"Melodizer Rock: A Constraint Programming Tool for Composing Rock Music"

Lepeltier, Félix ; Otlet, Sophie

ABSTRACT

This master's thesis presents Melodizer Rock, a tool which aims to assist composers in their rock music creation process. It is important to specify that the aim isn't to replace the musician's creativity with this tool. On the contrary, it is a tool that can and should be used to inspire composers. Melodizer Rock builds on top of three previous theses. Firstly, Baptiste Lapière's work, which was a rhythm-oriented thesis [1], generated scores which respect rhythm-specific rules given by the user. Soon thereafter, Damien Sprockeels' work on Melodizer, a pitch-oriented thesis [2], generated melodies which respect constraints given by the user. Lastly, Melodizer 2.0 aimed to combine both works, and created a tool allowing pitches and rhythms to be played simultaneously [3]. This was the work of Clément Chardon, Amaury Diels, and Federico Gobbi. Now, Melodizer Rock adds to the capabilities of Melodizer 2.0, by encoding the structure of a complete rock song within the tool. Said structure was extracted from Drew Nobile's thesis "A Structural Approach to the Analysis of Rock Music" [4], and is based on the hierarchical AABA, and srdc structure. The composer's musical ideas are given to the tool, through an easy to use interface, and are then used to build a Constraint Satisfaction Problem (CSP). Ideas are typically represented by easily quantifiable metrics, such as the pitch range or note length of a piece. However, such ideas can very well be short melodies which the composer is keen to expand on, or create a whole musical piece based off of. The aforementioned CSP is defin...

CITE THIS VERSION

Lepeltier, Félix ; Otlet, Sophie. *Melodizer Rock: A Constraint Programming Tool for Composing Rock Music.* Ecole polytechnique de Louvain, Université catholique de Louvain, 2023. Prom. : Van Roy, Peter. http://hdl.handle.net/2078.1/thesis:40695

UCLouvain

École polytechnique de Louvain

Melodizer Rock

A Constraint Programming Tool for Composing Rock Music

Authors: **Félix Lepeltier, Sophie Otlet**
Supervisor: **Peter Van Roy**
Readers: **Damien Sprockeels, Karim Haddad, Vianney Coppé**
Academic year 2022–2023
Master [120] in Computer Science and Engineering

Abstract

This master's thesis presents Melodizer Rock, a tool which aims to assist composers in their rock music creation process. It is important to specify that the aim isn't to replace the musician's creativity with this tool. On the contrary, it is a tool that can and should be used to inspire composers. Melodizer Rock builds on top of three previous theses. Firstly, Baptiste Lapière's work, which was a rhythm-oriented thesis [1], generated scores which respect rhythm-specific rules given by the user. Soon thereafter, Damien Sprockeels' work on Melodizer, a pitch-oriented thesis [2], generated melodies which respect constraints given by the user. Lastly, Melodizer 2.0 aimed to combine both works, and created a tool allowing pitches and rhythms to be played simultaneously [3]. This was the work of Clément Chardon, Amaury Diels, and Federico Gobbi. Now, Melodizer Rock adds to the capabilities of Melodizer 2.0, by encoding the structure of a complete rock song within the tool. Said structure was extracted from Drew Nobile's thesis *A Structural Approach to the Analysis of Rock Music* [4], and is based on the hierarchical *AABA*, and *srdc* structure.

The composer's musical ideas are given to the tool, through an easy to use interface, and are then used to build a Constraint Satisfaction Problem (CSP). Ideas are typically represented by easily quantifiable metrics, such as the pitch range or note length of a piece. However, such ideas can very well be short melodies which the composer is keen to expand on, or create a whole musical piece based off of. The aforementioned CSP is defined by the composer's musical ideas, to which each solution represents a potentially interesting and novel musical piece that might inspire them. Melodizer Rock is built as a library supplementing OpenMusic, a musical composition tool developed by IRCAM. GiL was used to connect OpenMusic to the constraint programming library Gecode, as OpenMusic is written in *Common Lisp* and Gecode in `C++`.

Acknowledgements

We would like to express our sincere gratitude to
Peter Van Roy,
Damien Sprockeels,
Karim Haddad from IRCAM,
Vianney Coppé,
Vanessa Maons and the INGI System Team
For the invaluable help they provided throughout our master's thesis.

