IntVarArray(home, 3*(thirdSpeciesHarmonicIntervals.size()-2), IntSet
235     ({0, varietyCost}));

```

```

224     directCostArray = IntVarArray(home, thirdSpeciesMotions.size()-1, IntSet({0, directMoveCost
    }));
225
226     //1.H7,H8 adapted
227     rel(home, thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==UNISSON||
    thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==MINOR_THIRD||
    thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==PERFECT_FIFTH||
228         thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==MAJOR_SIXTH);
229
230     //3.H6 : harmonic triad should be used on the second or third beat
231     thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
232     for(int i = 0; i < thirdHTriadArray.size(); i++){
233         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)&&
234             (thirdSpeciesHarmonicIntervals[(i*4)+2]!=UNISSON&&thirdSpeciesHarmonicIntervals[(i
    *4)+2]!=MINOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)+2]!=MAJOR_THIRD&&
    thirdSpeciesHarmonicIntervals[(i*4)+2]!=PERFECT_FIFTH)) >>
235             (thirdHTriadArray[i]==triad3rdCost));
236         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)||
237             (thirdSpeciesHarmonicIntervals[(i*4)+2]==UNISSON||thirdSpeciesHarmonicIntervals[(i
    *4)+2]==MINOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)+2]==MAJOR_THIRD||
    thirdSpeciesHarmonicIntervals[(i*4)+2]==PERFECT_FIFTH)) >>
238             (thirdHTriadArray[i]==0));
239     }
240
241     //1.P1 3 voices version
242     for(int j = 0; j < firstSpeciesMotions.size()-1; j++){
243         //set a cost when it is reached through direct motion, it is 0 when not
244         rel(home, (thirdSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||
    firstSpeciesHarmonicIntervals[j+1]==7))>>
245             (directCostArray[j]==directMoveCost));
246         rel(home, (thirdSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&
    firstSpeciesHarmonicIntervals[j+1]!=7))>>
247             (directCostArray[j]==0));
248     }
249
250     costs = IntVarArray(home, 10, 0, 10000);
251     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct", "
    cambiata", "m2", "triad3"};
252     //need to set cost[0] to be off cost
253     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), offCostArray.size()))), costs);
254     //set cost[1] to be fifth cost
255     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), fifthCostArray.size()))), costs);
256     //set cost[2] to be octave cost
257     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), octaveCostArray.size()))), costs);
258     //need to set cost[3] to be variety cost
259     add_cost(home, 3, varietyCostArray, costs);
260     //set cost[4] to be motion cost
261     add_cost(home, 4, thirdSpeciesMotionCosts, costs);
262     //set cost[5] to be melodic cost
263     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
    THIRD_SPECIES), melodicDegreeCost.size()))), costs);
264     //need to set cost[6] to be direct cost
265     add_cost(home, 6, directCostArray, costs);
266     //set cost[7] to be cambiata sixth cost
267     add_cost(home, 7, cambiataCostArray, costs);
268     //set cost[8] to be m2 zero cost
269     add_cost(home, 8, m2ZeroArray, costs);
270     //set cost[9] to be triad h third cost
271     add_cost(home, 9, thirdHTriadArray, costs);
272
273 }
274
275 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, int size, vector<int> cf, int lb,
    int ub, Stratum* low, CantusFirmus* c, int v_type,
276     vector<int> m_costs, vector<int> g_costs, vector<int> s_costs, int bm, int nV1, int nV2,
    int nV3):

```

```

277 ThirdSpeciesCounterpoint(home, size, cf, lb, ub, THIRD_SPECIES, low, c, v_type, m_costs,
278 { g_costs, s_costs, bm, nv3)
279 {
280     varietyCostArray = IntVarArray(home, 3*(thirdSpeciesHarmonicIntervals.size()-2), IntSet
281     ({0, varietyCost}));
282     directCostArray = IntVarArray(home, thirdSpeciesMotions.size()-1, IntSet({0, 2,
283     directMoveCost}));
284
285     //1.H7,H8 adapted
286     //rel(home, thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==UNISSON
287     ||thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==MINOR_THIRD||
288     thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==PERFECT_FIFTH||
289     // thirdSpeciesHarmonicIntervals[thirdSpeciesHarmonicIntervals.size()-1]==MAJOR_SIXTH);
290
291     //3.H6 : harmonic triad should be used on the second or third beat
292     thirdHTriadArray = IntVarArray(home, nMeasures-1, IntSet({0, triad3rdCost}));
293     for(int i = 0; i < thirdHTriadArray.size(); i++){
294         rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]!=UNISSON&&
295         thirdSpeciesHarmonicIntervals[(i*4)+1]!=MINOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)
296         +1]!=MAJOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)+1]!=PERFECT_FIFTH)&&
297         (thirdSpeciesHarmonicIntervals[(i*4)+2]!=UNISSON&&thirdSpeciesHarmonicIntervals[(i
298         *4)+2]!=MINOR_THIRD&&thirdSpeciesHarmonicIntervals[(i*4)+2]!=MAJOR_THIRD&&
299         thirdSpeciesHarmonicIntervals[(i*4)+2]!=PERFECT_FIFTH)) >>
300         (thirdHTriadArray[i]==triad3rdCost));
301         rel(home, ((thirdSpeciesHarmonicIntervals[(i*4)+1]==UNISSON||
302         thirdSpeciesHarmonicIntervals[(i*4)+1]==MINOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)
303         +1]==MAJOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)+1]==PERFECT_FIFTH)||
304         (thirdSpeciesHarmonicIntervals[(i*4)+2]==UNISSON||thirdSpeciesHarmonicIntervals[(i
305         *4)+2]==MINOR_THIRD||thirdSpeciesHarmonicIntervals[(i*4)+2]==MAJOR_THIRD||
306         thirdSpeciesHarmonicIntervals[(i*4)+2]==PERFECT_FIFTH)) >>
307         (thirdHTriadArray[i]==0));
308     }
309
310     //1.P1 3 voices version
311     for(int j = 0; j < firstSpeciesMotions.size()-1; j++){
312         //set a cost when it is reached through direct motion, it is 0 when not
313         rel(home, (thirdSpeciesMotions[j]==2&&(firstSpeciesHarmonicIntervals[j+1]==0||
314         firstSpeciesHarmonicIntervals[j+1]==7))>>
315         (directCostArray[j]==directMoveCost));
316         rel(home, (thirdSpeciesMotions[j]!=2||(firstSpeciesHarmonicIntervals[j+1]!=0&&
317         firstSpeciesHarmonicIntervals[j+1]!=7))>>
318         (directCostArray[j]==0));
319     }
320
321     costs = IntVarArray(home, 10, 0, 10000);
322     cost_names = {"borrow", "fifth", "octave", "variety", "motion", "melodic", "direct", "
323     cambiata", "m2", "triad3"};
324     //need to set cost[0] to be off cost
325     add_cost(home, 0, IntVarArray(home, offCostArray.slice(0, 4/notesPerMeasure.at(
326     THIRD_SPECIES), offCostArray.size()))), costs);
327     //set cost[1] to be fifth cost
328     add_cost(home, 1, IntVarArray(home, fifthCostArray.slice(0, 4/notesPerMeasure.at(
329     THIRD_SPECIES), fifthCostArray.size()))), costs);
330     //set cost[2] to be octave cost
331     add_cost(home, 2, IntVarArray(home, octaveCostArray.slice(0, 4/notesPerMeasure.at(
332     THIRD_SPECIES), octaveCostArray.size()))), costs);
333     //need to set cost[3] to be variety cost
334     add_cost(home, 3, varietyCostArray, costs);
335     //set cost[4] to be motion cost
336     add_cost(home, 4, thirdSpeciesMotionCosts, costs);
337     //set cost[5] to be melodic cost
338     add_cost(home, 5, IntVarArray(home, melodicDegreeCost.slice(0, 4/notesPerMeasure.at(
339     THIRD_SPECIES), melodicDegreeCost.size()))), costs);
340     //need to set cost[6] to be direct cost
341     add_cost(home, 6, directCostArray, costs);
342     //set cost[7] to be cambiata sixth cost
343     add_cost(home, 7, cambiataCostArray, costs);
344     //set cost[8] to be m2 zero cost
345     add_cost(home, 8, m2ZeroArray, costs);
346     //set cost[9] to be triad h third cost
347     add_cost(home, 9, thirdHTriadArray, costs);
348 }
349

```

```

330 string ThirdSpeciesCounterpoint::to_string() const {
331     string text = Part::to_string() + "\nThird species isConsonance intervals : \n";
332     text += boolVarArray_to_string(isConsonance) += "\n";
333     text += "Third species is Lowest : \n";
334     text += boolVarArray_to_string(isNotLowest) += "\n";
335     text += "Third species motions : \n";
336     text += intVarArray_to_string(thirdSpeciesMotions) += "\n";
337     text += "Third species m intervals : \n";
338     text += intVarArray_to_string(thirdSpeciesMelodicIntervals) += "\n";
339     text += "m intervals : \n";
340     text += intVarArray_to_string(m_intervals_brut) += "\n";
341     text += "Third species is5QN array : \n";
342     text += boolVarArray_to_string(is5QNArray) += "\n";
343     return text;
344 }
345
346 ThirdSpeciesCounterpoint::ThirdSpeciesCounterpoint(Home home, ThirdSpeciesCounterpoint &s):
    FirstSpeciesCounterpoint(home, s){
347     thirdSpeciesNotesCp.update(home, s.thirdSpeciesNotesCp);
348     thirdSpeciesMotions.update(home, s.thirdSpeciesMotions);
349     thirdSpeciesMotionCosts.update(home, s.thirdSpeciesMotionCosts);
350     m2IntervalsArray.update(home, s.m2IntervalsArray);
351     m2ZeroArray.update(home, s.m2ZeroArray);
352     thirdHTriadArray.update(home, s.thirdHTriadArray);
353     thirdSpeciesAbsMelodic.update(home, s.thirdSpeciesAbsMelodic);
354 }
355
356 ThirdSpeciesCounterpoint* ThirdSpeciesCounterpoint::clone(Home home){
357     return new ThirdSpeciesCounterpoint(home, *this);
358 }
359
360 IntVarArray ThirdSpeciesCounterpoint::getFirstHInterval(){
361     return firstSpeciesHarmonicIntervals;
362 }
363
364 IntVarArray ThirdSpeciesCounterpoint::getMotions(){
365     return thirdSpeciesMotions;
366 }
367
368 IntVarArray ThirdSpeciesCounterpoint::getFirstMInterval(){
369     return firstSpeciesMelodicIntervals;
370 }
371
372 IntVarArray ThirdSpeciesCounterpoint::getBranchingNotes(){
373     return thirdSpeciesNotesCp;
374 }
375
376 int ThirdSpeciesCounterpoint::getHIntervalSize(){
377     return thirdSpeciesHarmonicIntervals.size();
378 }

```

C.6 Other source files

C.6.1 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
10         goes through every note of the counterpoint
11         IntVarArray res = IntVarArray(home, part->getOffDomain().size(), 0, 1);
12         IntVar sm = IntVar(home, 0, part->getOffDomain().size());
13         for(int l = 0; l < part->getOffDomain().size(); l++){ //loop
14             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());
18 for(int t = 0; t < part->getOffDomain().size(); t++){
19     x[t] = res[t]; //
create an IntVarArgs array of the contents of res[l] (for sum)
20 }
21 rel(home, sm, IRT_EQ, expr(home, sum(x))); //set
the sm variable to be the sum of the IntVarArgs
22 rel(home, sm, IRT_GR, 0, Reify(part->getIsOffArray()[i])); //if
the sum is greater than 0, then the note is borrowed (off key)
23 }
24 }
25
26 void twoFifthSpeciesDiversity_3v(Home home, Part* cp1, Part* cp2){
27     if(cp1->getSpecies()==FIFTH_SPECIES && cp2->getSpecies()==FIFTH_SPECIES){
28         BoolVarArray isSameSpecies = BoolVarArray(home, cp1->getNotes().size(), 0, 1);
29         IntVarArray isSameSpeciesInt = IntVarArray(home, cp1->getNotes().size(), 0, 1);
30         IntVar percentageSame = IntVar(home, 0, cp1->getNotes().size());
31         for(int i = 0; i < cp1->getNotes().size(); i++){
32             rel(home, cp1->getSpeciesArray()[i], IRT_EQ, cp2->getSpeciesArray()[i], Reify(
isSameSpecies[i]));
33             rel(home, isSameSpeciesInt[i], IRT_EQ, 1, Reify(isSameSpecies[i]));
34         }
35         rel(home, percentageSame, IRT_EQ, expr(home, sum(isSameSpeciesInt)));
36         rel(home, percentageSame, IRT_LE, floor(cp1->getNotes().size()/2));
37     }
38 }
39
40 void noMinorSecondBetweenUpper(Home home, vector<Part*> parts){
41     for(int v1 = 0; v1 < parts.size(); v1++){
42         for(int v2 = v1+1; v2 < parts.size(); v2++){
43             for(int i = 0; i < parts[v1]->getNMeasures(); i++){
44                 BoolVar noneLowest = BoolVar(home, 0, 1);
45                 rel(home, parts[v1]->getIsNotLowest()[i], IRT_EQ, parts[v2]->getIsNotLowest()[
i], Reify(noneLowest, RM_PMI));
46
47                 IntVar interval = IntVar(home, -PERFECT_OCTAVE, PERFECT_OCTAVE);
48                 if(i==parts[v1]->getNMeasures()-1){
49                     rel(home, interval == ((parts[v1]->getFirstNotes()[i]-parts[v2]->
getFirstNotes()[i])%12));
50                 }
51                 else if(parts[v1]->getSpecies()!=FOURTH_SPECIES&&parts[v2]->getSpecies()!=
FOURTH_SPECIES){
52                     rel(home, interval == ((parts[v1]->getFirstNotes()[i]-parts[v2]->
getFirstNotes()[i])%12));
53                 } else if(parts[v1]->getSpecies()==FOURTH_SPECIES&&parts[v2]->getSpecies()!=
FOURTH_SPECIES){
54                     rel(home, interval == ((parts[v1]->getNotes()[i*4+2]-parts[v2]->
getFirstNotes()[i])%12));
55                 } else if(parts[v1]->getSpecies()!=FOURTH_SPECIES&&parts[v2]->getSpecies()==
FOURTH_SPECIES){
56                     rel(home, interval == ((parts[v1]->getFirstNotes()[i]-parts[v2]->getNotes
()[i*4+2])%12));
57                 } else {
58                     rel(home, interval == ((parts[v1]->getNotes()[i*4+2]-parts[v2]->getNotes
()[i*4+2])%12));
59                 }
60                 rel(home, (noneLowest==1) >> (expr(home, abs(interval)!=1)));
61             }
62         }
63     }
64 }
65
66 void G4_counterpointMustBeInTheSameKey(Home home, Part* part){
67     for(int i = 0; i < part->getIsOffArray().size(); i++){
68         rel(home, (part->getIsOffArray()[i]==0) >> (part->getOffCostArray()[i]==0));
//sets no cost if not borrowed

```

```

69     rel(home, (part->getIsOffArray()[i]==1) >> (part->getOffCostArray()[i]==part->
70     getBorrowCost())); //sets a cost if borrowed
71 }
72
73 void G6_noChromaticMelodies(Home home, Part* part, int mSpec){
74
75     //forbids that three notes one after the other increase by a step, making the melody
76     chromatic
77
78     for(int i = 0; i < part->getMelodicIntervals().size()-1; i+=4/notesPerMeasure.at(mSpec)){
79         rel(home, expr(home, part->getMelodicIntervals()[i]==1), BOT_AND, expr(home, part->
80         getMelodicIntervals()[i+1]==1), 0);
81         rel(home, expr(home, part->getMelodicIntervals()[i]==-1), BOT_AND, expr(home, part->
82         getMelodicIntervals()[i+1]==-1), 0);
83     }
84 }
85
86 void G7_melodicIntervalsShouldBeSmall(Home home, Part* part, int mSpec){
87
88     //loop goes through every note of the melodic interval and sets the cost
89
90     for(int i = 0; i < part->getMelodicIntervals().size(); i+=4/notesPerMeasure.at(mSpec)){
91         rel(home, (abs(part->getMelodicIntervals()[i])<MINOR_THIRD) >> (part->
92         getMelodicDegreeCost()[i]==part->getSecondCost()));
93         rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_THIRD || abs(part->
94         getMelodicIntervals()[i])==MAJOR_THIRD) >> (part->getMelodicDegreeCost()[i]==part->
95         getThirdCost()));
96         rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_FOURTH) >> (part->
97         getMelodicDegreeCost()[i]==part->getFourthCost()));
98         rel(home, (abs(part->getMelodicIntervals()[i])==TRITONE) >> (part->
99         getMelodicDegreeCost()[i]==part->getTritoneCost()));
100         rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_FIFTH) >> (part->
101         getMelodicDegreeCost()[i]==part->getFifthCost()));
102         rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_SIXTH || abs(part->
103         getMelodicIntervals()[i])==MAJOR_SIXTH) >> (part->getMelodicDegreeCost()[i]==part->
104         getSixthCost()));
105         rel(home, (abs(part->getMelodicIntervals()[i])==MINOR_SEVENTH || abs(part->
106         getMelodicIntervals()[i])==MAJOR_SEVENTH) >> (part->getMelodicDegreeCost()[i]==part->
107         getSeventhCost()));
108         rel(home, (abs(part->getMelodicIntervals()[i])==PERFECT_OCTAVE) >> (part->
109         getMelodicDegreeCost()[i]==part->getOctaveCost()));
110     }
111 }
112
113 void G9_lastChordSameAsFundamental(Home home, Stratum* lowest, Part* cantusFirmus){
114
115     //we check the last note of the cantusFirmus to set the last note of the counterpoint to
116     be in the same key
117
118     rel(home, expr(home, lowest->getNotes()[lowest->getNotes().size()-1]%12), IRT_EQ, expr(
119     home, cantusFirmus->getNotes()[cantusFirmus->getNotes().size()-1]%12));
120 }
121
122 void H1_1_harmonicIntervalsAreConsonances(Home home, Part* part){
123
124     //we also include negative values, since the fourth species could render the harmonic
125     intervals to be negative because of its displacement
126     //mostly the harmonic intervals are going to be positive
127
128     dom(home, part->getFirstSpeciesHIntervals(), IntSet({UNISSON, MINOR_THIRD, MAJOR_THIRD,
129     PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH, PERFECT_OCTAVE,
130     -MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FIFTH, -MINOR_SIXTH, -MAJOR_SIXTH, -
131     PERFECT_OCTAVE}));
132 }
133
134 void H1_3_fiveConsecutiveNotesByJointDegree(Home home, Part* part){
135     for(int i = 0; i < part->getIs5QNArray().size(); i++){
136         rel(home, (part->getIs5QNArray()[i]==1) >> (part->getConsonance()[i*(4)+2]==1));
137     }
138 }
139
140 void H2_1_startWithPerfectConsonance(Home home, Part* part){