Contents

Chapter 1

Introduction

Nowadays, more and more tasks are executable with the aid of computers. This digital revolution has led to the creation of tools with incredible capabilities, notably with the recent advances in the field of generative Artificial Intelligence. Any person can now use a broadly available model such as *ChatGPT-4*, and submit this prompt: *"Generate the melody for a piece of Rock Music similar to the Beatles"*. However these data-driven approaches generate answers based off of existing data. The problem with this approach is that entirely novel solutions won't ever be found.

The technical approach used in Melodizer Rock, Constraint Programming, represents music as a problem which it tries to solve. Such an approach allows these novel solutions to be found when they exist, and gives seemingly creative results which the composer might not have thought of.

Among all existing music genres, why rock? Rock was chosen for its broad appeal and popularity, along with its strong rhythmic foundation, and dynamic variations. All of which are key factors to conveying emotions to its listeners. Over the past decade, works such as *A Structural Approach to the Analysis of Rock Music* [4] showcased insightful and approachable structures of rock music, giving the foundational knowledge needed to achieve Melodizer Rock's goals.

Obviously, a tool such as Melodizer Rock won't create the perfect song by itself. It will still need the composer's input, and might only serve as an inspiration. The Rolling Stones said it best:

> *"You can't always get what you want, but if you try,*
> *sometimes, you might find, you get what you need."*
> The Rolling Stones (1969)

1.1 Context and Outline

This master's thesis presents Melodizer Rock, a tool which aims to assist composers in their rock music creation process. It is important to specify that the aim isn't to replace the musician's creativity with this tool. On the contrary, it is a tool that can and should be used to inspire composers. Melodizer Rock builds on top of three previous theses.

Firstly, Baptiste Lapière's work, which was a rhythm-oriented thesis [1], generated scores which respect rhythm-specific rules given by the user. Soon thereafter, Damien Sprockeels' work on Melodizer, a pitch-oriented thesis [2], generated melodies which respect constraints given by the user. Lastly, Melodizer 2.0 aimed to combine both works, and created a tool allowing pitches and rhythms to be played simultaneously [3]. This was the work of Clément Chardon, Amaury Diels, and Federico Gobbi.

Now, Melodizer Rock adds to the capabilities of Melodizer 2.0, by encoding the structure of a complete rock song within the tool. Said structure was extracted from Drew Nobile's thesis *A Structural Approach to the Analysis of Rock Music* [4], and is based on the hierarchical $AABA$, and *srdc* structure. Melodizer Rock was thought of such that composers can give a high level representation of the type of music they wish to compose, alongside some potential source melodies, and create music scores which respect the given specifications.

In practice, the composer's musical ideas are given to the tool, through an easy to use interface, and are then used to build a Constraint Satisfaction Problem (CSP). Ideas are typically represented by easily quantifiable metrics, such as the pitch range or note length of a piece. However, such ideas can very well be short melodies which the composer is keen to expand on, or create a whole musical piece based off of. The aforementioned CSP is defined by the composer's musical ideas, to which each solution represents a potentially interesting and novel musical piece that might inspire them.

The tools used to build Melodizer Rock are the same as those used for the previous versions of Melodizer. Melodizer Rock is built as a library supplementing OpenMusic, a musical composition tool developed by IRCAM. Modelling the CSP was done through Gecode, and GiL was used to connect OpenMusic to this constraint programming library, as OpenMusic is written in *Common Lisp* and Gecode in `C++`.

1.2 Road-map

It is important to note that some chapters are quite technically demanding, and that as a composer chapters 5 and 6 will be the most relevant. The following road-map gives a brief overview of what each chapter covers.

- **Chapter 2** covers the theoretical background that is required to fully understand this thesis. It contains western tonal music theory concepts and definitions used throughout the thesis, rock music composition concepts on which Melodizer Rock is built, and an overview of what constraint programming is.

- **Chapter 3** goes over the tools which Melodizer Rock is built on. Covering the use of the constraint programming library Gecode, IRCAM's OpenMusic software which Melodizer Rock serves as a library to, and the previous iterations of Melodizer. The discussion on Melodizer 1.0 contains an explanation of the GiL library used to interface Gecode and Common Lisp. Melodizer 2.0's discussion has detailed explanations on how various parts of it served as inspiration to Melodizer Rock.

- **Chapter 4** describes Melodizer Rock's implementation. It discusses the chosen musical representation, the implementation structure, general and block-specific constraints defining Melodizer Rock's Constraint Satisfaction Problem (CSP), and the chosen solver used to solve this CSP.

- **Chapter 5** gives a thorough description of the interface, and is primarily destined for Melodizer Rock's users, meaning composers. It aims to be very comprehensive and uses musical rather than scientific terminology when possible.

- **Chapter 6** is mostly destined to composers, and provides examples on how Melodizer Rock can be used to compose rock music. These examples are progressive and range from rather simple examples, to a full song using source melodies from a rock hit. It aims to be very comprehensive and uses musical rather than scientific terminology when possible.

- **Chapter 7** suggests improvements for extending Melodizer Rock. These improvements are split into various categories and can be thought of as either deepening Melodizer Rock's scope, broadening it, or improving Melodizer Rock's performance.

- **Chapter 8** summarises Melodizer Rock's contributions, and discusses the importance of building such a tool.

Chapter 2

Theoretical Framework

What defines the music that people listen to? How does one write, or read it? What makes it interesting to listen to? To answer these questions, the representation of music must first be defined, then analysed.

Music is a very large domain, it includes several genres themselves divided into different sub-genres, some of which are illustrated in Figure 2.1. The theory presented in this thesis focuses on one specific subset of music: 1960's to 1990's Rock Music

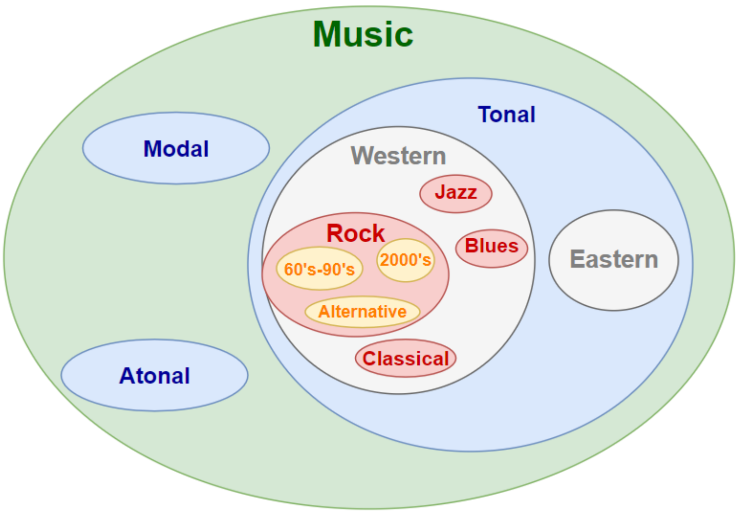


Figure 2.1: Non exhaustive representation of the domains of music

This chapter will introduce the essentials needed to understand the discussions of this thesis. Firstly by explaining the basics of music theory in section 2.1, including the terms, symbols and notations that will be used throughout the following chapters. Then section 2.2 will discuss the different notions inherent to rock music composition. Finally, section 2.3 will describe the basics concepts of Constraint Programming used in the implementation of Melodizer Rock.

2.1 Music Theory

Music theory is not, as its name might suggest, a set of rules that a musician must follow in order to compose a piece, but an ensemble of regulations that can be followed or broken. It is a tool used by musicians to communicate about music. It defines the base on which any musical composition stands to allow other artists to understand, play or adapt the piece.

It is therefore important for anyone that wants to study, compose, or play music, to understand the terms and basics of music theory. The concepts used throughout this thesis are heavily based on the following pieces of literature:

- The simple and clear explanations and definitions of the book *Music Theory for dummies* by M. Pilhofer and H. Day [5],
- The work accomplished by our predecessors, C. Chardon, A. Diels and F. Gobbi for their master thesis *Melodizer 2.0: A Constraint Programming Tool For Computer-aided Musical Composition* [3],
- The more advanced theory defined by R. Gauldin in *Harmonic Practice in Tonal Music* [6].

2.1.1 Music Terminology

This section's aim is to define the musical terms used throughout this thesis, which will be of great use to readers with little musical background, and might serve as a reminder to others.