```

```

122     dom(home, part->getHInterval()[0], IntSet(IntArgs(PERFECT_CONSONANCES)));
123 }
124
125 void H2_2_arsisHarmoniesCannotBeDissonant(Home home, Part* part){
126
127     for(int i = 0; i < part->getNMeasures()-1; i++){
128         if(i != part->getNMeasures()-2){ //if it is the penultimate measure
129             for(int d : DISONANCE){
130                 rel(home, part->getHInterval()[i*4+2], IRT_EQ, d, Reify(part->
131                     getIsDiminution()[i], RM_PMI));
132             }
133         }
134     }
135
136 void H2_3_disonanceImpliesDiminution(Home home, Part* part){
137     for(int i = 0; i < part->getIsDiminution().size(); i++){
138         BoolVar band1 = BoolVar(home, 0, 1);
139         BoolVar band2 = BoolVar(home, 0, 1);
140
141         rel(home, part->getConsonance()[i*4+2], BOT_OR, part->getIsDiminution()[i], 1);
142     }
143 }
144
145 void H3_1_endWithPerfectConsonance(Home home, Part* part){
146     dom(home, part->getHInterval()[part->getHInterval().size()-1], IntSet(IntArgs(
147         PERFECT_CONSONANCES)));
148 }
149
150 void H3_2_penultimateNoteDomain(Home home, Part* part){
151     dom(home, expr(home, abs(part->getHInterval()[part->getHIntervalSize()-3])), IntSet({
152         UNISSON, PERFECT_FIFTH, MINOR_SIXTH, MAJOR_SIXTH}));
153
154     rel(home, (part->getHInterval()[part->getHIntervalSize()-3] != PERFECT_FIFTH) >> (part->
155         getPenultCostArray()[0] == part->getPenultCost()));
156     rel(home, (part->getHInterval()[part->getHIntervalSize()-3] == PERFECT_FIFTH) >> (part->
157         getPenultCostArray()[0] == 0));
158 }
159
160 void H3_3_cambiataCost(Home home, Part* part){
161     for(int i = 0; i < part->getCambiataCostArray().size(); i++){
162         rel(home, ((part->getThirdSpeciesHIntervals()[i*4+1] == UNISSON || part->
163             getThirdSpeciesHIntervals()[i*4+1] == PERFECT_FIFTH) && (part->getThirdSpeciesHIntervals()[i*4+2] == UNISSON || part->getThirdSpeciesHIntervals()[i*4+2] == PERFECT_FIFTH)
164             && (abs(part->getThirdSpeciesMIntervals()[i*4+1]) <= 2)) >> (part->
165             getCambiataCostArray()[i] == part->getCambiataCost()));
166         rel(home, ((part->getThirdSpeciesHIntervals()[i*4+1] != UNISSON && part->
167             getThirdSpeciesHIntervals()[i*4+1] != PERFECT_FIFTH) || (part->getThirdSpeciesHIntervals()[i*4+2] != UNISSON && part->getThirdSpeciesHIntervals()[i*4+2] != PERFECT_FIFTH)
168             || (abs(part->getThirdSpeciesMIntervals()[i*4+1]) > 2)) >> (part->
169             getCambiataCostArray()[i] == 0));
170     }
171 }
172
173 void H5_1_cpAndCfdDifferentNotes(Home home, Part* part, Part* cf){
174     for(int i = 1; i < part->getFirstSpeciesNotes().size()-1; i++){
175         rel(home, part->getFirstSpeciesNotes()[i], IRT_NQ, cf->getNotes()[i]);
176     }
177 }
178
179 void H5_1_differentNotes(Home home, vector<Part*> parts){
180     for(int v1 = 0; v1 < parts.size(); v1++){
181         for(int v2 = v1+1; v2 < parts.size(); v2++){
182             for(int i = 1; i < parts[v1]->getNotes().size()-1; i++){
183                 if(parts[v1]->getSpecies() == CANTUS_FIRMUS){
184                     rel(home, parts[v1]->getNotes()[floor(i/4)], IRT_NQ, parts[v2]->getNotes()
185                         [i]);
186                 } else {
187                     rel(home, parts[v1]->getNotes()[i], IRT_NQ, parts[v2]->getNotes()[i]);
188                 }
189             }
190         }
191     }
192 }

```

```

183 }
184
185 void H6_1_preferImperfectConsonances(Home home, Part* part){
186     for(int i = 0; i < part->getHInterval().size(); i++){
187         //set the octave cost
188         rel(home, part->getOctaveCostArray()[i], IRT_EQ, part->getHOctaveCost(), Reify(expr(
189             home, part->getHInterval()[i]==UNISSON), RM_PMI));
190         rel(home, part->getOctaveCostArray()[i], IRT_EQ, 0, Reify(expr(home, part->
191             getHInterval()[i]!=UNISSON), RM_PMI));
192
193         //set the fifth cost
194         rel(home, part->getFifthCostArray()[i], IRT_EQ, part->getHFifthCost(), Reify(expr(home
195             , part->getHInterval()[i]==PERFECT_FIFTH), RM_PMI));
196         rel(home, part->getFifthCostArray()[i], IRT_EQ, 0, Reify(expr(home, part->getHInterval
197             ())[i]!=PERFECT_FIFTH), RM_PMI));
198     }
199 }
200
201 void H7_1_2v_penultimateSixthOrThird(Home home, Part* part){
202     int p = part->getFirstSpeciesNotes().size()-2; //index of the penultimate note
203
204     //if it is the lowest, then it is a major sixth (cf version in the cantusFirmus class)
205     rel(home, part->getFirstSpeciesHIntervals()[part->getFirstSpeciesHIntervals().size()-2],
206         IRT_EQ, MAJOR_SIXTH,
207         Reify(part->getIsNotLowest()[part->getIsNotLowest().size()-2], RM_IMP));
208 }
209
210 void H7_1_3v_penultimateSixthOrThird(Home home, Part* part){
211     dom(home, part->getFirstSpeciesHIntervals()[part->getFirstSpeciesHIntervals().size()-2],
212         IntSet({UNISSON, MINOR_THIRD, PERFECT_FIFTH, MAJOR_SIXTH}));
213 }
214
215 void H8_3v_preferHarmonicTriad(Home home, Part* part, IntVarArray triadCostArray, Stratum*
216     upper1, Stratum* upper2){
217     for(int i = 0; i < triadCostArray.size(); i++){
218         BoolVar h1_3 = BoolVar(home, 0, 1); //check if the first interval is a minor
219         third
220         BoolVar h1_4 = BoolVar(home, 0, 1); //check if the first interval is a major
221         third
222         BoolVar h1_third = BoolVar(home, 0, 1); //check if the first interval is a third
223         BoolVar h1_7 = BoolVar(home, 0, 1); //check if the first interval is a perfect
224         fifth
225
226         BoolVar h2_3 = BoolVar(home, 0, 1); //check if the second interval is a minor
227         third
228         BoolVar h2_4 = BoolVar(home, 0, 1); //check if the second interval is a major
229         third
230         BoolVar h2_third = BoolVar(home, 0, 1); //check if the second interval is a third
231         BoolVar h2_7 = BoolVar(home, 0, 1); //check if the second interval is a
232         perfect fifth
233
234         BoolVar h_firstPoss = BoolVar(home, 0, 1); //check if the first interval is a third
235         and the second a perfect fifth
236         BoolVar h_secondPoss = BoolVar(home, 0, 1); //check if the second interval is a third
237         and the first a perfect fifth
238         BoolVar triad = BoolVar(home, 0, 1); //check if it is a triad
239         BoolVar not_triad = BoolVar(home, 0, 1); //check if it is not a triad
240
241         //check for the first possibility
242         rel(home, expr(home, abs(upper1->getHInterval()[i*4])), IRT_EQ, 3, Reify(h1_3));
243         rel(home, expr(home, abs(upper1->getHInterval()[i*4])), IRT_EQ, 4, Reify(h1_4));
244         rel(home, expr(home, abs(upper2->getHInterval()[i*4])), IRT_EQ, 7, Reify(h2_7));
245         rel(home, h1_3, BOT_OR, h1_4, h1_third);
246         rel(home, h1_third, BOT_AND, h2_7, h_firstPoss);
247
248         //check for the second possibility
249         rel(home, expr(home, abs(upper2->getHInterval()[i*4])), IRT_EQ, 3, Reify(h2_3));
250         rel(home, expr(home, abs(upper2->getHInterval()[i*4])), IRT_EQ, 4, Reify(h2_4));
251         rel(home, expr(home, abs(upper1->getHInterval()[i*4])), IRT_EQ, 7, Reify(h1_7));
252         rel(home, h2_3, BOT_OR, h2_4, h2_third);
253         rel(home, h2_third, BOT_AND, h1_7, h_secondPoss);
254     }
255 }

```

```

240     rel(home, h_firstPoss, BOT_OR, h_secondPoss, triad); //is a triad if it is either
the first or the second possibility
241     rel(home, triad, BOT_XOR, not_triad, 1); //set not triad
242     rel(home, triadCostArray[i], IRT_EQ, 0, Reify(triad, RM_IMP));
243     rel(home, triadCostArray[i], IRT_EQ, part->getTriadCost(), Reify(not_triad, RM_IMP));
244 }
245 }
246
247 void H8_4v_preferHarmonicTriad(Home home, IntVarArray triadCostArray, Stratum* upper1, Stratum
* upper2, Stratum* upper3){
248     for(int i = 0; i < triadCostArray.size(); i++){
249
250         IntVar H_b = upper1->getHInterval()[i*4];
251         IntVar H_c = upper2->getHInterval()[i*4];
252         IntVar H_d = upper3->getHInterval()[i*4];
253
254
255         BoolVar H_b_is_third = expr(home, H_b==MINOR_THIRD || H_b==MAJOR_THIRD);
256         BoolVar H_b_is_fifth = expr(home, H_b==PERFECT_FIFTH);
257         BoolVar H_b_is_octave = expr(home, H_b==UNISSON);
258
259         BoolVar H_c_is_third = expr(home, H_c==MINOR_THIRD || H_c==MAJOR_THIRD);
260         BoolVar H_c_is_fifth = expr(home, H_c==PERFECT_FIFTH);
261         BoolVar H_c_is_octave = expr(home, H_c==UNISSON);
262
263         BoolVar H_d_is_third = expr(home, H_d==MINOR_THIRD || H_d==MAJOR_THIRD);
264         BoolVar H_d_is_fifth = expr(home, H_d==PERFECT_FIFTH);
265         BoolVar H_d_is_octave = expr(home, H_d==UNISSON);
266
267
268         // worst case : not a harmonic triad with at least a third and a fifth
269         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);
270         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);
271
272         rel(home, (note_outside_harmonic_triad || no_fifth_or_no_third) >> (triadCostArray[i]
== not_harmonic_triad_cost));
273
274         // now we are left with only combinations with at a third, a fifth, and another note
of the harmonic triad.
275         // Doubling the fifth
276         rel(home, expr(home, H_b_is_fifth + H_c_is_fifth + H_d_is_fifth == 2) >> (
triadCostArray[i] == double_fifths_cost));
277
278         // Doubling the third, and ensure it is the same type of third (not a major and a
minor)
279         rel(home, expr(home, H_b_is_third + H_c_is_third + H_d_is_third == 2) >> (
triadCostArray[i] == double_thirds_cost));
280         rel(home, (H_b_is_third && H_c_is_third) >> (H_b == H_c));
281         rel(home, (H_b_is_third && H_d_is_third) >> (H_b == H_d));
282         rel(home, (H_c_is_third && H_d_is_third) >> (H_c == H_d));
283
284         // triad with octave
285         rel(home, (H_b_is_fifth && H_c_is_third && H_d_is_octave) >> (triadCostArray[i] ==
triad_with_octave_cost)); // 5 3 8
286         rel(home, (H_b_is_third && H_c_is_octave && H_d_is_fifth) >> (triadCostArray[i] ==
triad_with_octave_cost)); // 3 8 5
287         rel(home, (H_b_is_third && H_c_is_fifth && H_d_is_octave) >> (triadCostArray[i] ==
triad_with_octave_cost)); // 3 5 8
288         rel(home, (H_b_is_octave && H_c_is_third && H_d_is_fifth) >> (triadCostArray[i] ==
triad_with_octave_cost)); // 8 3 5
289         rel(home, (H_b_is_octave && H_c_is_fifth && H_d_is_third) >> (triadCostArray[i] ==
triad_with_octave_cost)); // 8 5 3
290
291         // Best case : 5 8 3
292         rel(home, (H_b_is_fifth && H_c_is_octave && H_d_is_third) >> (triadCostArray[i] == 0))
;
293     }
294 }
295 }
296

```

```

297 void M1_1_2v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part){
298     rel(home, part->getFirstSpeciesMIntervals(), IRT_LQ, MINOR_SIXTH);
299     rel(home, part->getFirstSpeciesMIntervals(), IRT_GQ, -MINOR_SIXTH);
300 }
301
302 void M1_1_3v_melodicIntervalsNotExceedMinorSixth(Home home, Part* part){
303     dom(home, part->getMelodicIntervals(), IntSet({-UNISSON, -MINOR_SECOND, -MAJOR_SECOND, -
304         MINOR_THIRD, -MAJOR_THIRD, -PERFECT_FOURTH, -TRITONE,
305         -PERFECT_FIFTH, -MINOR_SIXTH, -PERFECT_OCTAVE, UNISSON, MINOR_SECOND, MAJOR_SECOND
306         , MINOR_THIRD, MAJOR_THIRD, PERFECT_FOURTH, TRITONE,
307         PERFECT_FIFTH, MINOR_SIXTH, PERFECT_OCTAVE}));
308 }
309
310 void M1_2_octaveLeap(Home home, Part* part, Stratum* low){
311     for(int j = 0; j < part->getFirstSpeciesMIntervals().size(); j++){
312         rel(home, part->getFirstSpeciesMIntervals()[j], IRT_EQ, 12, Reify(expr(home, (abs(part
313             ->getFirstSpeciesHIntervals()[j])>=4)&&
314             (expr(home, expr(home, abs(low->getMelodicIntervals()[j])>0) == part->
315             getIsNotLowest()[j]))), RM_PMI));
316     }
317 }
318
319 void M2_2_2v_twoConsecutiveNotesAreNotTheSame(Home home, Part* part){
320     rel(home, part->getSecondSpeciesMIntervals(), IRT_NQ, 0);
321 }
322
323 void M2_2_3v_melodicIntervalsNotExceedMinorSixth(Home home, vector<Part*> parts, bool
324     containsThirdSpecies){
325     for(int i = 1; i < parts.size(); i++){
326         if(parts[i]->getSpecies()==THIRD_SPECIES){
327             containsThirdSpecies=1;
328             break;
329         }
330     }
331     for(int p1 = 1; p1 < parts.size(); p1++){
332         for(int p2 = 1; p2 < parts.size(); p2++){
333             if(p1!=p2 && parts[p1]->getSpecies()==SECOND_SPECIES){
334                 if(!containsThirdSpecies){
335                     for(int i = 0; i < parts[p1]->getBranchingNotes().size()-4; i++){
336                         rel(home, parts[p1]->getMelodicIntervals().slice(0, notesPerMeasure.at
337                             (SECOND_SPECIES), parts[p1]->getMelodicIntervals().size())[i],
338                             IRT_NQ, 0);
339                     }
340                 } else {
341                     for(int i = 0; i < parts[p1]->getBranchingNotes().size()-4; i++){
342                         rel(home, parts[p1]->getMelodicIntervals().slice(0, notesPerMeasure.at
343                             (SECOND_SPECIES), parts[p1]->getMelodicIntervals().size())[i],
344                             IRT_NQ, 0);
345                     }
346                     //no unison in the two last notes
347                     rel(home, parts[p1]->getBranchingNotes()[parts[p1]->getBranchingNotes().
348                         size()-2], IRT_NQ, parts[p1]->getBranchingNotes()[parts[p1]->getBranchingNotes().size()
349                         -1]);
350                 }
351             }
352         }
353     }
354 }
355
356 void M2_1_varietyCost(Home home, vector<Part*> parts){
357     for(int i = 1; i < parts.size(); i++){
358         Part* p = parts[i];
359         cout << "Branching notes size : " << endl;
360         cout << p->getBranchingNotes().size() << endl;
361         int temp = 0;
362         IntVarArray notes = p->getBranchingNotes();
363         for(int j = 0; j < p->getHIntervalSize()-1; j++){
364             cout << j << endl;
365             int upbnd = 0;
366             if(j+3<p->getHIntervalSize()){
367                 upbnd = j+4;
368             } else {upbnd = p->getHIntervalSize();}
369             for(int k = j+1; k < upbnd;k++){

```

```

361         //setting a cost if notes inside a window are the same in a part
362         rel(home, (notes[j]==notes[k])>>(p->getVarietyArray(temp)==p->getVarietyCost()
    ));
363         rel(home, (notes[j]!=notes[k])>>(p->getVarietyArray(temp)==0));
364         temp++;
365     }
366 }
367 }
368 }
369 }
370
371 void P1_1_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
372     for(int j = 0; j < part->getFirstSpeciesMotions().size(); j++){
373
374         rel(home, ((part->getFirstSpeciesHIntervals()[j+1]==UNISSON || part->
getFirstSpeciesHIntervals()[j+1]==PERFECT_FIFTH)&&part->getIsNotLowest()[j+1]==1) >>
375             (part->getFirstSpeciesMotions()[j]!=PARALLEL_MOTION));
376
377     }
378 }
379
380 void P1_1_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
381     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
382
383         //if the counterpoint is the lowest part -> the cost is 0
384         rel(home, (!part->getIsNotLowest()[j]) >> (part->getDirectCostArray()[j]==0));
385
386         //if the counterpoint is neither the lowest nor the highest part, and the condition is
true, then add a cost of 2
387         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part->
getFirstSpeciesMotions()[j]==2&&(part->getFirstSpeciesHIntervals()[j+1]==0||
388             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))==7))) >> (part->
getDirectCostArray()[j]==2));
389         //else the cost is 0
390         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part->
getFirstSpeciesMotions()[j]!=2||(part->getFirstSpeciesHIntervals()[j+1]!=0
391             && expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))!=7))) >> (part->
getDirectCostArray()[j]==0));
392
393         //if the counterpoint is the highest part and the condition is true, then add a direct
move cost (8 in this case)
394         rel(home, (part->getIsHighest()[j] && (part->getFirstSpeciesMotions()[j]==2&&(part->
getFirstSpeciesHIntervals()[j+1]==0||
395             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))==7))) >> (part->
getDirectCostArray()[j]==part->getDirectMoveCost()));
396         //else the cost is 0
397         rel(home, (part->getIsHighest()[j] && (part->getFirstSpeciesMotions()[j]!=2||(part->
getFirstSpeciesHIntervals()[j+1]!=0 &&
398             expr(home, abs(part->getFirstSpeciesHIntervals()[j+1]))!=7))) >> (part->
getDirectCostArray()[j]==0));
399
400     }
401 }
402
403 void P1_2_2v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
404     for(int j = 0; j < part->getSecondSpeciesRealMotions().size(); j++){
405         rel(home, ((part->getFirstSpeciesHIntervals()[j+1]==UNISSON || part->
getFirstSpeciesHIntervals()[j+1]==PERFECT_FIFTH)&&part->getIsNotLowest()[j+1]==1) >>
406             (part->getSecondSpeciesRealMotions()[j]!=PARALLEL_MOTION));
407     }
408 }
409
410 void P1_2_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
411
412     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
413         rel(home, (part->getSecondSpeciesRealMotions()[j]==2&&(part->getFirstSpeciesHIntervals
()[j+1]==0||part->getFirstSpeciesHIntervals()[j+1]==7))>>
414             (part->getDirectCostArray()[j]==part->getDirectMoveCost()));
415         rel(home, (part->getSecondSpeciesRealMotions()[j]!=2||(part->getFirstSpeciesHIntervals
()[j+1]!=0&&part->getFirstSpeciesHIntervals()[j+1]!=7))>>
416             (part->getDirectCostArray()[j]==0));
417     }
418 }

```

```

419 }
420
421 void P1_2_4v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
422     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
423
424         rel(home, (!part->getIsNotLowest()[j]) >> (part->getDirectCostArray()[j]==0));
425
426         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part->
getSecondSpeciesRealMotions()[j]==2&&(part->getFirstSpeciesHIntervals()[j+1]==0||
427             part->getFirstSpeciesHIntervals()[j+1]==7))) >> (part->getDirectCostArray()[j]==2)
);
428         rel(home, (part->getIsNotLowest()[j] && !part->getIsHighest()[j] && (part->
getSecondSpeciesRealMotions()[j]!=2||part->getFirstSpeciesHIntervals()[j+1]!=0 &&
429             part->getFirstSpeciesHIntervals()[j+1]!=7))) >> (part->getDirectCostArray()[j]==0)
);
430
431         rel(home, (part->getIsHighest()[j] && (part->getSecondSpeciesRealMotions()[j]==2&&(
part->getFirstSpeciesHIntervals()[j+1]==0||
432             part->getFirstSpeciesHIntervals()[j+1]==7))) >> (part->getDirectCostArray()[j]==
part->getDirectMoveCost()));
433         rel(home, (part->getIsHighest()[j] && (part->getSecondSpeciesRealMotions()[j]!=2|| (
part->getFirstSpeciesHIntervals()[j+1]!=0 &&
434             part->getFirstSpeciesHIntervals()[j+1]!=7))) >> (part->getDirectCostArray()[j]==0)
);
435
436     }
437 }
438
439 void P3_0_noBattuta(Home home, Part* part){
440     for(int j = 0; j < part->getMotions().size(); j++){
441         rel(home, expr(home, part->getMotions()[j]==CONTRARY_MOTION && part->getHInterval()[j
+1]==0 &&
442             part->getMelodicIntervals()[j]<-4), BOT_AND, part->getIsNotLowest()[j], 0);
443     }
444 }
445
446 void P3_1_noBattuta(Home home, Part* part){
447     for(int j = 0; j < part->getFirstSpeciesMotions().size(); j++){
448         rel(home, expr(home, part->getFirstSpeciesMotions()[j]==CONTRARY_MOTION && part->
getFirstSpeciesHIntervals()[j+1]==0 &&
449             part->getFirstSpeciesMIntervals()[j]<-4), BOT_AND, part->getIsNotLowest()[j], 0);
450     }
451 }
452
453 void P3_2_noBattuta(Home home, Part* part){
454     for(int j = 0; j < part->getSecondSpeciesMotions().size(); j++){
455         rel(home, expr(home, part->getSecondSpeciesMotions()[j]==CONTRARY_MOTION && part->
getFirstSpeciesHIntervals()[j+1]==0 &&
456             part->getFirstSpeciesMIntervals()[j]<-4), BOT_AND, part->getIsNotLowest()[j], 0);
457     }
458 }
459
460 void P4_successiveCost(Home home, vector<Part*> parts, int scc_cz, IntVarArray
successiveCostArray, vector<Species> species){
461     int idx = 0;
462     for(int v1 = 1; v1 < parts.size(); v1++){
463         for(int v2 = v1+1; v2 < parts.size(); v2++){
464             IntVarArray hIntervals12 = IntVarArray(home, parts[v1]->getNMeasures(), 0,
MAJOR_SEVENTH);
465             BoolVarArray isPCons12 = BoolVarArray(home, parts[v1]->getNMeasures(), 0, 1);
466
467             if(parts[v1]->getSpecies()==FOURTH_SPECIES || parts[v2]->getSpecies()==
FOURTH_SPECIES){
468                 if(parts[v1]->getSpecies()==FOURTH_SPECIES && parts[v2]->getSpecies()==
FOURTH_SPECIES){
469                     for(int i = 0; i < parts[v1]->getNMeasures()-1; i++){
470                         rel(home, hIntervals12[i]==(abs(parts[v1]->getNotes()[i*4+2]-parts[v2
]->getNotes()[i*4+2])%12));
471                     }
472                 } else if(parts[v1]->getSpecies()==FOURTH_SPECIES && parts[v2]->getSpecies()!=
FOURTH_SPECIES){
473                     for(int i = 0; i < parts[v1]->getNMeasures()-1; i++){