Accompaniment: "the use of additional voices to support a lead melodic line."[5]

Beat: "one of a series of repeating and consistent pulsations of time in music" [5]. It is used as the basic unit of time to appropriately interpret the intended pace of the song.

Cadence: "the ending of a musical phrase containing points of repose or release of tension".[5]

Chord: "the simultaneous sounding of at least two pitches or notes".[5]

Clef: "the symbol at the beginning of the staff that indicates the pitches of the notes on the staff. There are two predominant clefs, the treble clef for pitches higher than the middle C and the bass clef for pitches lower than the middle C". [3]

Harmony: "the pitches heard simultaneously in ways that produce chords and chord progressions."[5]

Interval: "the distance or difference between the pitches of two notes."[5]

Key note: "the principal and lowest note of the scale in which a piece of music is set" [3]. With a given mode, it defines the scale itself.

Measure: "a segment of written music, contained within two vertical bars, that includes as many beats as the top number of the key signature indicates. It can also be called a bar".[5]

Melody: "a succession of musical tones, usually of varying pitches and rhythms, that together have an identifiable shape and meaning".[5]

Mode: the series of notes into which the octave is divided. It defines the intervals between the different notes of a scale.

Note: "a symbol used to represent the duration of a sound and, when placed on a music staff, the pitch and the sound."[5]

Octave: "two tones that span an interval of twelve semitones. They have the same pitch quality and the same pitch names in Western music."[5]

Pitch: the frequency of vibration of a note, in Western notation. This thesis will use the English notation that uses the first alphabetical letters, from A to G.

Quality: "the number of half steps from one note to another."[5]

Rest: "a symbol used to to notate a period of silence in a musical score."[5]

Rhythm: "a pattern of regular or irregular pulses in music."[5]

Scale: "a series of notes in ascending or descending order that presents the pitches of a tonality, beginning and ending on the tonic of that key."[5]

Score: "the printed representation of a piece of music"[5], composed of at least one staff.

Semitone: "in Western music, it is the smallest interval between two pitches."[3]

Staff: "the five horizontal and parallel lines, containing four spaces between them, on which notes and rests are written."[5]

Tempo: "the rate or speed of the beat in a music piece" [5], generally expressed as beats per minute (bpm).

Time Signature: the notation comprised of two numbers (such as 3/4), which is at the beginning of a piece of music. The top number indicates how many beats are in one measure, and the bottom number indicates the fraction of a whole note representing one beat.

Tonality: "the organisation of a musical piece based on a tonic note (or key note) and a mode."[3]

Tone: a full, or whole, step between pitches. It corresponds to an interval of two semitones.

2.1.2 Rhythm

Rhythm, melody and harmony are the three pillars of music. They form the blueprint of musical composition and are tightly dependant on one another.

Rhythm is one of the basic music concepts that helps with distinguishing different genres. For example, a rock song could be converted to a Waltz by changing only its rhythm. But M. Pilhofer and H. Day [5] point out how important it is to differentiate it from the **surface rhythm** and from the **tempo**. The surface rhythm is the one the listener hears, for example the rhythmic pattern of the drums. Whereas the tempo defines the speed, or frequency, of a piece's rhythm. Meanwhile, the defined rhythm of a piece creates the basic pulse of a song, using the time signature at the beginning of a staff.

Figure 2.2 shows the relation between different note lengths used in this thesis. The smallest one, at the leafs, is called a **sixteenth note** and the longest one, at the root, is called a **whole note**. Each level of the tree has an equal beat duration. The time signature defines the fraction of a whole note used as a beat, as well as the number of beats the bar contains. For example, a time signature of $\frac{3}{4}$ defines that a measure contains 3 fourth-notes (3^d level of the tree).

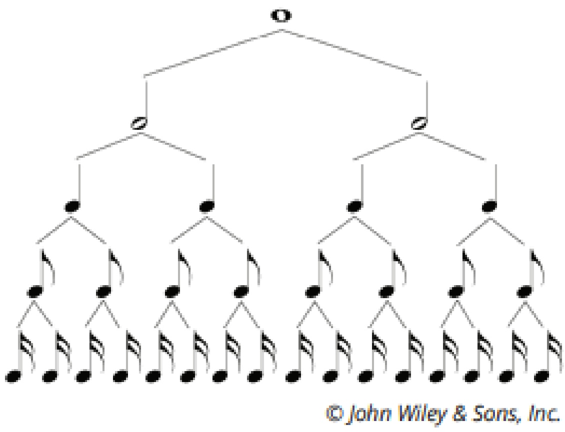


Figure 2.2: Relations between the note lengths from a whole note to sixteenth notes [5]

2.1.3 Melody

Melody is the pitch sequence of a piece of music. In rock music, it is most often the singing line of the song. Two main principles are important to compose a melodic line.