```

```

474         rel(home, hIntervals12[i]==(abs(parts[v1]->getNotes()[i*4+2]-parts[v2]
]->getNotes()[i*4])%12));
475     }
476     } else{
477         for(int i = 0; i < parts[v1]->getNMeasures()-1; i++){
478             rel(home, hIntervals12[i]==(abs(parts[v1]->getNotes()[i*4]-parts[v2]->
getNotes()[i*4+2])%12));
479         }
480     }
481     rel(home, hIntervals12[hIntervals12.size()-1]==(abs(parts[v1]->getNotes()[
parts[v1]->getNotes().size()-1]-
482     parts[v2]->getNotes()[parts[v2]->getNotes().size()-1])%12));
483 } else {
484     for(int i = 0; i < parts[v1]->getNMeasures(); i++){
485         rel(home, hIntervals12[i]==(abs(parts[v1]->getNotes()[i*4]-parts[v2]->
getNotes()[i*4])%12));
486     }
487 }
488
489     for(int i = 0; i < hIntervals12.size(); i++){
490         rel(home, expr(home, hIntervals12[i]==UNISSON), BOT_OR, expr(home,
hIntervals12[i]==PERFECT_FIFTH), isPCons12[i]);
491     }
492
493     if(parts[v1]->getSpecies()!=SECOND_SPECIES && parts[v2]->getSpecies()!=
SECOND_SPECIES && parts[v1]->getSpecies()!=FOURTH_SPECIES &&
494     parts[v2]->getSpecies()!=FOURTH_SPECIES)
495     {
496         for(int i = 0; i < isPCons12.size()-1; i++){
497             BoolVar succPCons = BoolVar(home, 0, 1);
498             rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], succPCons);
499             rel(home, (succPCons==1) >> (successiveCostArray[idx]==parts[v1]->
getSuccCost()));
500             rel(home, (succPCons==0) >> (successiveCostArray[idx]==0));
501             idx++;
502         }
503     }
504     else if(parts[v1]->getSpecies()==SECOND_SPECIES){
505         for(int i = 0; i < isPCons12.size()-1; i++){
506             BoolVar firstNotFifth = BoolVar(home, 0, 1);
507             BoolVar secondNotFifth = BoolVar(home, 0, 1);
508             BoolVar notSuccessiveFifths = BoolVar(home, 0, 1);
509             BoolVar succPCons = BoolVar(home, 0, 1);
510             BoolVar succPConsAndNotSuccFifths = BoolVar(home, 0, 1);
511
512             BoolVar mNotThird1 = BoolVar(home, 0, 1);
513             BoolVar mNotThird2 = BoolVar(home, 0, 1);
514             BoolVar mNotThird = BoolVar(home, 0, 1);
515             BoolVar firstFifth = BoolVar(home, 0, 1);
516             BoolVar secondFifth = BoolVar(home, 0, 1);
517             BoolVar succFifth = BoolVar(home, 0, 1);
518             BoolVar succFifthNotThird = BoolVar(home, 0, 1);
519
520             BoolVar applyCost = BoolVar(home, 0, 1);
521
522             rel(home, hIntervals12[i], IRT_NQ, 7, Reify(firstNotFifth));
523             rel(home, hIntervals12[i+1], IRT_NQ, 7, Reify(secondNotFifth));
524             rel(home, firstNotFifth, BOT_OR, secondNotFifth, notSuccessiveFifths);
525
526             rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], succPCons);
527             rel(home, succPCons, BOT_AND, notSuccessiveFifths,
succPConsAndNotSuccFifths);
528
529             rel(home, parts[v1]->getMelodicIntervals()[i*2], IRT_NQ, 3, mNotThird1);
530             rel(home, parts[v1]->getMelodicIntervals()[i*2], IRT_NQ, 4, mNotThird2);
531             rel(home, mNotThird1, BOT_AND, mNotThird2, mNotThird);
532
533             rel(home, hIntervals12[i], IRT_EQ, 7, Reify(firstFifth));
534             rel(home, hIntervals12[i+1], IRT_EQ, 7, Reify(secondFifth));
535             rel(home, firstFifth, BOT_AND, secondFifth, succFifth);
536             rel(home, mNotThird, BOT_AND, succFifth, succFifthNotThird);
537

```

```

538         rel(home, succPConsAndNotSuccFifths, BOT_OR, succFifthNotThird, applyCost)
539     ;
540     rel(home, (applyCost==1) >> (successiveCostArray[idx]==parts[v1]->
getSuccCost()));
541     rel(home, (applyCost==0) >> (successiveCostArray[idx]==0));
542
543     idx++;
544 }
545 else if(parts[v2]->getSpecies()==SECOND_SPECIES){
546     for(int i = 0; i < isPCons12.size()-1; i++){
547         BoolVar firstNotFifth = BoolVar(home, 0, 1);
548         BoolVar secondNotFifth = BoolVar(home, 0, 1);
549         BoolVar notSuccessiveFifths = BoolVar(home, 0, 1);
550         BoolVar succPCons = BoolVar(home, 0, 1);
551         BoolVar succPConsAndNotSuccFifths = BoolVar(home, 0, 1);
552
553         BoolVar mNotThird1 = BoolVar(home, 0, 1);
554         BoolVar mNotThird2 = BoolVar(home, 0, 1);
555         BoolVar mNotThird = BoolVar(home, 0, 1);
556         BoolVar firstFifth = BoolVar(home, 0, 1);
557         BoolVar secondFifth = BoolVar(home, 0, 1);
558         BoolVar succFifth = BoolVar(home, 0, 1);
559         BoolVar succFifthNotThird = BoolVar(home, 0, 1);
560
561         BoolVar applyCost = BoolVar(home, 0, 1);
562
563         rel(home, hIntervals12[i], IRT_NQ, 7, Reify(firstNotFifth));
564         rel(home, hIntervals12[i+1], IRT_NQ, 7, Reify(secondNotFifth));
565         rel(home, firstNotFifth, BOT_OR, secondNotFifth, notSuccessiveFifths);
566
567         rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], succPCons);
568         rel(home, succPCons, BOT_AND, notSuccessiveFifths,
succPConsAndNotSuccFifths);
569
570         rel(home, parts[v2]->getMelodicIntervals()[i*2], IRT_NQ, 3, mNotThird1);
571         rel(home, parts[v2]->getMelodicIntervals()[i*2], IRT_NQ, 4, mNotThird2);
572         rel(home, mNotThird1, BOT_AND, mNotThird2, mNotThird);
573
574         rel(home, hIntervals12[i], IRT_EQ, 7, Reify(firstFifth));
575         rel(home, hIntervals12[i+1], IRT_EQ, 7, Reify(secondFifth));
576         rel(home, firstFifth, BOT_AND, secondFifth, succFifth);
577         rel(home, mNotThird, BOT_AND, succFifth, succFifthNotThird);
578
579         rel(home, succPConsAndNotSuccFifths, BOT_OR, succFifthNotThird, applyCost)
580     ;
581     rel(home, (applyCost==1) >> (successiveCostArray[idx]==parts[v1]->
getSuccCost()));
582     rel(home, (applyCost==0) >> (successiveCostArray[idx]==0));
583
584     idx++;
585 } else if(parts[v1]->getSpecies()==FOURTH_SPECIES || parts[v2]->getSpecies()==
FOURTH_SPECIES){
586     for(int i = 0; i < hIntervals12.size()-1; i++){
587         BoolVar firstNotFifth = BoolVar(home, 0, 1);
588         BoolVar secondNotFifth = BoolVar(home, 0, 1);
589         BoolVar notSuccessiveFifths = BoolVar(home, 0, 1);
590
591         BoolVar succPCons = BoolVar(home, 0, 1);
592         BoolVar succPConsNotFifths = BoolVar(home, 0, 1);
593
594         rel(home, hIntervals12[i], IRT_NQ, 7, Reify(firstNotFifth));
595         rel(home, hIntervals12[i+1], IRT_NQ, 7, Reify(secondNotFifth));
596         rel(home, firstNotFifth, BOT_OR, secondNotFifth, notSuccessiveFifths);
597
598         rel(home, isPCons12[i], BOT_AND, isPCons12[i+1], succPCons);
599         rel(home, succPCons, BOT_AND, notSuccessiveFifths, succPConsNotFifths);
600         rel(home, (succPConsNotFifths==1) >> (successiveCostArray[idx]==parts[v1]
->getSuccCost()));
601         rel(home, (succPConsNotFifths==0) >> (successiveCostArray[idx]==0));
602         idx++;
603     }

```

```

604     }
605 }
606 }
607 }
608
609 void P6_noMoveInSameDirection(Home home, vector<Part*> parts){
610     for(int i = 0; i < parts[0]->getMotions().size(); i++){
611         rel(home, expr(home, parts[0]->getMotions()[i]==2 && parts[1]->getMotions()[i]==2),
612             BOT_AND, expr(home, parts[2]->getMotions()[i]==2), 0);
613     }
614 }
615
616 void P7_noSuccessiveAscendingSixths(Home home, vector<Part*> parts){
617     for(int p1 = 0; p1 < parts.size(); p1++){
618         for(int p2 = p1+1; p2 < parts.size(); p2++){
619             for(int j = 1; j < parts[p1]->getFirstHInterval().size()-1; j++){
620                 rel(home, ((parts[p1]->getFirstHInterval()[j-1]!=MINOR_SIXTH && parts[p1]->
621                     getFirstHInterval()[j-1]!=MAJOR_SIXTH) &&
622                     (parts[p2]->getFirstHInterval()[j-1]!=MINOR_SIXTH && parts[p2]->
623                     getFirstHInterval()[j-1]!=MAJOR_SIXTH)) || (
624                     (parts[p1]->getFirstHInterval()[j]!=-MINOR_SIXTH && parts[p1]->
625                     getFirstHInterval()[j]!=-MAJOR_SIXTH) &&
626                     (parts[p2]->getFirstHInterval()[j]!=-MINOR_SIXTH && parts[p2]->
627                     getFirstHInterval()[j]!=-MAJOR_SIXTH)) || (
628                     parts[p1]->getFirstMInterval()[j]>0 || parts[p2]->getFirstMInterval()[
629                     j] > 0));
630             }
631         }
632     }
633 }
634
635 void P1_1_3v_noDirectMotionFromPerfectConsonance(Home home, Part* part){
636     for(int j = 0; j < part->getFirstSpeciesMotions().size()-1; j++){
637         //set a cost when it is reached through direct motion, it is 0 when not
638         rel(home, (part->getFirstSpeciesMotions()[j]==2&&(part->getFirstSpeciesHIntervals()[j
639             +1]==0||part->getFirstSpeciesHIntervals()[j+1]==7))>>
640             (part->getDirectCostArray()[j]==part->getDirectCost()));
641         rel(home, (part->getFirstSpeciesMotions()[j]!=2||(part->getFirstSpeciesHIntervals()[j
642             +1]!=0&&part->getFirstSpeciesHIntervals()[j+1]!=7))>>
643             (part->getDirectCostArray()[j]==0));
644     }
645 }

```

C.6.2 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
4 // counterpoints.
5 //
6 #include "../headers/CounterpointUtils.hpp"
7
8
9 Part* create_counterpoint(Home home, int species, int nMeasures, vector<int> cantusFirmus, int
10     lowerBound, int upperBound, Stratum* low,
11     CantusFirmus* c, int v_type, vector<int> m_costs, vector<int> g_costs, vector<int> s_costs
12     , int bm, int nV){
13     switch(nV) {
14         case TWO_VOICES :
15             switch (species) { /// call the appropriate constructor for the counterpoint
16                 case FIRST_SPECIES:
17                     return new FirstSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound,
18                         upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
19                     break;
20                 case SECOND_SPECIES:
21                     return new SecondSpeciesCounterpoint(home, nMeasures, cantusFirmus, lowerBound
22                         , upperBound, low, c, v_type, m_costs, g_costs, s_costs, bm, TWO_VOICES);
23                     break;
24                 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
92 , vector<int> m_costs, vector<int> g_costs,
93 vector<int> s_costs, vector<int> imp, int bm){
94
95     switch (spList.size())
96     {
97     case 1:
98         return new TwoVoiceCounterpoint(cf, spList[0], v_type[0], m_costs, g_costs, s_costs,
99         imp, bm);
100     case 2:
101         return new ThreeVoiceCounterpoint(cf, spList, v_type, m_costs, g_costs, s_costs, imp,
102         bm);
103     case 3:
104         return new FourVoiceCounterpoint(cf, spList, v_type, m_costs, g_costs, s_costs, imp,
105         bm);
106     default:
107         throw std::invalid_argument("The number of voices you asked for is not implemented (
108         yet).");
109     }
110 }
111
112 bool notInt(char* argv){
113     bool noInt = false;
114     for(int i = 0; i < strlen(argv); i++){
115         if(!isdigit(argv[i])){
116             noInt = true;
117             break;
118         }
119     }
120     return noInt;
}

```

C.6.3 fuxTest.cpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 // This file contains the testing framework implementation.
4 //
5
6 #include "../headers/fuxTest.hpp"
7
8 FuxTest::FuxTest(char* test){
9
10     cantusFirmus = {60, 62, 65, 64, 67, 65, 64, 62, 60};
11     size = cantusFirmus.size();
12     melodic_params = {0, 1, 1, 576, 2, 2, 2, 1};
13     general_params = {4, 1, 1, 2, 2, 2, 8, 1};
14     specific_params = {8, 4, 0, 2, 1, 8, 50};
15     importance = {8,7,5,2,9,3,14,12,6,11,4,10,1,13};
16     borrowMode = 1;
17     CounterpointProblem* problem;
18     problem = dispatcher(test);
19     BAB<CounterpointProblem> e(problem);
20     int nb_sol = 0;
21     while(CounterpointProblem* pb = e.next()){
22         nb_sol++;
23         cout << pb->to_string() << endl;

```

```

24     delete pb;
25     if (nb_sol >= 1)
26         break;
27 }
28 if(nb_sol==0){
29     cout << "This constraint works. It prohibits a forbidden configuration." << endl;
30 } else {
31     cout << "This constraint does NOT work. It allows a forbidden configuration." << endl;
32 }
33
34 }
35
36 CounterpointProblem* FuxTest::dispatcher(char* test){
37     if(strcmp(test, "1H1")==0){
38         return test_1sp_H1();
39     } else if(strcmp(test, "1H2")==0){
40         return test_1sp_H2();
41     } else if(strcmp(test, "1H32")==0){
42         return test_1sp_H3();
43     } else if(strcmp(test, "1H33")==0){
44         return test_1sp_H3_2();
45     } else if(strcmp(test, "1H41")==0){
46         return test_1sp_H4_1();
47     } else if(strcmp(test, "1H42")==0){
48         return test_1sp_H4_2();
49     } else if(strcmp(test, "1H5")==0){
50         return test_1sp_H5();
51     } else if(strcmp(test, "1H7")==0){
52         return test_1sp_H7();
53     } else if(strcmp(test, "1H72")==0){
54         return test_1sp_H7_2();
55     } else if(strcmp(test, "2H2")==0){
56         return test_2sp_H2();
57     } else if(strcmp(test, "3H1")==0){
58         return test_3sp_H1();
59     } else if(strcmp(test, "4H2")==0){
60         return test_4sp_H2();
61     } else {
62         std::invalid_argument("Test for constraint not found!");
63         return nullptr;
64     }
65 }
66
67 FuxTest::FuxTest(int testNumber): FuxTest(testNumber, 0){
68     idx = cp.size();
69 }
70
71 FuxTest::FuxTest(int testNumber, int i){
72     if(testNumber==22){
73         test_2v_1sp_fig22_setter(i);
74     } else if(testNumber==23){
75         test_2v_1sp_fig23_setter(i);
76     } else if(testNumber==38){
77         test_2v_2sp_fig38_setter(i);
78     } else if(testNumber==39){
79         test_2v_2sp_fig39_setter(i);
80     } else if(testNumber==40){
81         test_2v_2sp_fig40_setter(i);
82     } else if(testNumber==55){
83         test_2v_3sp_fig55_setter(i);
84     } else if(testNumber==74){
85         test_2v_4sp_fig74_setter(i);
86     } else if(testNumber==82){
87         test_2v_5sp_fig82_setter(i);
88     } else if(testNumber==108){
89         test_3v_1sp_fig108_setter(i);
90     } else if(testNumber==109){
91         test_3v_1sp_fig109_setter(i);
92     } else if(testNumber==110){
93         test_3v_1sp_fig110_setter(i);
94     } else if(testNumber==111){
95         test_3v_1sp_fig111_setter(i);
96     } else if(testNumber==125){

```

```

97     test_3v_2sp_fig125_setter(i);
98 } else if(testNumber==133){
99     test_3v_3sp_fig133_setter(i);
100 } else if(testNumber==146){
101     test_3v_4sp_fig146_setter(i);
102 } else if(testNumber==166){
103     test_4v_1sp_fig166_setter(i);
104 } else if(testNumber==167){
105     test_4v_1sp_fig167_setter(i);
106 } else if(testNumber==176){
107     test_4v_2sp_fig176_setter(i);
108 } else if(testNumber==183){
109     test_4v_3sp_fig183_setter(i);
110 }
111 else {
112     throw std::invalid_argument("This test number has not been implemented (yet)");
113 }
114 }
115
116 CounterpointProblem* FuxTest::test_1sp_H1(){
117     spList = {FIRST_SPECIES};
118     v_type = {2};
119     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
120     general_params, specific_params,
121     importance, borrowMode);
122     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 72); //does not work since
123     it is not a consonant
124     return problem;
125 }
126
127 CounterpointProblem* FuxTest::test_1sp_H2(){
128     spList = {FIRST_SPECIES};
129     v_type = {2};
130     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
131     general_params, specific_params,
132     importance, borrowMode);
133     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 64);
134     return problem;
135 }
136
137 CounterpointProblem* FuxTest::test_1sp_H3(){
138     spList = {FIRST_SPECIES};
139     v_type = {2};
140     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
141     general_params, specific_params,
142     importance, borrowMode);
143     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-1], IRT_EQ, 64);
144     return problem;
145 }
146
147 CounterpointProblem* FuxTest::test_1sp_H3_2(){
148     spList = {FIRST_SPECIES, FIRST_SPECIES};
149     v_type = {2, 1};
150     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
151     general_params, specific_params,
152     importance, borrowMode);
153     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()/2-1], IRT_EQ, 67);
154     ;
155     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-1], IRT_EQ, 65);
156     return problem;
157 }
158
159 CounterpointProblem* FuxTest::test_1sp_H4_1(){
160     spList = {FIRST_SPECIES};
161     v_type = {1};
162     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
163     general_params, specific_params,
164     importance, borrowMode);
165     rel(problem->getHome(), problem->getSolutionArray()[0], IRT_EQ, 67);
166     return problem;
167 }
168
169 CounterpointProblem* FuxTest::test_1sp_H4_2(){

```

```

163     spList = {FIRST_SPECIES};
164     v_type = {1};
165     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
166                                   general_params, specific_params,
167                                   importance, borrowMode);
168     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-1], IRT_EQ, 67);
169     return problem;
170 }
171 CounterpointProblem* FuxTest::test_1sp_H5(){
172     spList = {FIRST_SPECIES};
173     v_type = {0};
174     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
175                                   general_params, specific_params,
176                                   importance, borrowMode);
177     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 62);
178     return problem;
179 }
180 CounterpointProblem* FuxTest::test_1sp_H7(){
181     spList = {FIRST_SPECIES};
182     v_type = {0};
183     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
184                                   general_params, specific_params,
185                                   importance, borrowMode);
186     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-2], IRT_EQ, 69);
187     return problem;
188 }
189 CounterpointProblem* FuxTest::test_1sp_H7_2(){
190     spList = {FIRST_SPECIES};
191     v_type = {-1};
192     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
193                                   general_params, specific_params,
194                                   importance, borrowMode);
195     rel(problem->getHome(), problem->getSolutionArray()[problem->getSize()-2], IRT_EQ, 55);
196     return problem;
197 }
198 CounterpointProblem* FuxTest::test_2sp_H2(){
199     spList = {SECOND_SPECIES};
200     v_type = {2};
201     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
202                                   general_params, specific_params,
203                                   importance, borrowMode);
204     //these three lines are an example of the constraint allowing a dissonance
205     rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 78);
206     rel(problem->getHome(), problem->getSolutionArray()[3], IRT_EQ, 79);
207     rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 81);
208     //these three lines are an example of the constraint blocking a dissonance
209     rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 74);
210     rel(problem->getHome(), problem->getSolutionArray()[3], IRT_EQ, 79);
211     rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 81);
212     return problem;
213 }
214 CounterpointProblem* FuxTest::test_3sp_H1(){
215     spList = {THIRD_SPECIES};
216     v_type = {2};
217     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
218                                   general_params, specific_params,
219                                   importance, borrowMode);
220     rel(problem->getHome(), problem->getSolutionArray()[4], IRT_EQ, 66);
221     rel(problem->getHome(), problem->getSolutionArray()[5], IRT_EQ, 66);
222     rel(problem->getHome(), problem->getSolutionArray()[6], IRT_EQ, 67);
223     rel(problem->getHome(), problem->getSolutionArray()[7], IRT_EQ, 69);
224     rel(problem->getHome(), problem->getSolutionArray()[8], IRT_EQ, 69);
225     return problem;
226 }
227 CounterpointProblem* FuxTest::test_4sp_H2(){
228     spList = {FOURTH_SPECIES};
229     v_type = {-1};

```

```

230     auto* problem = create_problem(cantusFirmus, spList, v_type, melodic_params,
231                                   general_params, specific_params,
232                                   importance, borrowMode);
233     rel(problem->getHome(), problem->getSolutionArray()[1], IRT_EQ, 52);
234     rel(problem->getHome(), problem->getSolutionArray()[2], IRT_EQ, 52);
235     return problem;
236 }
237 vector<Species> FuxTest::getSpList(){
238     return spList;
239 }
240
241 vector<int> FuxTest::getCf(){
242     return cantusFirmus;
243 }
244
245 vector<int> FuxTest::getVType(){
246     return v_type;
247 }
248
249 int FuxTest::getBMode(){
250     return borrowMode;
251 }
252
253 vector<int> FuxTest::getCp(){
254     return cp;
255 }
256
257 int FuxTest::getIdx(){
258     return idx;
259 }
260
261 void FuxTest::test_2v_1sp_fig22_setter(int i){
262     cout << "Fig. 22" << endl;
263     spList = {FIRST_SPECIES};
264     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57}; //1sp 2v cf
265     cp = {69,64,67,65,64,72,69,71,71,69,68,69};
266     v_type = {1};
267     idx = i;
268     borrowMode = 1;
269 }
270
271 void FuxTest::test_2v_1sp_fig23_setter(int i){
272     cout << "Fig. 23" << endl;
273     spList = {FIRST_SPECIES};
274     cantusFirmus = {57,60,59,62,60,64,65,64,62,60,59,57}; //1sp 2v cf
275     cp = {57,57,55,53,52,52,50,48,55,57,56,57};
276     v_type = {-1};
277     idx = i;
278     borrowMode = 1;
279 }
280
281 void FuxTest::test_2v_2sp_fig38_setter(int i){
282     cout << "Fig. 38" << endl;
283     spList = {SECOND_SPECIES};
284     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53}; //1sp 2v cf
285     cp = {0,65,
286          64,62,
287          60,58,
288          57,55,
289          53,57,
290          60,58,
291          57,69,
292          67,64,
293          65,67,
294          69,65,
295          62,64,
296          65};
297     v_type = {1};
298     idx = i;
299     borrowMode = 1;
300 }
301

```

```

302 void FuxTest::test_2v_2sp_fig39_setter(int i){
303     cout << "Fig. 39 (the voice types as defined previously don't work here)" << endl;
304     spList = {SECOND_SPECIES};
305     cantusFirmus = {53,55,57,53,50,52,53,60,57,53,55,53}; //1sp 2v cf
306     cp = {0,53,52,48,53,52,50,48,46,58,55,60,57,53,52,48,53,41,45,50,48,52,53};
307     v_type = {-1};
308     idx = i;
309     borrowMode = 1;
310 }
311
312 void FuxTest::test_2v_2sp_fig40_setter(int i){
313     cout << "Fig. 40" << endl;
314     spList = {SECOND_SPECIES};
315     cantusFirmus = {55,60,59,55,60,64,62,67,64,60,62,59,57,55}; //1sp 2v cf
316     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};
317     v_type = {2};
318     idx = i;
319     borrowMode = 1;
320 }
321
322 void FuxTest::test_2v_3sp_fig55_setter(int i){
323     cout << "Fig. 55" << endl;
324     spList = {THIRD_SPECIES};
325     cantusFirmus = {62,65,64,62,67,65,69,67,65,64,62}; //1sp 2v cf
326     cp = {62,64,65,67, //ok
327          69,71,72,74, //ok
328          76,74,71,72, //ok
329          74,72,70,69, //ok
330          70,72,74,76, //ok
331          77,65,69,71, //ok, but check for maybe bemol on last note
332          72,69,70,72, //ok
333          70,69,67,70, //HERE FIRST NOTE ERROR
334          69,62,64,65,
335          67,69,71,73,
336          74};
337     cout << "CP size : " << endl;
338     cout << cp.size() << endl;
339     v_type = {1};
340     idx = i;
341     borrowMode = 1;
342 }
343
344 void FuxTest::test_2v_4sp_fig74_setter(int i){
345     cout << "Fig. 74" << endl;
346     spList = {FOURTH_SPECIES};
347     cantusFirmus = {62, 65, 64, 62, 67, 65, 69, 67, 65, 64, 62}; //1sp 2v
cf
348     cp = { 50,50,62,62,60,60,59,59,64,64,62,62,65,65,64,64,62,62,61,62};
349     cout << cp.size() << endl;
350     v_type = {-1};
351     idx = i;
352     borrowMode = 1;
353 }
354
355 void FuxTest::test_2v_5sp_fig82_setter(int i){
356     cout << "Fig. 82" << endl;
357     spList = {FIFTH_SPECIES};
358     cantusFirmus = {62, 65, 64, 62, 67, 65, 69, 67, 65, 64, 62}; //1sp 2v
cf
359     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};
360     cout << cp.size() << endl;
361     v_type = {1};
362     idx = i;
363     borrowMode = 1;
364 }
365
366 void FuxTest::test_3v_1sp_fig108_setter(int i){
367     cout << "Fig. 108" << endl;
368     spList = {FIRST_SPECIES, FIRST_SPECIES};
369     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
370     cp = {67,64,65,69,72,72,76,72,69,68,

```

```

371         52,57,50,53,53,53,60,60,62,64};
372     v_type = {1,-2};
373     idx = i;
374     borrowMode = 1;
375 }
376
377 void FuxTest::test_3v_1sp_fig109_setter(int i){
378     cout << "Fig. 109" << endl;
379     spList = {FIRST_SPECIES, FIRST_SPECIES};
380     cantusFirmus = {52,48,50,48,45,57,55,52,53,52};
381     cp = {67,64,65,64,64,64,67,67,62,64,
382          59,60,57,57,60,60,59,59,57,59};
383     v_type = {2,0};
384     idx = i;
385     borrowMode = 1;
386 }
387
388 void FuxTest::test_3v_1sp_fig110_setter(int i){
389     cout << "Fig. 110" << endl;
390     spList = {FIRST_SPECIES, FIRST_SPECIES};
391     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
392     cp = {60,60,60,62,58,59,57,69,65,62,64,65,
393          53,52,53,50,55,55,53,45,50,50,48,41};
394     v_type = {-1 ,-3};
395     idx = i;
396     borrowMode = 1;
397 }
398
399 void FuxTest::test_3v_1sp_fig111_setter(int i){
400     cout << "Fig. 111" << endl;
401     spList = {FIRST_SPECIES, FIRST_SPECIES};
402     cantusFirmus = {65,67,69,65,62,64,65,72,69,65,67,65};
403     cp = {69,72,72,74,74,70,69,67,65,77,76,77,
404          65,64,65,62,58,55,53,52,53,50,48,53};
405     v_type = {1 ,-2};
406     idx = i;
407     borrowMode = 1;
408 }
409
410 void FuxTest::test_3v_2sp_fig125_setter(int i){
411     cout << "Fig. 125" << endl;
412     spList = {FIRST_SPECIES, SECOND_SPECIES};
413     cantusFirmus = {52, 48, 50, 48, 45, 57, 55, 52, 53, 52};
414     cp = {64, 64, 65, 64, 65, 64, 67, 67, 69, 68,
415          00,59,60,59,57,53,55,52,57,59,60,57,59,62,64,62,60,57,59};
416     v_type = {2 ,1};
417     idx = i;
418     borrowMode = 1;
419 }
420
421 void FuxTest::test_3v_3sp_fig133_setter(int i){
422     cout << "Fig. 133" << endl;
423     spList = {THIRD_SPECIES, FIRST_SPECIES};
424     cantusFirmus = {62, 65, 64, 62, 67, 65,
425          69, 67, 65, 64, 62};
426     cp = {53,50,53,55, 57,53,57,59, 60,52,55,57, 59,62,59,57, 55,57,59,61,
427          62,64,65,62, 60,57,60,62, 64,62,60,59, 57,62,57,59, 61,57,59,60, 62,
428          50, 50, 48, 55, 52, 50,
429          53, 48, 50, 57, 50};
430     v_type = {-1 ,-2};
431     idx = i;
432     borrowMode = 1;
433 }
434
435 void FuxTest::test_3v_4sp_fig146_setter(int i){
436     cout << "Fig. 146" << endl;
437     spList = {FOURTH_SPECIES, FIRST_SPECIES};
438     cantusFirmus = {64, 60, 62, 60, 57, 69, 67, 64, 65, 64};
439     cp = { 76,76,72,72,71,71,69,69,72,72,74,74,72,72,69,69,71,68,
440          52, 57, 55, 57, 53, 53, 52, 48, 50, 52};
441     v_type = {1 ,-2};
442     idx = i;
443     borrowMode = 1;

```

```

441 }
442
443 void FuxTest::test_4v_1sp_fig166_setter(int i){
444     cout << "Fig. 166" << endl;
445     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
446     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
447     cp =
448         {59,57,57,57,60,60,64,60,62,59,
449          56,57,53,52,52,53,60,60,57,56,
450          52,53,50,45,45,41,40,45,50,52};
451     v_type = {-1, -1, -3};
452     idx = i;
453     borrowMode = 1;
454 }
455
456 void FuxTest::test_4v_1sp_fig167_setter(int i){
457     cout << "Fig. 167" << endl;
458     spList = {FIRST_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
459     cantusFirmus = {64,60,62,60,57,69,67,64,65,64};
460     cp =
461         {68,69,65,64,64,65,72,72,69,68,
462          59,57,57,57,60,60,64,60,62,59,
463          52,53,50,45,45,53,52,57,50,52};
464     v_type = {0, -1, -3};
465     idx = i;
466     borrowMode = 1;
467 }
468
469 void FuxTest::test_4v_2sp_fig176_setter(int i){
470     cout << "Fig. 173" << endl;
471     spList = {SECOND_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
472     cantusFirmus = {50, 53, 52, 50, 55, 53, 57, 55, 53, 52, 50};
473     cp =
474         {00,62,60,62,64,67,65,64,62,64,65,69,64,65,67,62,65,69,67,64,66,
475          65, 69, 72, 74, 70, 69, 69, 71, 74, 73, 74,
476          57, 57, 55, 57, 58, 60, 60, 62, 57, 64, 57};
477     v_type = {2, 3, 1};
478     idx = i;
479     borrowMode = 1;
480 }
481
482 void FuxTest::test_4v_3sp_fig183_setter(int i){
483     cout << "Fig. 183" << endl;
484     spList = {THIRD_SPECIES, FIRST_SPECIES, FIRST_SPECIES};
485     cantusFirmus = {64, 60, 62, 60, 57, 69,
486                    67, 64, 65, 64};
487     cp =
488         {71,64,71,72, 69,72,69,67, 65,62,64,65, 67,64,65,67, 69,65,69,71,
489          72,69,72,74, 76,71,76,74, 72,76,72,71, 69,74,69,71, 68,
490          56, 57, 57, 55, 60, 57,
491          59, 60, 62, 59, 55, 60, 57,
492          52, 53, 50, 52, 53, 53,
493          52, 57, 50, 52};
494     v_type = {0, -1, -2};
495     idx = i;
496     borrowMode = 1;
497 }

```

C.6.4 problem_wrapper.cpp

```

1 //
2 // This file contains the implementations of the C++ interface functions.
3 //
4
5 #include "../headers/problem_wrapper.hpp"
6 #include "../headers/CounterpointProblems/CounterpointProblem.hpp"
7
8 #include "../headers/Utilities.hpp"
9 #include "../headers/CounterpointUtils.hpp"
10
11 std::vector<Species> convertToSpeciesVector(const std::vector<int>& intVec);
12
13
14 /**
15  * Wraps the Problem constructor.

```

```

16  * @return A pointer to a Problem object casted as a void*
17  */
18  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,
19      int* motion_params, int* melodic, int* specific, int* importance, int t_off, int* scl
    , int scale_size,
20      int* chromatic, int chrom_size, int* borrow, int borrow_size){
21
22      writeToLogFile("Entered problem_wrapper.cpp . ");
23
24      vector<int> cf(int_pointer_to_vector(cantusFirmus, size));
25      vector<int> sp(int_pointer_to_vector(splist, n_cp));
26      vector<int> mot(int_pointer_to_vector(motion_params, 3));
27      vector<int> vt(int_pointer_to_vector(v_types, n_cp));
28      vector<int> sc(int_pointer_to_vector(scl, scale_size));
29      vector<int> chr(int_pointer_to_vector(chromatic, chrom_size));
30      vector<int> brw(int_pointer_to_vector(borrow, borrow_size));
31      vector<int> mel(int_pointer_to_vector(melodic, 8));
32      vector<int> gen(int_pointer_to_vector(general_params, 8));
33      vector<int> spec(int_pointer_to_vector(specific, 7));
34      vector<int> imp(int_pointer_to_vector(importance, 14));
35
36      vector<Species> speciesList = convertToSpeciesVector(sp);
37      return create_problem(cf, speciesList, vt, mel, gen, spec, imp, b_mode);
38  }
39
40  /**
41   * returns the size of the problem
42   * @param sp a void* pointer to a CounterpointProblem object
43   * @return an integer representing the size of the problem
44   */
45  int get_size(void* sp){
46      return static_cast<CounterpointProblem*>(sp)->getSize();
47  }
48
49
50  void delete_pointer(void* p){
51      writeToLogFile("deleting counterpointproblem pointer...");
52      delete static_cast<CounterpointProblem*>(p);
53      writeToLogFile("deleted CounterpointProblem pointer . ");
54  }
55
56  void delete_solver_pointer(void* p){
57      delete static_cast<DFS<CounterpointProblem*>>(p);
58      writeToLogFile("deleted solver pointer");
59  }
60
61
62  /**
63   * returns the values of the variables for a solution
64   * @param sp a void* pointer to a Problem object
65   * @return an int* pointer representing the values of the variables
66   */
67  int* return_solution(void* sp){
68      return static_cast<CounterpointProblem*>(sp)->return_solution();
69  }
70
71
72  // ===== NEEDED BY SSP PARSER =====
73
74  int* return_species_array_5sp(void* sp, int ctp_index){
75      return static_cast<CounterpointProblem*>(sp)->get_species_array_5sp(ctp_index);
76  }
77
78  int* return_extended_cp_domain(void* sp, int ctp_index){
79      return static_cast<CounterpointProblem*>(sp)->get_extended_cp_domain(ctp_index);
80  }
81
82  int get_extended_cp_domain_size(void* sp, int ctp_index){
83      return static_cast<CounterpointProblem*>(sp)->get_ext_cp_domain_size(ctp_index);
84  }
85
86  // =====

```

```

87
88
89 /**
90  * creates a search engine for Problem objects
91  * @param sp a void* pointer to a Problem object
92  * @return a void* cast of a Base<Problem>* pointer
93  */
94 void* create_solver(void* sp, int type){
95     return (void*) make_solver(static_cast<CounterpointProblem*>(sp), type); // TODO
96 }
97
98 /**
99  * returns the next solution space, it should be bound. If not, it will return NULL.
100  * @param solver a void* pointer to a Base<CounterpointProblem>* pointer for the search engine
    of the problem
101  * @return a void* cast of a CounterpointProblem* pointer
102  */
103 void* return_next_solution_space(void* solver){
104     writeToLogFile("return_next_solution_space function called");
105     return (void*) get_next_solution_space(static_cast<BAB<CounterpointProblem*>*>(solver));
106 }
107
108
109
110 int search_stopped(void* solver){
111     return static_cast<int>(static_cast<Base<CounterpointProblem*>*>(solver)->stopped());
112 }
113
114
115
116
117
118 // UTILITY FUNCTION JUST FOR PROBLEMWRAPPER
119 // Conversion function from std::vector<int> to std::vector<species>, DOES NOT HANDLE CANTUS
    FIRMUS because it is not supposed to be in sList
120 std::vector<Species> convertToSpeciesVector(const std::vector<int>& intVec) {
121     std::vector<Species> speciesVec;
122     speciesVec.reserve(intVec.size()); // Reserve space to avoid multiple allocations
123
124     for (int value : intVec) {
125         switch (value) {
126             case 1:
127                 speciesVec.push_back(FIRST_SPECIES);
128                 break;
129             case 2:
130                 speciesVec.push_back(SECOND_SPECIES);
131                 break;
132             case 3:
133                 speciesVec.push_back(THIRD_SPECIES);
134                 break;
135             case 4:
136                 speciesVec.push_back(FOURTH_SPECIES);
137                 break;
138             case 5:
139                 speciesVec.push_back(FIFTH_SPECIES);
140                 break;
141             default:
142                 std::cerr << "Warning: Value " << value << " is out of range for species enum\
143 n";
144                 // Handle out of range values if needed
145                 // throw std::out_of_range("Value out of range for species enum");
146                 // speciesVec.push_back(static_cast<species>(0)); // Or handle it
147                 appropriately
148                 break;
149         }
150     }
151     return speciesVec;
152 }

```

C.6.5 Stratum.cpp

```

1 //
2 // Created by Luc Cleenewerk and Diego de Patoul.
3 //
4
5 #include "../headers/Stratum.hpp"
6
7 Stratum::Stratum(Home home, int nMes, int lb, int ub) : Voice(home, nMes, lb, ub){
8     //notes = IntVarArray(home, nMes*4-3, 0, 127);
9     //h_intervals = IntVarArray(home, notes.size(), -PERFECT_OCTAVE, PERFECT_OCTAVE);
10    //m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
11 }
12
13 //if the stratum is in the upper strata, this constructor will create the h_intervals to the
    lowest stratum
14 Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes) : Stratum(home,
    nMes, lb, ub){
15     for(int i = 0; i < h_intervals.size(); i++){
16         rel(home, (h_intervals[i])=((notes[i]-lowestNotes[i])%12));
17     }
18 }
19
20 //this constructor applies 3rd voice specific constraints which apply to the upper strata
21 Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int nV) :
    Stratum(home, nMes, lb, ub, lowestNotes){
22
23     //1.H3 (formerly G8) Last chord can only consist of notes of the harmonic triad
24     dom(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IntSet(IntArgs(TRIAD)));
25
26     //H10 No tenths in last chord
27     rel(home, ((notes[notes.size()-4]-lowestNotes[lowestNotes.size()-1])>12) >> (expr(home,
28         abs(h_intervals[h_intervals.size()-1]))!=MINOR_THIRD &&
29         expr(home, abs(h_intervals[h_intervals.size()-4]))!=MAJOR_THIRD));
30
31     //H12 Last chord cannot include a minor third
32     rel(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IRT_NQ, 3);
33 }
34
35 Stratum::Stratum(Home home, int nMes, int lb, int ub, IntVarArray lowestNotes, int nV1, int
    nV2) : Stratum(home, nMes, lb, ub, lowestNotes){
36     //1.H3 (formerly G8) Last chord can only consist of notes of the harmonic triad
37     dom(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IntSet(IntArgs(TRIAD)));
38
39     //H12 Last chord cannot include a minor third
40     rel(home, expr(home, abs(h_intervals[h_intervals.size()-1])), IRT_NQ, 3);
41 }
42
43
44 string Stratum::to_string() const {
45     string text = "Notes : ";
46     text += intVarArray_to_string(notes);
47     text += "\n";
48     text += "H intervals : ";
49     text += intVarArray_to_string(h_intervals);
50     text += "\n";
51     text += "M intervals : ";
52     text += intVarArray_to_string(m_intervals_brut);
53     text += "\n";
54     return text;
55 }
56
57 // clone constructor
58 Stratum::Stratum(Home home, Stratum &s) : Voice(home, s){
59     // update/clone here all variables that are not in voice
60 }
61
62
63 Stratum* Stratum::clone(Home home){
64     return new Stratum(home, *this);
65 }
66
67 void Stratum::setNote(Home home, int index, IntVar note){