Intervals

A first principle inherent to melodic writing is the notion of **intervals**. An interval is the distance, the frequency, between two pitches. R. Gauldin [7] explains some of their basic principles, paraphrased hereafter:

- Stepwise motion is always preferable to leaps. Leaps over a perfect fifth should be avoided.
- Leaps involving augmented intervals should be avoided, diminished intervals, however, are acceptable.
- Consecutive leaps in the same direction should be avoided unless they outline a triad.

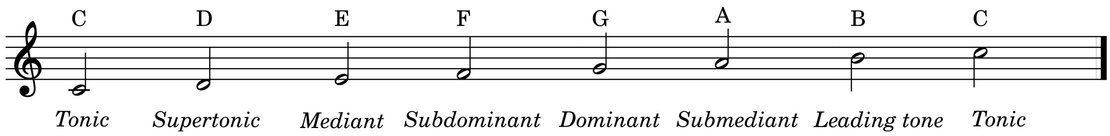
Only a simplified version of the first principle is used in Melodizer Rock as some examples seen in section 2.2 show augmented intervals.

Scales

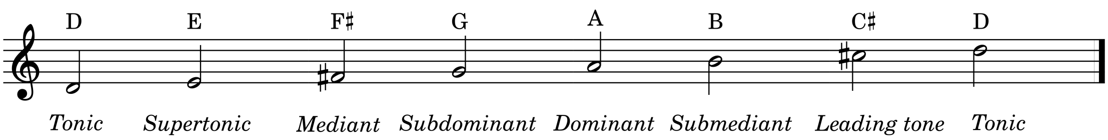
Secondly, to compose a melody, it is important to understand the concept of **scales** on which the notes are chosen. This thesis focuses on 4 modes, differing from one another by the intervals between the different notes. In the following paragraphs, **W** represents a whole step, thus a tone, and **H** a half step, thus a semitone, the sum of both **W+H** represents three semitones.

Since the **major mode** is the common base for other modes, it is defined first. The corresponding intervals are easy to remember, it is mostly one tone between each pitch, except for two notes. As described by M. Pilhofer and H. Day [5], the major scale follows the **WWHWWWH** pattern. Figure 2.3 displays the scale this pattern gives for the C and D keys. The difference due to the placement of the intervals is visible in the alterations on some of the notes. Each note is given a name, or degree, according to its place on the obtained scale. This can be seen on those same Figures, the three most important being:

- **Tonic:** "1^{st} and 8^{th} note on the scale that determine the name of the scale."[5]
- **Sub-dominant:** 4^{th} note on the scale.
- **Dominant:** 5^{th} note on the scale.



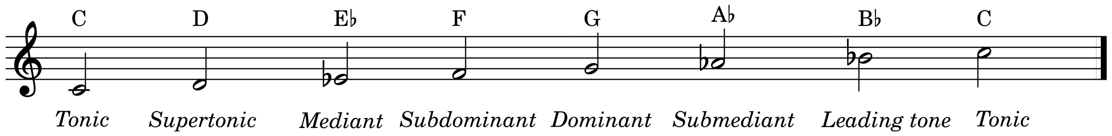
(a) C Major scale



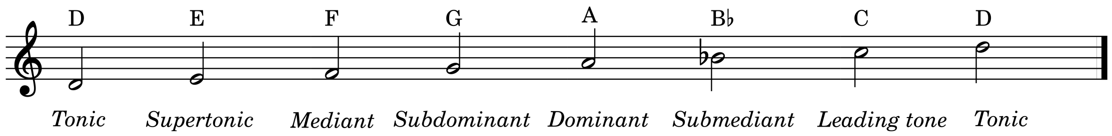
(b) D Major scale

Figure 2.3: Major scales examples

The **natural minor mode** follows the **WHWWHWW** pattern. For a same key, it can be constructed from the major scale by lowering the third, sixth, and seventh degrees by one semitone. Figure 2.4 shows the minor scale for C and D. There exists two other types of minor scales, called **harmonic** and **melodic**, but they will not be further developed in this thesis.