```

```

68     rel(home, notes[index], IRT_EQ, note);
69 }

```

C.6.6 Utilities.cpp

```

1  //
2  // Created by Luc Cleenewerk and Diego de Patoul.
3  // This file contains the implementations of general utility functions.
4  //
5
6  #include "../headers/Utilities.hpp"
7
8  /**
9   * For a given set of intervals between notes that loops and a starting note, returns all the
   possible notes
10  * @param note the starting note
11  * @param intervals the set of intervals between notes. It must make a loop. For example, to
   get all notes from a major
12  * scale from note, use {2, 2, 1, 2, 2, 2, 1}. To get all notes from a minor chord, use {3, 4,
   5}.
13  * @return vector<int> all the notes
14  */
15  vector<int> get_all_notes_from_interval_loop(int n, vector<int> intervals)
16  {
17      int note = n % PERFECT_OCTAVE; // bring the root back to [12,23] in case the argument is
   wrong
18      vector<int> notes;
19
20      int i = 0;
21      while (note <= 127)
22      {
23          notes.push_back(note);
24          note += intervals[i % intervals.size()];
25          ++i;
26      }
27      return notes;
28  }
29
30  /**
31  * For a given tonality (root + mode), returns all the possible notes
32  * @param root the root of the tonality (in [0,11])
33  * @param scale the set of tones and semitones that define the scale
34  * @return vector<int> all the possible notes from that tonality
35  */
36  vector<int> get_all_notes_from_scale(int root, vector<int> scale)
37  {
38      return get_all_notes_from_interval_loop(root, scale);
39  }
40
41  /**
42  * For a given chord (root + mode), returns all the possible notes
43  * @param root the root of the chord
44  * @param quality the set of tones and semitones that define the chord
45  * @return vector<int> all the possible notes from that chord
46  */
47  vector<int> get_all_notes_from_chord(int root, vector<int> quality)
48  {
49      return get_all_notes_from_interval_loop(root, quality);
50  }
51
52  /**
53  * Get all values for a given note
54  * @param note a note
55  * @return vector<int> a vector containing all the given notes
56  */
57  vector<int> get_all_given_note(int note)
58  {
59      int current = note % PERFECT_OCTAVE;
60      vector<int> notes;
61      while (current < 127)
62      {

```

```

63     notes.push_back(current);
64     current += 12;
65 }
66 return notes;
67 }
68
69 /**
70  * Transforms a vector of integers into a string
71  * @param vector a vector of integers
72  * @return string the string representation of the vector
73  */
74 string int_vector_to_string(vector<int> vector){
75     string s;
76     for (int i = 0; i < vector.size(); ++i) {
77         s += to_string(vector[i]);
78         if(i != vector.size() - 1)
79             s += " , ";
80     }
81     return s;
82 }
83
84 /**
85  * Transforms an int* into a vector<int>
86  * @param ptr an int* pointer
87  * @param size the size of the array
88  * @return a vector<int> containing the same values as the array
89  */
90 vector<int> int_pointer_to_vector(int* ptr, int size){
91     vector<int> v;
92     for(int i = 0; i < size; i++){
93         v.push_back(ptr[i]);
94     }
95     return v;
96 }
97
98 /**
99  * Union algorithm found and adapted from GeeksForGeeks.com
100  */
101 vector<int> vector_union(vector<int> v1, vector<int> v2){
102     vector<int> v(v1.size()
103                 + v2.size());
104     vector<int>::iterator it, st;
105
106     it = set_union(v1.begin(), v1.end(),
107                   v2.begin(),
108                   v2.end(), v.begin());
109
110     vector<int> res = {};
111     for(st = v.begin(); st != it; ++st){
112         res.push_back(*st);
113     }
114     return res;
115 }
116
117 /**
118  * Intersection algorithm found and adapted from GeeksForGeeks.com
119  */
120 vector<int> vector_intersection(vector<int> v1, vector<int> v2){
121     vector<int> v(v1.size()
122                 + v2.size());
123     vector<int>::iterator it, st;
124
125     it = set_intersection(v1.begin(), v1.end(),
126                           v2.begin(),
127                           v2.end(), v.begin());
128
129     vector<int> res = {};
130     for(st = v.begin(); st != it; ++st){
131         res.push_back(*st);
132     }
133     return res;
134 }
135

```

```

136 /**
137  * Difference algorithm found and adapted from GeeksForGeeks.com
138  */
139 vector<int> vector_difference(vector<int> v1, int lb, int ub){
140     vector<int> v = {};
141     for(int i = lb; i <= ub; i++){
142         int t = 0;
143         for(int j = 0; j < v1.size(); j++){
144             if(i==v1[j]){
145                 t=1;
146                 break;
147             }
148         }
149         if(t==0){
150             v.push_back(i);
151         }
152     }
153     return v;
154 }
155
156 /**
157  * Prints the Search::Statistics object into a readable format
158  * @param stats a Search::Statistics object representing the statistics of a search
159  * @return The string representation of the statistics object
160  */
161 string statistics_to_string(Search::Statistics stats){
162     string s = "Nodes traversed: " + to_string(stats.node) + "\n";
163     s += "Failed nodes explored: " + to_string(stats.fail) + "\n";
164     s += "Restarts performed: " + to_string(stats.restart) + "\n";
165     s += "Propagators executed: " + to_string(stats.propagate) + "\n";
166     s += "No goods generated: " + to_string(stats.nogood) + "\n";
167     s += "Maximal depth of explored tree: " + to_string(stats.depth) + "\n";
168     return s;
169 }
170
171 /**
172  * Prints the Search::Statistics object into a csv format (coma separated)
173  * @param stats a Search::Statistics object representing the statistics of a search
174  * @return The string representation of the statistics object
175  */
176 string statistics_to_csv_string(Search::Statistics stats){
177     string s = to_string(stats.node) + ",";
178     s += to_string(stats.fail) + ",";
179     s += to_string(stats.restart) + ",";
180     s += to_string(stats.propagate) + ",";
181     s += to_string(stats.nogood) + ",";
182     s += to_string(stats.depth) + ",";
183     return s;
184 }
185
186 /**
187  * Returns the value of a variable as a string
188  * @param var an integer variable
189  * @param absolute a boolean indicating if the value should be returned as an absolute value (
190     default is false)
191  * @return a string representing the value of the variable
192  */
193 string intVar_to_string(const IntVar &var, bool absolute) {
194     if (var.assigned()){
195         if(absolute)
196             return to_string(abs(var.val()));
197         return to_string(var.val());
198     }
199     return "<not assigned>";
200 }
201
202 /**
203  * Returns the values of an array of variables as a string
204  * @param vars an array of integer variables
205  * @return a string representing the values of the variables
206  */
207 string intVarArray_to_string(IntVarArray vars){
208     int s = vars.size();

```

```

208     string res = "{";
209     for(int i = 0; i < s; i++){
210         res += intVar_to_string(vars[i]);
211         if(i != s - 1)
212             res += ", ";
213     }
214     res += "}";
215     return res;
216 }
217
218 /**
219  * Returns the value of a variable as a string
220  * @param var an integer variable
221  * @param absolute a boolean indicating if the value should be returned as an absolute value (
222     default is false)
223  * @return a string representing the value of the variable
224  */
225 string boolVar_to_string(const BoolVar &var, bool absolute) {
226     if (var.assigned()){
227         if(absolute)
228             return to_string(abs(var.val()));
229         return to_string(var.val());
230     }
231     return "<not assigned>";
232 }
233
234 /**
235  * Returns the values of an array of variables as a string
236  * @param vars an array of integer variables
237  * @return a string representing the values of the variables
238  */
239 string boolVarArray_to_string(BoolVarArray vars){
240     int s = vars.size();
241     string res = "{";
242     for(int i = 0; i < s; i++){
243         res += boolVar_to_string(vars[i]);
244         if(i != s - 1)
245             res += ", ";
246     }
247     res += "}";
248     return res;
249 }
250
251 /**
252  * Returns the values of an array of variables as a string
253  * @param vars an array of integer variables
254  * @return a string representing the values of the variables, ignoring unassigned variables
255  */
256 string cleanIntVarArray_to_string(IntVarArray vars){
257     int s = vars.size();
258     string res = "{";
259     for(int i = 0; i < s; i++){
260         if (vars[i].assigned()){
261             res += intVar_to_string(vars[i]);
262             if(i != s - 1)
263                 res += ", ";
264         }
265     }
266     res += "}";
267     return res;
268 }
269
270 /**
271  * Returns the values of an IntVarArgs as a string
272  * @param args an IntVarArgs
273  * @return a string representing the values
274  */
275 string intVarArgs_to_string(IntVarArgs args){
276     int s = args.size();
277     string res;
278     for(int i = 0; i < s; i++){
279         res += intVar_to_string(args[i], 0);
280         if(i != s - 1)

```

```

280         res += ", ";
281     }
282     return res;
283 }
284
285
286
287 /**
288  * Returns the name of a note based on its MIDI value
289  * @param note an integer
290  */
291 string midi_to_letter(int note){
292     return noteNames[note % PERFECT_OCTAVE];
293 }
294
295 /**
296  * Returns the name of a mode based on its integer value
297  * @param mode an integer
298  * @return a string representing the name of the mode
299  */
300 string mode_int_to_name(int mode){
301     return modeNames[mode];
302 }
303
304 /**
305  * returns a string with the time
306  * @return a string with the time
307  */
308 string time(){
309     /// date and time for logs
310     std::time_t currentTime = std::time(nullptr); // Get the current time
311     std::string timeString = std::asctime(std::localtime(&currentTime)); // Convert to string
312     return timeString;
313 }
314
315 /**
316  * Write a text into a log file
317  * Useful for debugging in the OM environment
318  * @param message the text to write
319  */
320 void write_to_log_file(const char *message, const string& filename) { // TODO : TO ACTIVATE
    LOGGING, UNCOMMENT THE CONTENTS OF THIS FUNCTION AND THE NEXT FUNCTION.
321     // const char* homeDir = std::getenv("HOME"); // Get the user's home directory
322     // if (homeDir) {
323     //     std::string filePath(homeDir);
324     //     filePath += "/Documents/Libraries/MusicConstraints/out/" + filename; // Specify the
        desired file path, such as $HOME/log.txt
325
326     //     std::ofstream myfile(filePath, std::ios::app); // append mode
327     //     if (myfile.is_open()) {
328     //         myfile << message << endl;
329     //         myfile.close();
330     //     }
331     // }
332 }
333
334 /**
335  * Write a text into a log file
336  * @param message the text to write
337  */
338 void writeToLogFile(const char* message){ // TODO : TO ACTIVATE LOGGING, UNCOMMENT THE
    CONTENTS OF THIS FUNCTION AND THE PREVIOUS FUNCTION.
339     // std::time_t currentTime = std::time(nullptr); // Get the current time
340     // std::string timeString = std::asctime(std::localtime(&currentTime)); // Convert to
        string
341
342     // const char* homeDir = std::getenv("HOME"); // Get the user's home directory
343     // if (homeDir) {
344     //     std::string filePath(homeDir);
345     //     filePath += "/log.txt"; // Specify the desired file path, such as $HOME/log.txt
346
347     //     std::ofstream myfile(filePath, std::ios::app); // append mode
348     //     if (myfile.is_open()) {

```

```

349 //      myfile <<timeString<< endl << message << endl;
350 //      myfile.close();
351 //      }
352 // }
353 }

```

C.6.7 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
16     //h_interval, since it is only calculated using the first note
17     //(so 2nd, 3rd or 4th note couldn't be smaller than 0)
18     h_intervals = IntVarArray(home, size-3, -PERFECT_OCTAVE, PERFECT_OCTAVE);
19     m_intervals_brut = IntVarArray(home, notes.size()-1, -PERFECT_OCTAVE, PERFECT_OCTAVE);
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::getHInterval(){
56     return h_intervals;
57 }
58
59 IntVarArgs Voice::getSecondHInterval(){
60     return h_intervals.slice(2,4,h_intervals.size());
61 }

```

C.7 Lisp files

C.7.1 interface.lisp

```
1 (in-package :fuxcp)
2
3 ; Author: Thibault Wafflard and Anton Lamotte. Adapted for the migration of FuxCP to C++ by
4 ; Luc Cleenewerk and Diego de Patoul in August 2024.
5 ; Date: June 3, 2023 and January 2024
6 ; This file contains all the cp-params interface.
7 ; That is to say the interface blocks, as well as the global variables updated via the
8 ; interface.
9
10 ;;=====
11 ;;= cp-params OBJECT =
12 ;;=====
13
14 (print "Loading cp-params object...")
15
16 (defparameter DFS 0)
17 (defparameter BAB 1)
18
19 (om::defclass! cp-params ()
20   ;attributes
21   (
22     ; ----- Input cantus firmus -----
23     (cf-voice :accessor cf-voice :initarg :cf-voice :initform nil :documentation "")
24     ; ----- Solver parameters -----
25     (species-param :accessor species-param :initform (list "1st" "1st" "1st") :type string :
26       documentation "")
27     (voice-type-param :accessor voice-type-param :initform (list "Really far above" "Above" "
28       Below") :type string :documentation "")
29     (min-skips-slider-param :accessor min-skips-slider-param :initform 0 :type integer :
30       documentation "")
31     (borrow-mode-param :accessor borrow-mode-param :initform "Major" :type string :
32       documentation "")
33     ; ----- Output & Stop -----
34     (current-csp :accessor current-csp :initform nil :documentation "")
35     (result-voice :accessor result-voice :initarg :result-voice :initform nil :documentation "
36       ")
37     ; ----- Cost order -----
38     (linear-combination :accessor linear-combination :initform "Linear combination" :type
39       string :documentation "")
40   )
41   (:icon 225)
42   (:documentation "This class implements FuxCP.
43     FuxCP is a constraints programming based tool aiming to generate counterpoints based on
44     cantus firmus.")
45 )
46
47 ; the editor for the object
48 (defclass params-editor (om::editorview) ())
49
50 (defmethod om::class-has-editor-p ((self cp-params)) t)
51 (defmethod om::get-editor-class ((self cp-params)) 'params-editor)
52
53 (defmethod om::om-draw-contents ((view params-editor))
54   (let* ((object (om::object view)))
55     (om::om-with-focused-view view)
56   )
57 )
58
59 ; this function creates the elements for the main panel
60 (defun make-main-view (editor)
61   ; background colour
62   (om::om-set-bg-color editor om::*om-light-gray-color*)
63 )
64
65 ; To access the melodizer object, (om::object self)
66 (defmethod initialize-instance ((self params-editor) &rest args)
67   ;; do what needs to be done by default
```

```

60 (call-next-method) ; start the search by default?
61 (make-interface self)
62 )
63
64 (defun make-interface (editor)
65   (let* (
66     (melodic-subcosts '(
67       (:name "Steps" :value "No cost" :cannot-be-forbidden t :param m-step-cost)
68       (:name "Third skips" :value "Low cost" :param m-third-cost)
69       (:name "Fourth leaps" :value "Low cost" :param m-fourth-cost)
70       (:name "Tritone leaps" :value "Forbidden" :param m-tritone-cost)
71       (:name "Fifth leaps" :value "Medium cost" :param m-fifth-cost)
72       (:name "Sixth leaps" :value "Medium cost" :param m-sixth-cost)
73       (:name "Seventh leaps" :value "Medium cost" :param m-seventh-cost)
74       (:name "Octave leaps" :value "Low cost" :param m-octave-cost)
75     ))
76
77     (melodic-preferences '( ; care it is a special apostrophe here (needed to evaluate
every value that has a comma in this list, and not to take their symbols)
78       (:section "Melodic Preferences" :name "Melodic cost" :display nil :importance "13"
:value nil :subcosts ,melodic-subcosts :param m-degrees-cost)
79       ;; Add more cost data as needed
80     ))
81
82     (motion-subcosts '(
83       (:name "Direct motion" :value "Medium cost" :cannot-be-forbidden t :param
dir-motion-cost)
84       (:name "Oblique motion" :value "Low cost" :param obl-motion-cost)
85       (:name "Contrary motion" :value "No cost" :cannot-be-forbidden t :param
con-motion-cost)
86     ))
87
88     (general-preferences '( ; care it is a special apostrophe here (needed to evaluate
every value that has a comma in this list, and not to take their symbols)
89       (:section "General preferences" :name "Borrowed notes" :display nil :importance "8"
:value "High cost" :param borrow-cost :cannot-be-forbidden t)
90       (:section "General preferences" :name "Harmonic fifths on the downbeat" :display
nil :importance "7" :value "Low cost" :param h-fifth-cost :cannot-be-forbidden t)
91       (:section "General preferences" :name "Harmonic octaves on the downbeat" :display
nil :importance "5" :value "Low cost" :param h-octave-cost :cannot-be-forbidden t)
92       (:section "General preferences" :name "Successive perfect consonances" :display
nil :importance "2" :value "Medium cost" :param succ-p-cons-cost :cannot-be-forbidden t)
93       (:section "General preferences" :name "Repeating notes" :display nil :importance "
9" :value "Medium cost" :param variety-cost)
94       (:section "General preferences" :name "No harmonic triad" :display nil :importance
"3" :value "Medium cost" :param h-triad-cost :cannot-be-forbidden t)
95       (:section "General preferences" :name "Direct motion to perf. consonance" :display
nil :importance "14" :value "Last resort" :param direct-move-to-p-cons-cost :
cannot-be-forbidden t)
96       (:section "General preferences" :name "Motion cost" :display nil :importance "12"
:value nil :subcosts ,motion-subcosts :param motions-cost)
97       (:section "General preferences" :name "Apply specific penultimate note rules" :
value "Yes" :special-range ("Yes" "No") :param penult-rule-check)
98
99       ;; Add more cost data as needed
100     ))
101
102     (specific-preferences '( ; care it is a special apostrophe here (needed to evaluate
every value that has a comma in this list, and not to take their symbols)
103       (:section "Second species specific pref." :name "Penultimate downbeat note is a
fifth" :importance "6" :value "Last resort" :param penult-sixth-cost :cannot-be-forbidden
t)
104       (:section "Third species specific pref." :name "No cambiata" :importance "11" :
value "High cost" :param non-cambiata-cost :cannot-be-forbidden t)
105       (:section "Third species specific pref." :name "Force contrary motion after skip"
:value "No" :special-range ("Yes" "No") :param con-m-after-skip-check)
106       (:section "Third species specific pref." :name "No hamonic triad in 2nd/3rd beat"
:display nil :importance "4" :value "Medium cost" :param h-triad-3rd-species-cost)
107       (:section "Thirld and fourth species specific pref." :name "Same note in downbeat
and upbeat" :importance "10" :value "Low cost" :param m2-eq-zero-cost)
108       (:section "Fourth species specific pref." :name "No ligatures" :importance "1" :
value "Last resort" :param no-syncopation-cost :cannot-be-forbidden t)

```

```

109     (:section "Fifth species specific pref." :name "Many quarters (left) or many
syncopations (right)" :value 50 :make-slider t :param pref-species-slider)
110     ;; Add more cost data as needed
111     ))
112     )
113     ;; Add the cost table to the main view
114     (om::om-add-subviews editor (make-cost-panel editor general-preferences #|x-offset:|#
0 #|y-offset:|# 0 #|size:|# 540 #|colour:|# om::*azulote*))
115     (om::om-add-subviews editor (make-cost-panel editor melodic-preferences #|x-offset:|#
526 #|y-offset:|# 0 #|size:|# 350 #|colour:|# om::*azulito*))
116     (om::om-add-subviews editor (make-cost-panel editor specific-preferences #|x-offset:|#
1052 #|y-offset:|# 0 #|size:|# 500 #|colour:|# (om::make-color-255 230 190 165)))
117     (om::om-add-subviews editor (make-explanation-panel editor #|x-offset:|#
0 #|y-offset:|# 541 #|size:|# 160 #|colour:|# (om::make-color-255 255 240 120)))
118     (om::om-add-subviews editor (make-search-params-panel editor #|x-offset:|# 526
#|y-offset:|# 351 #|size:|# 350 #|colour:|# om::*maq-color*))
119     (om::om-add-subviews editor (make-search-buttons editor #|x-offset:|# 1052
#|y-offset:|# 501 #|size:|# 200 #|colour:|# om::*workspace-color* melodic-subcosts
melodic-preferences motion-subcosts general-preferences specific-preferences))
120
121     )
122
123     ;; ... (existing code)
124
125
126     editor ; Return the editor
127 )
128
129 (defun make-explanation-panel (editor panel-x-offset panel-y-offset size colour)
130     (let* (
131         ;; Explanation text
132         (explanation-text "First choose the importance of each preference (1 being the most
important and 14 being the least important). The solver will give priority to the most
important preferences. The cost value is taken into account if two costs have the same
importance. ")
133
134         ;; Create a view for the explanation panel
135         (explanation-panel (om::om-make-view 'om::om-view
:size (om::om-make-point 525 size)
:position (om::om-make-point panel-x-offset panel-y-offset)
:bg-color colour))
136
137         ;; Create a text element for the explanation
138         (explanation-label (om::om-make-dialog-item 'om::om-static-text
(om::om-make-point 10 10) (om::om-make-point 500 800)
explanation-text
))
139
140         )
141
142     ;; Add the text element to the explanation panel
143     (om::om-add-subviews explanation-panel explanation-label)
144
145     (om::om-add-subviews explanation-panel
146         (om::om-make-dialog-item
147             'om::om-static-text
148             (om::om-make-point 10 100)
149             (om::om-make-point 400 20)
150             "If two costs are ranked the same, perform between them a:"
151             )
152         (om::om-make-dialog-item
153             'om::pop-up-menu
154             (om::om-make-point 215 120)
155             (om::om-make-point 280 20)
156             "Linear combination"
157             :range (list "Linear combination" "Maximum minimisation")
158             :value (linear-combination (om::object editor))
159             :di-action #'(lambda (cost)
160                 (setf (linear-combination (om::object editor)) (nth (om::
om-get-selected-item-index cost) (om::om-get-item-list cost))))
161             )
162         )
163     )
164 )
165

```

```

170 ;; Add the explanation panel to the main view
171 explanation-panel
172 )
173 )
174
175 (defun make-cost-panel (editor cost-data panel-x-offset panel-y-offset y-size colour)
176   (let* (
177     (cost-table (om::om-make-view 'om::om-view
178       :size (om::om-make-point 525 y-size)
179       :position (om::om-make-point panel-x-offset panel-y-offset)
180       :bg-color colour))
181
182     (importance-column (om::om-make-dialog-item 'om::om-static-text
183       (om::om-make-point 275 0) (om::om-make-point 150 20) "
184       Importance"
185       :font om::*om-default-font2b*
186     ))
187
188     (value-column (om::om-make-dialog-item 'om::om-static-text
189       (om::om-make-point 400 0) (om::om-make-point 150 20) "Value"
190       :font om::*om-default-font2b*
191     ))
192   )
193   ;; Add header columns to the cost table
194   (om::om-add-subviews cost-table importance-column value-column)
195
196   ;; Populate the cost data dynamically
197   (let (
198     (current-section nil)
199     (y-offset 0)
200   )
201     (loop for index from 0 below (length cost-data)
202       do
203         (let* ((cost (nth index cost-data))
204           (section (getf cost :section))
205           (y-position (+ y-offset (* 45 index)))
206           (name (if (getf cost :display) (getf cost :display) (getf cost :name)))
207           (importance (getf cost :importance))
208           (value (getf cost :value))
209           (is-new-section (not (string= current-section section))))
210           )
211
212           ;; Add subsection header if it's a new section
213           (when is-new-section
214             (let ((section-label (om::om-make-dialog-item 'om::om-static-text
215               (om::om-make-point 15 y-position) (om::om-make-point
216                 300 20) section-label
217               :font om::*om-default-font2b*)))
218               (om::om-add-subviews cost-table section-label))
219               (setf current-section section)
220               (incf y-offset 35)
221               (incf y-position 35)
222             )
223             ;; Add the row to the cost table
224             (let* (
225               (name-label (om::om-make-dialog-item 'om::om-static-text
226                 (om::om-make-point 25 y-position) (om::om-make-point 500 20)
227                 name))
228               (importance-popup (om::om-make-dialog-item 'om::pop-up-menu
229                 (om::om-make-point 275 (- y-position 7)) (om::
230                   om-make-point 70 20)
231                 (format nil "~A" importance)
232                 :value importance
233                 :range (importance-range)
234                 :di-action #'(lambda (x)
235                   (setf (getf cost :importance) (nth (om::
236                     om-get-selected-item-index x) (om::om-get-item-list x)))
237                   (print (getf cost :importance))
238                 )
239               )
240             )
241             (value-popup (if (getf cost :make-slider)
242

```

```

238         (make-slider cost y-position)
239         (om::om-make-dialog-item 'om::pop-up-menu
240         (om::om-make-point 345 (- y-position 7)) (om::
om-make-point 150 20)
241         (format nil "~A" value)
242         :value value
243         :range (if (getf cost :special-range)
244         (getf cost :special-range)
245         (value-range (getf cost :cannot-be-forbidden))
246         )
247         :di-action #'(lambda (x)
248         (setf (getf cost :value) (nth (om::
om-get-selected-item-index x) (om::om-get-item-list x)))
249         (print (getf cost :value))
250         )
251         )
252     ))
253 )
254 (cond
255 ((and value importance) (om::om-add-subviews cost-table name-label
importance-popup value-popup))
256 (value (om::om-add-subviews cost-table name-label value-popup))
257 (importance (om::om-add-subviews cost-table name-label
importance-popup))
258 )
259 ) ; end of row
260 (print (getf cost :subcosts))
261 (print (length (getf cost :subcosts)))
262 (if (getf cost :subcosts)
263     (loop for index from 0 below (length (getf cost :subcosts))
264     do
265         (let* ((cost (nth index (getf cost :subcosts)))
266         (y-position (+ y-position (* 35 (+ 1 index))))
267         (name (concatenate 'string "|---" (getf cost :name)))
268         (value (getf cost :value))
269         )
270         (incf y-offset 35)
271         ;; Add the row to the cost table
272         (let* (
273         (name-label (om::om-make-dialog-item 'om::om-static-text
274         (om::om-make-point 50 y-position) (om::
om-make-point 250 20) name))
275         (value-popup (om::om-make-dialog-item 'om::pop-up-menu
276         (om::om-make-point 345 (- y-position 7)) (om::
om-make-point 150 20)
277         (format nil "~A" value)
278         :value value
279         :range (value-range (getf cost :
cannot-be-forbidden))
280         :di-action #'(lambda (x)
281         (setf (getf cost :value) (nth (om::
om-get-selected-item-index x) (om::om-get-item-list x)))
282         (print (getf cost :value))
283         )
284         )
285         )
286         )
287         (om::om-add-subviews cost-table name-label value-popup)
288         ) ; end of subcost row
289         ) ; end of subcost
290         ) ; end of subcost loop
291     )
292     ) ; end of cost
293     ) ; end of loop
294 )
295 cost-table
296 ))
297
298 (defun make-search-params-panel (editor panel-x-offset panel-y-offset y-size colour)
299     (let* (
300         (search-params-panel (om::om-make-view 'om::om-view
301         :size (om::om-make-point 525 y-size)
302         :position (om::om-make-point panel-x-offset panel-y-offset)

```

```

303         :bg-color colour))
304     )
305     (om::om-add-subviews
306       search-params-panel
307       (om::om-make-dialog-item
308         'om::om-static-text
309         (om::om-make-point 15 0)
310         (om::om-make-point 200 20)
311         "Solver Configuration"
312         :font om::*om-default-font2b*
313       )
314
315       (om::om-make-dialog-item
316         'om::om-static-text
317         (om::om-make-point 25 30)
318         (om::om-make-point 150 20)
319         "First voice species"
320       )
321
322       (om::om-make-dialog-item
323         'om::pop-up-menu
324         (om::om-make-point 275 25)
325         (om::om-make-point 220 20)
326         "First voice species"
327         :range (list "1st" "2nd" "3rd" "4th" "5th")
328         :value (first (species-param (om::object editor)))
329         :di-action #'(lambda (cost)
330           (setf (first (species-param (om::object editor))) (nth (om::
331             om-get-selected-item-index cost) (om::om-get-item-list cost)))
332         )
333       )
334
335       (om::om-make-dialog-item
336         'om::om-static-text
337         (om::om-make-point 25 70)
338         (om::om-make-point 150 20)
339         "First voice range"
340       )
341
342       (om::om-make-dialog-item
343         'om::pop-up-menu
344         (om::om-make-point 275 65)
345         (om::om-make-point 220 20)
346         "Voice range"
347         :range (list "Really far above" "Far above" "Above" "Same range" "Below" "Far
348         below" "Really far below")
349         :value (first (voice-type-param (om::object editor)))
350         :di-action #'(lambda (cost)
351           (setf (first (voice-type-param (om::object editor))) (nth (om::
352             om-get-selected-item-index cost) (om::om-get-item-list cost)))
353         )
354       )
355
356       (om::om-make-dialog-item
357         'om::om-static-text
358         (om::om-make-point 25 110)
359         (om::om-make-point 150 20)
360         "Second voice species"
361       )
362
363       (om::om-make-dialog-item
364         'om::pop-up-menu
365         (om::om-make-point 275 105)
366         (om::om-make-point 220 20)
367         "Second voice species"
368         :range (list "None" "1st" "2nd" "3rd" "4th" "5th")
369         :value (second (species-param (om::object editor)))
370         :di-action #'(lambda (cost)
371           (setf (second (species-param (om::object editor))) (nth (om::

```

```

372 (om::om-make-dialog-item
373 'om::om-static-text
374 (om::om-make-point 25 150)
375 (om::om-make-point 150 20)
376 "Second voice range"
377 )
378
379 (om::om-make-dialog-item
380 'om::pop-up-menu
381 (om::om-make-point 275 145)
382 (om::om-make-point 220 20)
383 "Second voice range"
384 :range (list "Really far above" "Far above" "Above" "Same range" "Below" "Far
below" "Really far below")
385 :value (second (voice-type-param (om::object editor)))
386 :di-action #'(lambda (cost)
387 (setf (second (voice-type-param (om::object editor))) (nth (om::
om-get-selected-item-index cost) (om::om-get-item-list cost))))
388 )
389 )
390
391
392
393 (om::om-make-dialog-item
394 'om::om-static-text
395 (om::om-make-point 25 190)
396 (om::om-make-point 150 20)
397 "Third voice species"
398 )
399
400 (om::om-make-dialog-item
401 'om::pop-up-menu
402 (om::om-make-point 275 185)
403 (om::om-make-point 220 20)
404 "Third voice species"
405 :range (list "None" "1st" "2nd" "3rd" "4th" "5th")
406 :value (third (species-param (om::object editor)))
407 :di-action #'(lambda (cost)
408 (setf (third (species-param (om::object editor))) (nth (om::
om-get-selected-item-index cost) (om::om-get-item-list cost))))
409 )
410 )
411
412 (om::om-make-dialog-item
413 'om::om-static-text
414 (om::om-make-point 25 230)
415 (om::om-make-point 150 20)
416 "Third voice range"
417 )
418
419 (om::om-make-dialog-item
420 'om::pop-up-menu
421 (om::om-make-point 275 225)
422 (om::om-make-point 220 20)
423 "Third voice range"
424 :range (list "Really far above" "Far above" "Above" "Same range" "Below" "Far
below" "Really far below")
425 :value (third (voice-type-param (om::object editor)))
426 :di-action #'(lambda (cost)
427 (setf (third (voice-type-param (om::object editor))) (nth (om::
om-get-selected-item-index cost) (om::om-get-item-list cost))))
428 )
429 )
430
431
432 (om::om-make-dialog-item
433 'om::om-static-text
434 (om::om-make-point 25 270)
435 (om::om-make-point 150 20)
436 "Borrowing mode"
437 )
438
439 (om::om-make-dialog-item

```

```

440         'om::pop-up-menu
441         (om::om-make-point 275 265)
442         (om::om-make-point 220 20)
443         "Borrowing mode"
444         :range (list "None" "Major" "Minor")
445         :value (borrow-mode-param (om::object editor))
446         :di-action #'(lambda (cost)
447           (setf (borrow-mode-param (om::object editor)) (nth (om::
om-get-selected-item-index cost) (om::om-get-item-list cost))))
448       )
449     )
450
451     (om::om-make-dialog-item
452     'om::om-static-text
453     (om::om-make-point 25 310)
454     (om::om-make-point 150 20)
455     "Minimum % of skips"
456     )
457
458     (om::om-make-dialog-item
459     'om::om-slider
460     (om::om-make-point 275 305)
461     (om::om-make-point 220 20)
462     "Minimum % of skips"
463     :range '(0 100)
464     :increment 1
465     :value (min-skips-slider-param (om::object editor))
466     :di-action #'(lambda (s)
467       (setf (min-skips-slider-param (om::object editor)) (om::om-slider-value s))
468     )
469     )
470   )
471   search-params-panel
472 )
473 )
474
475 (defun make-search-buttons (editor panel-x-offset panel-y-offset y-size colour
melodic-subcosts melodic-preferences motion-subcosts general-preferences
specific-preferences)
476   (let* (
477     (search-buttons (om::om-make-view 'om::om-view
478       :size (om::om-make-point 525 y-size)
479       :position (om::om-make-point panel-x-offset panel-y-offset)
480       :bg-color colour))
481     )
482     (om::om-add-subviews
483     search-buttons
484     (om::om-make-dialog-item
485     'om::om-static-text
486     (om::om-make-point 187 25)
487     (om::om-make-point 150 20)
488     "Solver Launcher"
489     :font om::*om-default-font3b*
490     )
491
492     (om::om-make-dialog-item
493     'om::om-button
494     (om::om-make-point 97 60) ; position (horizontal, vertical)
495     (om::om-make-point 160 20) ; size (horizontal, vertical)
496     "Save Config"
497     :di-action #'(lambda (b)
498       (if (null (cf-voice (om::object editor))) ; if the problem is not initialized
499         (error "No voice has been given to the solver. Please set a cantus firmus
into the second input and try again.")
500       )
501
502       (if (not (null (current-csp (om::object editor)))) (delete-solver-pointer (
current-csp (om::object editor)))) ; delete the pointer to the current-csp if it is
already initialized
503
504       (set-global-cf-variables
505       (cf-voice (om::object editor))
506

```

```

507         (borrow-mode-param (om::object editor))
508     )
509
510     (defparameter *is-stopped nil)
511
512     (defvar melodic-params-list nil)
513     (defvar melodic-values-list nil)
514     (setq melodic-params-list nil)
515     (setq melodic-values-list nil)
516
517     (dolist (subcost melodic-subcosts)
518         (setq melodic-params-list (append melodic-params-list (list (getf subcost
:param))))
519         (setq melodic-values-list (append melodic-values-list (list (
convert-to-cost-integer (getf subcost :value))))))
520
521     ;; (print "melodic lists params then values: ")
522     ;; (print melodic-params-list)
523     ;; (print melodic-values-list)
524     ;; (print (length melodic-values-list))
525
526     (defvar general-params-list nil)
527     (defvar general-values-list nil)
528     (setq general-params-list nil)
529     (setq general-values-list nil)
530
531     (dolist (cost general-preferences)
532         (if (equal (getf cost :param) 'motions-cost)
533             nil ; motions-cost is treated by the subcosts
534             (if (equal (getf cost :param) 'penult-rule-check)
535                 (progn ; Use progn to group multiple expressions
536                     (setq general-params-list (append general-params-list (list (
getf cost :param))))
537                     (setq general-values-list (append general-values-list (list (
convertparam-yes-no (getf cost :value)))))) ; penult-rule-check is a yes no
538                 (progn ; Use progn to group multiple expressions
539                     (setq general-params-list (append general-params-list (list (
getf cost :param))))
540                     (setq general-values-list (append general-values-list (list (
convert-to-cost-integer (getf cost :value)))))) ; else
541
542             ;; (print "general lists params then values: ")
543             ;; (print general-params-list)
544             ;; (print general-values-list)
545             ;; (print (length general-values-list))
546
547     (defvar motion-params-list nil)
548     (defvar motion-values-list nil)
549     (setq motion-params-list nil)
550     (setq motion-values-list nil)
551
552     (dolist (subcost motion-subcosts)
553         (setq motion-params-list (append motion-params-list (list (getf subcost :
param))))
554         (setq motion-values-list (append motion-values-list (list (
convert-to-cost-integer (getf subcost :value))))))
555
556     ;; (print "motion lists params then values: ")
557     ;; (print motion-params-list)
558     ;; (print motion-values-list)
559     ;; (print (length motion-values-list))
560
561     (defvar specific-params-list nil)
562     (defvar specific-values-list nil)
563     (setq specific-params-list nil)
564     (setq specific-values-list nil)
565
566     (dolist (cost specific-preferences)
567         (if (equal (getf cost :param) 'pref-species-slider)
568             (progn ; Use progn to group multiple expressions

```

```

572         (setq specific-params-list (append specific-params-list (list (
getf cost :param))))
573         (setq specific-values-list (append specific-values-list (list (
getf cost :value)))) ; slider
574         (if (equal (getf cost :param) 'con-m-after-skip-check)
575             (progn ; Use progn to group multiple expressions
576                 (setq specific-params-list (append specific-params-list (list
(getf cost :param))))
577                 (setq specific-values-list (append specific-values-list (list
(convertparam-yes-no (getf cost :value)))))) ; con-m-after-skip-check is a yes no
578                 (progn ; Use progn to group multiple expressions
579                     (setq specific-params-list (append specific-params-list (list
(getf cost :param))))
580                     (setq specific-values-list (append specific-values-list (list
(convert-to-cost-integer (getf cost :value))))))) ; else
581
582         ;; (print "specific lists params then values: ")
583         ;; (print specific-params-list)
584         ;; (print specific-values-list)
585         ;; (print (length specific-values-list))
586
587
588
589         (defvar cost-params-list nil)
590         (defvar cost-values-list nil)
591         (setq cost-params-list nil)
592         (setq cost-values-list nil)
593
594         ;; preferences for the cost order
595         (defparameter *cost-preferences* (make-hash-table))
596         (dolist (current-list (list general-preferences specific-preferences
melodic-preferences))
597             (cost current-list)
598             (if (getf cost :importance)
599                 (progn ;
600                     (setq cost-params-list (append cost-params-list (list (getf
cost :name))))
601                     (setq cost-values-list (append cost-values-list (list (
parse-integer (getf cost :importance))))))
602                 )
603             )
604         )
605
606         ;; (print "cost name then values: ")
607         ;; (print cost-params-list)
608         ;; (print cost-values-list)
609         ;; (print (length cost-values-list))
610
611
612         (setf borrow-mode-int (map-mode-to-int (borrow-mode-param (om::object editor)
))
613         (setf min-skips-slider (min-skips-slider-param (om::object editor)))
614
615         ;; (print (borrow-mode-param (om::object editor)))
616         ;; (print (map-mode-to-int (borrow-mode-param (om::object editor))))
617         ;; (print borrow-mode-int)
618         ;; (print min-skips-slider)
619
620         (setf species-integer-list (convert-to-species-integer-list (species-param (om
::object editor))))
621         (setf *voices-types (convert-to-voice-integer-list (voice-type-param (om::
object editor))))
622
623         ;; (print species-integer-list)
624         ;; (print *voices-types)
625
626         (defparameter *species-list species-integer-list)
627
628         ;; (print *scale)
629
630         (print "calling new problem")
631

```

```

632 (defparameter problem-pointer (new-ctp-problem *cf species-integer-list *
        voices-types borrow-mode-int min-skips-slider general-values-list motion-values-list
        melodic-values-list specific-values-list cost-values-list *tonalite-offset *scale *
        chromatic-scale *borrowed-scale))
633 ;; (print problem-pointer)
634
635 (setf (current-csp (om::object editor)) (create-solver problem-pointer BAB))
636 ;; (print (current-csp (om::object editor)))
637
638 (print "Base<Problem>* (search engine) stored in current-csp object variable")
639
640 (delete-pointer problem-pointer)
641
642 (print "problem-pointer deleted")
643
644 ;; (setf (current-csp (om::object editor)) (fux-cp species-integer-list)) ;
        TODO : REPLACE BY CALL TO GECODE
645 )
646 )
647
648 (om::om-make-dialog-item
649 'om::om-button
650 (om::om-make-point 97 100) ; position
651 (om::om-make-point 160 20) ; size
652 "Next Solution"
653 :di-action #'(lambda (b)
654 (if (typep (current-csp (om::object editor)) 'null); if the problem is not
        initialized
655 (error "The problem has not been initialized. Please set the input and
        press Start.")
656 )
657 (print "Searching for the next solution")
658 ;reset the boolean because we want to continue the search
659 (setq *is-stopped nil)
660 ;get the next solution
661 (mp:process-run-function ; start a new thread for the execution of the next
        method
662 "solver-thread" ; name of the thread, not necessary but useful for
        debugging
663 nil ; process initialization keywords, not needed here
664 (lambda () ; function to call
665 (setf
666 (result-voice (om::object editor))
667 (search-next-fux-cp (current-csp (om::object editor))))
668 )
669 (om::openeditorframe ; open a voice window displaying the solution
670 (om::omNG-make-new-instance (result-voice (om::object editor)) "
        Current solution")
671 )
672 )
673 )
674 )
675 )
676
677 (om::om-make-dialog-item
678 'om::om-button
679 (om::om-make-point 262 100) ; position
680 (om::om-make-point 160 20) ; size
681 "Best Solution"
682 :di-action #'(lambda (b)
683 (if (typep (current-csp (om::object editor)) 'null); if the problem is not
        initialized
684 (error "The problem has not been initialized. Please set the input and
        press Start.")
685 )
686 (print "Searching for the best solution")
687 ;reset the boolean because we want to continue the search
688 (setq *is-stopped nil)
689 ;get the next solution
690 (mp:process-run-function ; start a new thread for the execution of the next
        method
691 "solver-thread" ; name of the thread, not necessary but useful for
        debugging

```

```

692         nil ; process initialization keywords, not needed here
693         (lambda () ; function to call
694           (let ((check 1) (result nil))
695             (loop while check do
696               (setf result (search-next-fux-cp (current-csp (om::object
editor))))))
697               (if result (setf (result-voice (om::object editor)) result) (
setf check nil))
698               (if *is-stopped (setf check nil))
699             )
700           )
701           (om::openeditorframe ; open a voice window displaying the solution
702             (om::omNG-make-new-instance (result-voice (om::object editor)) "
Current solution")
703           )
704         )
705       )
706     )
707   )
708
709   (om::om-make-dialog-item
710     'om::om-button
711     (om::om-make-point 262 60) ; position (horizontal, vertical)
712     (om::om-make-point 160 20) ; size (horizontal, vertical)
713     "Stop"
714     :di-action #'(lambda (b)
715       ;; (setq *is-stopped t)
716       (setf *is-stopped t)
717       (print "search stopped. *is-stopped parameter :")
718       (print *is-stopped)
719     )
720   )
721 )
722 search-buttons
723 )
724 )
725
726
727
728
729 (defun search-next-fux-cp (se)
730   (print "Searching next solution...")
731   ;; (print l)
732   ;; (#<|gil|::bab-engine 40100D21DB> (#<|gil|::int-var 40D038C6BB> #<|gil|::int-var 40
D038C9CB> #<|gil|::int-var 40D038CD93> #<|gil|::int-var 40D038D233> #<|gil|::int-var 40
D038D71B> #<|gil|::int-var 40D038DC03> #<|gil|::int-var 40D038E0EB> #<|gil|::int-var 40
D038E5D3> #<|gil|::int-var 40D038E993> #<|gil|::int-var 40D038C59B> #<|gil|::int-var 40
D038CB53> #<|gil|::int-var 40D038C81B> #<|gil|::int-var 40D038CF63> #<|gil|::int-var 40
D038CB3B> #<|gil|::int-var 40D038D44B> #<|gil|::int-var 40D038CF4B> #<|gil|::int-var 40
D038D933> #<|gil|::int-var 40D038D433> #<|gil|::int-var 40D038DE1B> #<|gil|::int-var 40
D038D91B> #<|gil|::int-var 40D038E303> #<|gil|::int-var 40D038DE03> #<|gil|::int-var 40
D038E753> #<|gil|::int-var 40D038E2EB> #<|gil|::int-var 40D038EAB3> #<|gil|::int-var 40
D03974D3>) #<|gil|::time-stop 40100D16A3> #<|gil|::search-options 40100D1BC3> (1 2))
733   (let (
734     (check t)
735     sol sol-pitches sol-species
736   )
737
738     (time (om::while check :do
739       (print "search next loop")
740       ;; (print *is-stopped)
741       ;; (print (getparam 'is-stopped))
742       ;; ; reset the tstop timer before launching the search
743       ;; (gil::time-stop-reset tstop) ; Done on Gecode side
744       ; try to find a solution
745       (time (setf sol (try-find-solution se))) ; "sol" will contain a void* cast of a
Problem* pointer (the solution space)
746       (if (null sol)
747         ; then check if there are solutions left and if the user wishes to continue
searching
748         ;; (stopped-or-ended (search-stopped se) (getparam 'is-stopped)) ; TODO
check git damien
749         (stopped-or-ended (search-stopped se) *is-stopped) ; TODO check git damien

```

```

750         ; else we have found a solution so break fthe loop
751         (setf check nil)
752     )
753 ))
754
755 (print sol)
756
757 (print "The solution can now be retrieved by evaluating the third output of cp-params.
758 ")
759 ;; (setq sol-pitches (gil::g-values sol the-cp)) ; store the values of the solution
760 (setq sol-pitches (solution-to-int-array sol)) ; store the values of the solution
761 ;; (print "sol-pitches :")
762 ;; (print sol-pitches)
763 (let (
764     (basic-rythmics (get-basic-rythmics *species-list *cf-len sol-pitches sol))
765     (sol-voices (make-list *N-COUNTERPOINTS :initial-element nil))
766 )
767     (loop for i from 0 below *N-COUNTERPOINTS do (progn
768         (setq rythmic+pitches (nth i basic-rythmics)) ; get the rythmic correpsonding
769         to the species
770         (setq rythmic-om (first rythmic+pitches))
771         (setq pitches-om (second rythmic+pitches))
772     )
773     (setf (nth i sol-voices) (make-instance 'voice :chords (to-midicent pitches-om
774 ) :tree (om::mktree rythmic-om '(4 4)) :tempo *cf-tempo))
775 )
776     (make-instance 'poly :voices sol-voices)
777 )
778 )
779 )
780
781 ; try to find a solution, catch errors from Gecode
782 (defun try-find-solution (se)
783     (print "try-find-solution")
784     (handler-case
785         (return-next-solution-space se) ; search the next solution, sol is the space of the
786         solution
787         (error (c)
788             (error "Error on Gecode side when calling return-next-solution-space. Wait a few
789 seconds, then press stop, then next solution again. If that doesn't work, press save
790 config and restart the search.")
791             ;(try-find-solution se)
792         )
793     )
794 )
795
796 ; determines if the search has been stopped by the solver because there are no more solutions
797 ; or if the user has stopped the search
798 (defun stopped-or-ended (stopped-se stop-user)
799     (print (list "stopped-se" stopped-se "stop-user" stop-user))
800     (if (= stopped-se 0); if the search has not been stopped by the TimeStop object, there is
801         no more solutions
802         (error "The search was stopped because no more solution was found. Either the best
803 solution was found or none exist.")
804         )
805     ;otherwise, check if the user wants to keep searching or not
806     (if stop-user
807         (error "The search was stopped. Press next to continue the search.")
808         )
809 )
810
811 ; get the basic rythmic pattern and the corresponding notes the given species
812 ; - species-list: the species [1 2 3 4]
813 ; - len: the length of the cantus-firmus
814 ; - sol-pitches: the whole solution array
815 ; - sol: the solutino space (only useful for the 5th species)
816 ; return format = '(('rythmics-1 pitches-1) '(rythmics-2 pitches-2) ... '(rythmics-n pitches-n
817 ))
818 ; examples:

```

```

813 ; ((1) 5) -> (((1 1 1 1 1) (60 62 64 65 60)))
814 ; ((1 2) 5) -> (((1 1 1 1 1) (60 62 64 65 60)) ((1/2 1/2 1/2 1/2 1/2 1/2 1/2 1/2 1) (60 62 64
      65 64 62 60 62 60)))
815 ; ((2) 5) -> ((1/2 1/2 1/2 1/2 1/2 1/2 1/2 1/2 1 (pitches))
816 ; ((3) 5) -> ((1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1/4 1 (pitches))
817 ; ((4) 5) -> ~((-1/2 1 1 1 1/2 1/2 1 (pitches)) depending on the counterpoint
818
819 (defun get-basic-rythmics (species-list len sol-pitches sol)
820   ;; (print "get-basic-rythmics function entered")
821   (setq len-1 (- len 1))
822   (setq len-2 (- len 2))
823   (let (
824     (rhythmic+pitches (make-list *N-COUNTERPOINTS :initial-element nil))
825     )
826     (loop for i from 0 below *N-COUNTERPOINTS do (progn
827       (case (nth i species-list)
828         (1 (progn
829             (setf (nth i rhythmic+pitches) (list
830               ; rhythm
831               (make-list len :initial-element 1)
832               ; pitches
833               (subseq sol-pitches 0 len)
834             ))
835           (setf sol-pitches (subseq sol-pitches len))
836         )
837         (2 (let (
838             (rhythmic (append (make-list (* 2 len-1) :initial-element 1/2) '(1)))
839             (pitches (subseq sol-pitches 0 (- (* 2 len) 1)))
840             )
841             (if (eq (car (last pitches 4)) (car (last pitches 3))) (progn ; if the
first note in the penult bar is the same as the last in the 2nd-to last
843               ; then ligature them
844               (setf rhythmic (append (butlast rhythmic 4) '(1) (last rhythmic 2)))
845               (loop
846                 for i from (- (length pitches) 4) below (- (length pitches) 1)
847                 do (setf (nth i pitches) (nth (+ i 1) pitches))
848               )
849             ))
850             (if (eq (car (last pitches 3)) (car (last pitches 2))) (progn ; same
but for 3rd-to-last and 2nd-to-last
851               (setf rhythmic (append (butlast rhythmic 3) '(1) (last rhythmic 1)))
852               (loop
853                 for i from (- (length pitches) 3) below (- (length pitches) 1)
854                 do (setf (nth i pitches) (nth (+ i 1) pitches))
855               )
856             ))
857             (setf (nth i rhythmic+pitches) (list
858               rhythmic
859               pitches
860             ))
861             ; remove all the notes we've just considered from sol-pitches
862             (setf sol-pitches (subseq sol-pitches (length pitches)))
863           )
864         )
865         (3 (progn
866             (setf (nth i rhythmic+pitches) (list
867               ; rhythm
868               (append (make-list (* 4 len-1) :initial-element 1/4) '(1))
869               ; pitches
870               (subseq sol-pitches 0 (- (* 4 len) 3))
871             ))
872             ; remove all the notes we've just considered from sol-pitches
873             (setf sol-pitches (subseq sol-pitches (- (* 4 len) 3)))
874           )
875         (4 ;(error "The fourth species is not implemented yet in get-basic-rythmics
function.")
876           (progn
877             (setf (nth i rhythmic+pitches) (build-rythmic-pattern
878               (get-4th-species-array len-2)
879               (get-4th-notes-array (subseq sol-pitches 0 (* 2 len-1)) (+ (* 4 len-1)
1))
880             ))

```



```

944
945 (defun importance-range ()
946   (mapcar #'(lambda (x) (format nil "~A" x)) (loop for i from 1 to 14 collect i))
947 )
948
949
950 ; convert a cost to an integer
951 (defun convert-to-cost-integer (param)
952   (cond
953     ((equal param "No cost") 0)
954     ((equal param "Low cost") 1)
955     ((equal param "Medium cost") 2)
956     ((equal param "High cost") 4)
957     ((equal param "Last resort") 8)
958     ((equal param "Cost prop. to length") (* 2 *cf-len))
959     ((equal param "Forbidden") (* 64 *cf-len))
960   )
961 )
962
963 ; convert a species to an integer
964 (defun convert-to-species-integer-list (param-list)
965   ;; (print param-list)
966   (let (
967     (species-list '())
968   )
969     (dolist (param param-list)
970       (progn
971         (cond
972           ((equal param "1st") (setf species-list (append species-list '(1))))
973           ((equal param "2nd") (setf species-list (append species-list '(2))))
974           ((equal param "3rd") (setf species-list (append species-list '(3))))
975           ((equal param "4th") (setf species-list (append species-list '(4))))
976           ((equal param "5th") (setf species-list (append species-list '(5))))
977           ((equal param "None") nil)
978         )
979       ))
980     (setq *N-COUNTERPOINTS (length species-list))
981     (setq *N-PARTS (+ 1 (length species-list)))
982     species-list
983   )
984 )
985
986 ;; convert the string for the voice type to an integer
987 ;; belong to {"Really far above" "Far above" "Above" "Same range" "Below" "Far below" "Really
988   far below"}
989 ;; convert to {-3 -2 -1 0 1 2 3}
990 (defun convert-to-voice-integer-list (params)
991   (let ((integer-list (make-list *N-COUNTERPOINTS :initial-element nil))) (loop for i from 0
992     below *N-COUNTERPOINTS do
993       (cond
994         ((equal (nth i params) "Really far above") (setf (nth i integer-list) 3))
995         ((equal (nth i params) "Far above") (setf (nth i integer-list) 2))
996         ((equal (nth i params) "Above") (setf (nth i integer-list) 1))
997         ((equal (nth i params) "Same range") (setf (nth i integer-list) 0))
998         ((equal (nth i params) "Below") (setf (nth i integer-list) -1))
999         ((equal (nth i params) "Far below") (setf (nth i integer-list) -2))
1000         ((equal (nth i params) "Really far below") (setf (nth i integer-list) -3))
1001       )
1002     )
1003   integer-list
1004 )
1005
1006 ; convert a slider value to a percentage
1007 (defun convert-to-percent (param)
1008   (float (/ param 100))
1009 )
1010
1011 ; convert a mode to an integer
1012 (defun convert-to-mode-integer (param tone)
1013   (cond
1014     ((equal param "Major") (mod tone 12))
1015     ((equal param "Minor") (mod (+ tone 3) 12))

```

```

1015 ((equal param "None") nil)
1016 )
1017 )
1018
1019 ; define all the global variables
1020 (defun set-global-cf-variables (cantus-firmus borrow-mode)
1021   (defparameter *prev-sol-check nil)
1022   (defparameter rhythmic-pitches nil)
1023   (defparameter rhythmic-om nil)
1024   (defparameter pitches-om nil)
1025   ; get the tonalite of the cantus firmus
1026   (defparameter *tonalite-offset (get-tone-offset cantus-firmus))
1027   ; get the *scale of the cantus firmus
1028   (defparameter *scale (build-scaleset (get-scale) *tonalite-offset))
1029   ; *chromatic *scale
1030   (defparameter *chromatic-scale (build-scaleset (get-scale "chromatic") *tonalite-offset))
1031   ; get the first note of each chord of the cantus firmus
1032   (defparameter *cf (mapcar #'first (to-pitch-list (om::chords cantus-firmus))))
1033   ; get the tempo of the cantus firmus
1034   (defparameter *cf-tempo (om::tempo cantus-firmus))
1035   ; get the first note of the cantus firmus ;; just used for the moment
1036   (defparameter *tone-pitch-cf (first *cf))
1037   ; get the borrowed scale of the cantus firmus, i.e. some notes borrowed from the natural
    scale of the tone (useful for modes)
1038   (setq mode-param (convert-to-mode-integer borrow-mode *tone-pitch-cf))
1039   (if mode-param
1040     (defparameter *borrowed-scale (build-scaleset (get-scale "borrowed") mode-param))
1041     (defparameter *borrowed-scale (list)))
1042   )
1043   ; get notes that are not in the natural scale of the tone
1044   (defparameter *off-scale (set-difference *chromatic-scale *scale))
1045   ; length of the cantus firmus
1046   (defparameter *cf-len (length *cf))
1047   ; *cf-last-index is the number of melodic intervals in the cantus firmus
1048   (defparameter *cf-last-index (- *cf-len 1))
1049   ; *cf-penult-index is the number of larger (n -> n+2) melodic intervals in the cantus
    firmus
1050   (defparameter *cf-penult-index (- *cf-len 2))
1051   ; COST_UB is the upper bound of the cost function
1052   (defparameter COST_UB (* *cf-len 20))
1053   ; *N-COUNTERPOINTS is the number of counterpoints in the counterpoint
1054   (defparameter *N-COUNTERPOINTS -1) ; will be defined when parsing the input
1055 )
1056
1057 ; return the tone offset of the voice
1058 ; => [0, ..., 11]
1059 ; 0 = C, 1 = C#, 2 = D, 3 = D#, 4 = E, 5 = F, 6 = F#, 7 = G, 8 = G#, 9 = A, 10 = A#, 11 = B
1060 (defun get-tone-offset (voice)
1061   (let (
1062     (tone (om::tonalite voice))
1063   )
1064     (if (eq tone nil)
1065       ; then default to C major
1066       0
1067       ; else check if the mode is major or minor
1068       (let (
1069         (mode (om::mode tone))
1070       )
1071         (if (eq (third mode) 300)
1072           (midicent-to-midi-offset (+ (om::tonmidi tone) 300))
1073           (midicent-to-midi-offset (om::tonmidi tone)))
1074       )
1075     )
1076   )
1077 )
1078 )
1079
1080 ; build the list of acceptable pitch based on the scale and a key offset
1081 (defun build-scaleset (scale offset)
1082   (let ((major-modified (adapt-scale scale))
1083         (scaleset (list)))
1084     (loop :for octave :from -1 :below 11 :by 1 append

```

```

1086         (setq scaleset (nconc scaleset (mapcar (lambda (n) (+ (+ n (* octave 12)) offset
1087         )) major-modified)))
1088     )
1089     (setq scaleset (remove-if 'minusp scaleset))
1090     ;; tibo: remove notes higher than 127
1091     (setq scaleset (remove 127 scaleset :test #'<))
1092 )
1093
1094 ; reformat a scale to be a canvas of pitch and not intervals
1095 (defun adapt-scale (scale)
1096   (let ((major-modified (list (first scale))))
1097     (loop :for i :from 1 :below (length scale) :by 1 :do
1098       (setq major-modified (nconc major-modified (list (+ (nth i scale) (nth (- i 1)
1099       major-modified))))))
1100   )
1101   (return-from adapt-scale major-modified)
1102 )
1103
1104 ; returns the list of intervals defining a given mode
1105 (defun get-scale (&optional (mode "ionian (major)"))
1106   (cond
1107     ((string-equal mode "ionian (major)")
1108      (list 2 2 1 2 2 2 1))
1109     ((string-equal mode "dorian")
1110      (list 2 1 2 2 2 1 2))
1111     ((string-equal mode "phrygian")
1112      (list 1 2 2 2 1 2 2))
1113     ((string-equal mode "lydian")
1114      (list 2 2 2 1 2 2 1))
1115     ((string-equal mode "mixolydian")
1116      (list 2 2 1 2 2 1 2))
1117     ((string-equal mode "aeolian (natural minor)")
1118      (list 2 1 2 2 1 2 2))
1119     ((string-equal mode "locrian")
1120      (list 1 2 2 1 2 2 2))
1121     ((string-equal mode "harmonic minor")
1122      (list 2 1 2 2 1 3 1))
1123     ((string-equal mode "pentatonic")
1124      (list 2 2 3 2 3))
1125     ((string-equal mode "chromatic")
1126      (list 1 1 1 1 1 1 1 1 1 1 1))
1127     ((string-equal mode "borrowed")
1128      (list 5 4 2 1))
1129   )
1130 )
1131
1132 ; <chords> a list of chord object
1133 ; Return the list of pitch contained in chords in midi format
1134 (defun to-pitch-list (chords)
1135   (loop :for n :from 0 :below (length chords) :by 1 collect (to-midi (om::lmidic (nth n
1136   chords))))
1137 )
1138
1139 ; convert from MIDICent to MIDI
1140 (defun to-midi (l)
1141   (if (null l)
1142       nil
1143       (cons (/ (first l) 100) (to-midi (rest l))))
1144 )
1145 )

```

```

1156
1157 ; converts a list of MIDI values to MIDICent
1158 (defun to-midicent (l)
1159   (if (null l)
1160       nil
1161       (cons (* 100 (first l)) (to-midicent (rest l))))
1162   )
1163 )
1164
1165
1166 (defun map-mode-to-int (mode)
1167   (cond
1168     ((equal mode "None") 0)
1169     ((equal mode "Major") 1)
1170     ((equal mode "Minor") 2)
1171   )
1172 )
1173
1174
1175 ; TODO : check
1176 (defun convertparam-yes-no (v)
1177   (cond
1178     ((equal v "Yes") 1)
1179     ((equal v "No") 0)
1180   )
1181 )
1182
1183
1184
1185
1186
1187 ; 4SP AND 5SP SPECIFIC
1188
1189 ; build the rythmic pattern for open music from the species array
1190 ; - species-arr: array of integer for species
1191 ; - cp-arr: array of integer for counterpoint notes
1192 ; - rythmic-arr: array of integer for the rythmic (supposed to be nil and then filled by the
1193   recursive function)
1194 ; - notes-arr: array of interger for notes (supposed to be nil and then filled by the
1195   recursive function)
1196 ; - b-debug: boolean to print debug info
1197 ; note: this function has noting to do with GECODE
1198 (defun build-rythmic-pattern (species-arr cp-arr &optional (rythmic-arr nil) (notes-arr nil) (
1199   extended-cp-domain nil) (b-debug nil))
1200   ; print debug info
1201   (if b-debug
1202       (progn
1203         (print "Current species and notes:")
1204         (print species-arr)
1205         (print cp-arr)
1206         (print "Current answer:")
1207         (print rythmic-arr)
1208         (print notes-arr)
1209       )
1210       )
1211   ; base case
1212   (if (null species-arr)
1213       ; then return the rythmic pattern
1214       (list rythmic-arr notes-arr)
1215       )
1216   )
1217   (let (
1218     (sn (first species-arr)) ; current species
1219     (sn+1 (second species-arr)) ; next species
1220     (sn+2 (third species-arr)) ; next next species
1221     (sn+3 (fourth species-arr)) ; next next next species
1222     (cn (first cp-arr)) ; current counterpoint note
1223     (cn+1 (second cp-arr)) ; next counterpoint note
1224     (cn+2 (third cp-arr)) ; next next counterpoint note
1225     (cn+3 (fourth cp-arr)) ; next next next counterpoint note
1226   )
1227     ; replace all nil by -1 for the species
1228     (if (null sn) (setf sn -1))

```

```

1226 (if (null sn+1) (setf sn+1 -1))
1227 (if (null sn+2) (setf sn+2 -1))
1228 (if (null sn+3) (setf sn+3 -1))
1229 ; replace all nil by -1 for the counterpoint
1230 (if (null cn) (setf cn -1))
1231 (if (null cn+1) (setf cn+1 -1))
1232 (if (null cn+2) (setf cn+2 -1))
1233 (if (null cn+3) (setf cn+3 -1))
1234
1235 (if b-debug
1236   (progn
1237     (print (format nil "sn: ~a, sn+1: ~a, sn+2: ~a, sn+3: ~a" sn sn+1 sn+2 sn+3))
1238     (print (format nil "cn: ~a, cn+1: ~a, cn+2: ~a, cn+3: ~a" cn cn+1 cn+2 cn+3))
1239   )
1240 )
1241
1242 (cond
1243   ; 1 if it is the last note [1 -1 ...]
1244   ((and (eq sn 1) (eq sn+1 -1))
1245    (list (append rhythmic-arr (list 1)) (append notes-arr (list cn)))
1246   )
1247
1248   ; if [4 0 4 ...] -> which syncope ?
1249   ((and (eq sn 4) (eq sn+1 0) (eq sn+2 4))
1250    (if (/= cn cn+2) ; syncopation but different notes ?
1251      ; then same as half note
1252      (if (eq sn+3 3)
1253          ; then 1/2 + 1/4 if [4 0 4 3] (syncopation catch up by a quarter note)
1254          (build-rythmic-pattern
1255            (nthcdr 3 species-arr)
1256            (nthcdr 3 cp-arr)
1257            (append rhythmic-arr (list 1/2 1/4))
1258            (append notes-arr (list cn cn+2))
1259            extended-cp-domain
1260          )
1261          ; else 1/2 + 1/2 if [4 0 4 0] (basic syncopation)
1262          (build-rythmic-pattern
1263            (nthcdr 4 species-arr)
1264            (nthcdr 4 cp-arr)
1265            (append rhythmic-arr (list 1/2 1/2))
1266            (append notes-arr (list cn cn+2))
1267            extended-cp-domain
1268          )
1269        )
1270      ; else same as full note syncopated
1271      (if (eq sn+3 3)
1272          ; then 3/4 if [4 0 4 3] (syncopation catch up by a quarter note)
1273          (build-rythmic-pattern
1274            (nthcdr 3 species-arr)
1275            (nthcdr 3 cp-arr)
1276            (append rhythmic-arr (list 3/4))
1277            (append notes-arr (list cn))
1278            extended-cp-domain
1279          )
1280          ; else 1 if [4 0 4 0] (basic syncopation)
1281          (build-rythmic-pattern
1282            (nthcdr 4 species-arr)
1283            (nthcdr 4 cp-arr)
1284            (append rhythmic-arr (list 1))
1285            (append notes-arr (list cn))
1286            extended-cp-domain
1287          )
1288        )
1289    ))
1290
1291   ; 1/8 note (croche) if cn == cn+1 AND [!0 (3 or 4) ...]
1292   ((and (eq cn cn+1) (/= sn 0) (or (eq sn+1 3) (eq sn+1 4)))
1293    (if (>= (lastone notes-arr) cn)
1294      ; then eighth note with the next lower note
1295      (build-rythmic-pattern
1296        (nthcdr 1 species-arr)
1297        (nthcdr 1 cp-arr)
1298        (append rhythmic-arr (list 1/8 1/8))

```

```

1299      (append notes-arr (list cn (find-next-note cn 'lower
extended-cp-domain))))
1300      extended-cp-domain
1301    )
1302    ; else eighth note with the next higher note
1303    (build-rythmic-pattern
1304      (nthcdr 1 species-arr)
1305      (nthcdr 1 cp-arr)
1306      (append rhythmic-arr (list 1/8 1/8))
1307      (append notes-arr (list cn (find-next-note cn 'higher
extended-cp-domain))))
1308      extended-cp-domain
1309    )
1310  )
1311 )
1312
1313 ; silence if [0 0 ...]
1314 ((and (eq sn 0) (eq sn+1 0))
1315   (build-rythmic-pattern
1316     (nthcdr 2 species-arr)
1317     (nthcdr 2 cp-arr)
1318     (append rhythmic-arr (list -1/2))
1319     notes-arr
1320     extended-cp-domain
1321   )
1322 )
1323
1324 ; 1 if [1 0 0 0] (full note)
1325 ((and (eq sn 1) (eq sn+1 0) (eq sn+2 0) (eq sn+3 0))
1326   (build-rythmic-pattern
1327     (nthcdr 4 species-arr)
1328     (nthcdr 4 cp-arr)
1329     (append rhythmic-arr (list 1))
1330     (append notes-arr (list cn))
1331     extended-cp-domain
1332   )
1333 )
1334
1335 ; 1/2 if [2 0 ...] (half note)
1336 ((and (eq sn 2) (eq sn+1 0))
1337   (build-rythmic-pattern
1338     (nthcdr 2 species-arr)
1339     (nthcdr 2 cp-arr)
1340     (append rhythmic-arr (list 1/2))
1341     (append notes-arr (list cn))
1342     extended-cp-domain
1343   )
1344 )
1345
1346 ; 1/4 if [3 ...] (quarter note)
1347 ((eq sn 3)
1348   (build-rythmic-pattern
1349     (nthcdr 1 species-arr)
1350     (nthcdr 1 cp-arr)
1351     (append rhythmic-arr (list 1/4))
1352     (append notes-arr (list cn))
1353     extended-cp-domain
1354   )
1355 )
1356
1357 ; 1/2 if [4 0 1 ...] (penultimate note for the 4th species)
1358 ((and (eq sn 4) (eq sn+1 0) (eq sn+2 1))
1359   (build-rythmic-pattern
1360     (nthcdr 2 species-arr)
1361     (nthcdr 2 cp-arr)
1362     (append rhythmic-arr (list 1/2))
1363     (append notes-arr (list cn))
1364     extended-cp-domain
1365   )
1366 )
1367 )
1368 )
1369 )

```

```

1370
1371
1372 ; return a species array for a 4th species counterpoint
1373 ; - len-2: the length of the counterpoint - 2
1374 (defun get-4th-species-array (len-2)
1375   (append (list 0 0) (get-n-4040 len-2) (list 4 0 1))
1376 )
1377
1378 ; return a note array for a 4th species counterpoint
1379 ; - len: the length of the cantus firmus
1380 (defun get-4th-notes-array (cp len)
1381   (let* (
1382     (notes (make-list len :initial-element 0)) ; notes that we don't care about can be 0
1383   )
1384     (loop
1385       for n from 2 below len by 2 ; we move from 4 to 4 (4 0 4 ...) after the silence (0 0)
1386       at the start
1387       for p in cp
1388       do
1389         (setf (nth n notes) p)
1390       notes
1391     )
1392 )
1393
1394 ; return a list with n * (4 0 4 0), used to build the rhythmic pattern for the 4th species
1395 ; - n: the number of times the pattern is repeated
1396 (defun get-n-4040 (n)
1397   (if (eq n 0)
1398     nil
1399     (append (list 4 0 4 0) (get-n-4040 (- n 1))))
1400 )
1401
1402
1403
1404
1405 ; parse the species array to get the corresponding rhythmic pattern for open music
1406 ; - species-arr: array of integer for species (returned by the next-solution algorithm)
1407 ; - cp-arr: array of integer for counterpoint notes (returned by the next-solution algorithm)
1408 ; note: this function has nothing to do with GECODE
1409 (defun parse-species-to-om-rhythmic (species-arr cp-arr extended-cp-domain)
1410   ; replace the last element of the species array by 1
1411   (setf (first (last species-arr)) 1)
1412   (build-rhythmic-pattern species-arr cp-arr nil nil extended-cp-domain)
1413 )
1414
1415
1416
1417
1418 ; return the last element of a list
1419 (defun lastone (l)
1420   (first (last l))
1421 )
1422
1423 ; return the rest of a list without its last element
1424 (defun restbutlast (l)
1425   (butlast (rest l))
1426 )
1427
1428 ; return the penultimate element of a list
1429 (defun penult (l)
1430   (nth (- (length l) 2) l)
1431 )
1432
1433
1434 ; find the next @type note in the borrowed scale,
1435 ; if there is no note in the range then return the note of the other @type
1436 ; - note: integer for the current note
1437 ; - type: atom [lower | higher] for the type of note to find
1438 ; note: this function has nothing to do with GECODE
1439 (defun find-next-note (note type extended-cp-domain)
1440   (let (
1441     ; first sort the scale corresponding to the type

```

```

1442 (sorted-scale (if (eq type 'lower)
1443 (sort extended-cp-domain #'>)
1444 (sort extended-cp-domain #'<))
1445 ))
1446 )
1447 (if (eq type 'lower)
1448 ; then we search the first note in the sorted scale that is lower than the current
note
1449 (progn
1450 (loop for n in sorted-scale do
1451 (if (< n note) (return-from find-next-note n))
1452 )
1453 ; no note so we return the penultimate element of the sorted scale
1454 (penult sorted-scale)
1455 )
1456 ; else we search the first note in the sorted scale that is higher than the
current note
1457 (progn
1458 (loop for n in sorted-scale do
1459 (if (> n note) (return-from find-next-note n))
1460 )
1461 ; no note so we return the penultimate element of the sorted scale
1462 (penult sorted-scale)
1463 )
1464 )
1465 )
1466 )

```

C.7.2 package.lisp

```

1 (in-package :om)
2
3 (defvar *FuxCP-path* (make-pathname :directory (append (pathname-directory *load-pathname*) (
list "FuxCP"))))
4
5 (require-library "GIL")
6
7 (defpackage :fuxcp
8 (:use "COMMON-LISP" "OM" "CL-USER"))

```

C.7.3 problem-wrapper.lisp

```

1 (cl:defpackage "gilf"
2 (:nicknames "GILF")
3 (:use common-lisp :cl-user :cl :cffi))
4
5 (in-package :gilf)
6 (in-package :fuxcp)
7
8 (print "Loading gecode-wrapper...")
9
10 (defparameter DFS 0)
11 (defparameter BAB 1)
12 ; corresponds to enum values in gecode_problem.h, but can be used graphically in om
13 (defun bab ()
14 BAB
15 )
16 (defun dfs ()
17 DFS
18 )
19
20 ;;;;;;;;;;;;;;
21 ;; Problem methods ;;
22 ;;;;;;;;;;;;;;
23
24
25 ;; This function is used to convert lisp lists into int pointers so they can be passed to c++
26 (defun new-ctp-problem (cf splist voicetypes borrowmode minskips generalcst motioncst
melodiccst specificcst costimp toffset scale chromscale borscale)

```

```

27 (let (
28   (cfi (cffi::foreign-alloc :int :initial-contents cf))
29   (spl (cffi::foreign-alloc :int :initial-contents splist))
30   (vt (cffi::foreign-alloc :int :initial-contents voicetypes))
31   (gen (cffi::foreign-alloc :int :initial-contents generalcst))
32   (mot (cffi::foreign-alloc :int :initial-contents motioncst))
33   (mel (cffi::foreign-alloc :int :initial-contents melodiccst))
34   (spe (cffi::foreign-alloc :int :initial-contents specificcst))
35   (cst (cffi::foreign-alloc :int :initial-contents costimp))
36   (sca (cffi::foreign-alloc :int :initial-contents scale))
37   (chr (cffi::foreign-alloc :int :initial-contents chromscale))
38   (bor (cffi::foreign-alloc :int :initial-contents borscale))
39 )
40 (new-problem cfi (length cf) (length splist) spl vt borrowmode minskips gen mot mel spe
41   cst toffset sca (length scale) chr (length chromscale) bor (length borscale))
42 )
43
44 (cffi::defcfun ("create_new_problem" new-problem) :pointer
45   "Creates a new instance of the problem. Returns a void* cast of a Problem*."
46
47   (cantus-firmus      :pointer :int) ; the cantus firmus, in MIDI values
48   (cf-size            :int)         ; the size of the cantus firmus
49   (n-counterpoints    :int)         ; number of counterpoints
50   (species-list       :pointer :int) ; list of the species of the counterpoints (size=
51     n-counterpoints)
52   (voice-types        :pointer :int) ; list of the voice types of the counterpoints (
53     size=n-counterpoints)
54   (borrow-mode        :int)         ; borrowing mode : 0 if "None", 1 if "Major", 2 if
55     "Minor"
56   (min-skips-percent  :int)         ; min skips slider
57   (general-cost-values :pointer :int) ; size = 8
58   (motion-cost-values :pointer :int) ; size = 3
59   (melodic-cost-values :pointer :int) ; size = 8
60   (specific-cost-values :pointer :int) ; size = 7
61   (cost-importances   :pointer :int) ; size = 14
62   (scale               :pointer :int)
63   (scale-size          :int)
64   (chromatic-scale     :pointer :int)
65   (chromatic-scale-size :int)
66   (borrowed-scale      :pointer :int)
67   (borrowed-scale-size :int)
68   ; TODO add here any additional arguments that your Problem constructor takes
69 )
70
71 (cffi::defcfun ("get_size" get-size) :int
72   "Returns the size of the space."
73   (sp :pointer) ; a void* cast of a Problem*
74 )
75
76 (cffi::defcfun ("delete_pointer" delete-pointer) :void
77   "Deletes the pointer passed as argument. "
78   (p :pointer)
79 )
80
81 (cffi::defcfun ("delete_solver_pointer" delete-solver-pointer) :void
82   "Deletes the pointer to a solver (search engine) passed as argument. "
83   (p :pointer)
84 )
85
86 ;;;;;;;;;;;;;;
87 ;; Search engine methods ;;
88 ;;;;;;;;;;;;;;
89
90 (cffi::defcfun ("create_solver" create-solver) :pointer
91   "Creates a DFS<Problem> object. Returns a void* cast of a DFS<Problem> object."
92   (sp :pointer) ; a void* cast of a Problem*
93   (solver-type :int); an integer representing the type of the solver (see above)
94 )
95
96 (cffi::defcfun ("return_next_solution_space" return-next-solution-space) :pointer

```

```

96     "Returns a pointer to the next solution of the problem. Returns a void* cast of a Problem
97     *."
98     (solver :pointer) ; a void* cast of a Base<Problem>* pointer
99 )
100
101 (cffi::defcfun ("search_stopped" search-stopped) :int
102     "Returns 1 if the seach was stopped by the TimeStop object, and 1 otherwise."
103     (solver :pointer) ; a void* cast of a Base<Problem>* pointer
104 )
105
106
107 ;;;;;;;;;;;;;;
108 ;; Solution handling ;;
109 ;;;;;;;;;;;;;;
110
111 (cffi::defcfun ("return_solution" return-solution) :pointer
112     "Returns a int* that are the values of a solution."
113     (sp :pointer) ; a void* cast of a Problem object that is a solution of the problem.
114 )
115
116
117 ;; ===== NEEDED BY 5SP PARSEER =====
118
119 (cffi::defcfun ("return_species_array_5sp" return-species-array-5sp) :pointer
120     "Returns a int* that are the values of the species array for the fifth species, of the
121     ctp-index'th counterpoint."
122     (sp :pointer) ; a void* cast of a Problem object that is a solution of the problem.
123     (ctp-index :int) ; the index of the counterpoint (is it the first, the second or the
124     third counterpoint)
125 )
126
127 (cffi::defcfun ("return_extended_cp_domain" return-extended-cp-domain) :pointer
128     "Returns a int* that are the values of the extebded cp domain (for the fifth species
129     parser), of the ctp-index'th counterpoint."
130     (sp :pointer) ; a void* cast of a Problem object that is a solution of the problem.
131     (ctp-index :int) ; the index of the counterpoint (is it the first, the second or the
132     third counterpoint)
133 )
134
135 (cffi::defcfun ("get_extended_cp_domain_size" get-extended-cp-domain-size) :int
136     "Returns the size of the extended cp domain."
137     (sp :pointer) ; a void* cast of a Problem*
138     (ctp-index :int)
139 )
140
141 (defun species-array-to-int-array (sp ctp-index cf-len)
142     "Returns the values the variables have taken in the solution as a list of integers. Casts
143     a int* into a list of numbers."
144     "sp is a void* cast of a Problem* that is a solution to the problem. Calling this funciton
145     on a non-solution
146     will result in an error."
147     (if (cffi::null-pointer-p sp)
148         (error "Error when retrieving species array for the fifth species.")
149         )
150     (let* (
151         (size (- (* cf-len 4) 3))
152         (ptr (return-species-array-5sp sp ctp-index))
153         )
154         (loop for i from 0 below size
155             collect (cffi::mem-aref ptr :int i)
156             )
157     )
158 )
159
160 (defun ext-domain-to-int-array (sp ctp-index)
161     "Returns the values the variables have taken in the solution as a list of integers. Casts
162     a int* into a list of numbers."
163     "sp is a void* cast of a Problem* that is a solution to the problem. Calling this funciton
164     on a non-solution
165     will result in an error."

```

```

160     (if (cffi::null-pointer-p sp) ; TODO check
161         (error "No (more) solutions."))
162     )
163     (let* (
164         (size (get-extended-cp-domain-size sp ctp-index))
165         (ptr (return-extended-cp-domain sp ctp-index))
166     )
167     (loop for i from 0 below size
168         collect (cffi::mem-aref ptr :int i)
169     )
170 )
171 )
172
173
174 ;; =====
175
176 (defun solution-to-int-array (sp)
177     "Returns the values the variables have taken in the solution as a list of integers. Casts
178     a int* into a list of numbers."
179     "sp is a void* cast of a Problem* that is a solution to the problem. Calling this function
180     on a non-solution
181     will result in an error."
182     (if (cffi::null-pointer-p sp) ; TODO check
183         (error "No (more) solutions."))
184     )
185     (let* (
186         (size (get-size sp))
187         (ptr (return-solution sp))
188     )
189     (loop for i from 0 below size
190         collect (cffi::mem-aref ptr :int i)
191     )
192 )
193
194 (print "finished loading gecode-wrapper")

```

C.7.4 FuxCP.lisp

```

1 ; if problem with rpath, use
2 ; install_name_tool -change gecode.framework/Versions/49/gecode /Library/Frameworks/gecode.
   framework/Versions/49/gecode libgecode.dylib
3
4 ;; (in-package :om)
5
6 (defvar *fuxcp-sources-dir* nil)
7 (setf *fuxcp-sources-dir* (make-pathname :directory (append (pathname-directory *load-pathname
   *) '("sources"))))
8
9 (defvar *libgecode* nil)
10
11 ; trying to load mac library and then linux library if mac doesn't work
12 (handler-case
13     (progn
14         (setf *libgecode* (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :
   name "libgecode.dylib"))
15         (if (equal (cffi:load-foreign-library *libgecode*) nil)
16             (print "There is a problem loading the Framework. Please double check that Gecode
   is correctly installed and you are using the appropriate version of GiL for your Operative
   System"))))
17     (t (c)
18         (progn
19             (setf *libgecode* (make-pathname :directory (pathname-directory *fuxcp-sources-dir
   *) :name "libgecode.so"))
20             (if (equal (cffi:load-foreign-library *libgecode*) nil)
21                 (print "There is a problem loading the Framework. Please double check that
   Gecode is correctly installed and you are using the appropriate version of GiL for your
   Operative System"))))
22         )
23 )

```

```
24
25
26 (compile&load (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "
    package" :type "lisp"))
27 (compile&load (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "
    problem-wrapper" :type "lisp"))
28 (compile&load (make-pathname :directory (pathname-directory *fuxcp-sources-dir*) :name "
    interface" :type "lisp"))
29
30
31 (fill-library '(
32   ("Solver" nil (fuxcp::cp-params) nil)
33 ))
34
35 (print "FuxCP Loaded")
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | www.uclouvain.be/epl