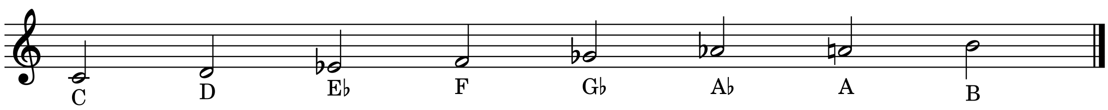
(a) C Minor scale



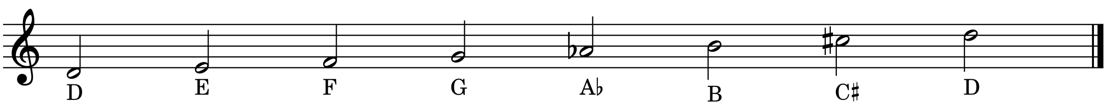
(b) D Minor scale

Figure 2.4: Minor scales examples

The **diminished mode** follows the pattern **WHWHWHW**, that is, every other interval is a whole tone. It can be constructed from the major scale of the same key by using a diminished third, fifth and sixth. As shown in Figure 2.5, it has one more note than a major or minor scale because of the smaller intervals.



(a) C diminished scale



(b) D diminished scale

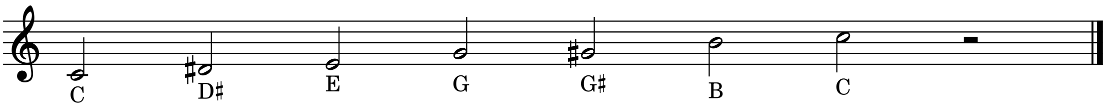
Figure 2.5: Diminished scales examples

The **augmented mode** is a bit more peculiar as it results in a **hexatonic scale**, that is, a scale of six notes. It follows a pattern with greater intervals: **(W+H)H(W+H)H(W+H)H**. Every other interval is thus an augmented second, or minor third. Examples are showed in Figure 2.6.

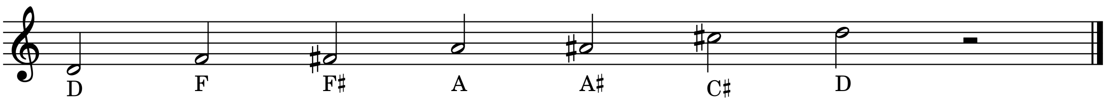
2.1.4 Harmony

Harmony complements the melody by filling out the musical ideas it expresses. It builds **chords**, that is, the tones coming from melody's scale. Then makes them convey emotions, or a sense of beginning or ending to the song. This ordering is also called a chord progression.

Chords are defined by the intervals separating their notes, but can also be built based on the corresponding scale. With the four scales explained in section 2.1.3,



(a) C augmented scale

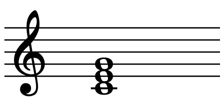


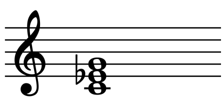
(b) D augmented scale

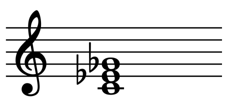
Figure 2.6: Augmented scales examples

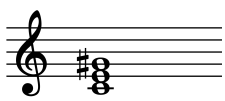
four main type of chords can be built by taking each time the first (tonic), third (mediant) and fifth (dominant) notes. Examples are shown in Figure 2.7 and their integration in Melodizer Rock is explained in section 4.3.1. The chords, defined by their intervals are as follow:

- **Major chords** are composed first, of the root note, then the major third, 4 semitones above the root, and the perfect fifth, 7 semitones above the root, thus a minor third after the second note. See Figures 2.7a & 2.7e.
- **Minor chords** are composed of the root note, the minor third, 3 semitones above the root, and the perfect fifth, or major third from the previous note. See Figures 2.7b & 2.7f.
- **Diminished chords** are composed of the root note, the minor third and the diminished fifth, 6 semitones above the root. It thus uses a minor third followed by a minor third. See Figures 2.7c & 2.7g.
- **Augmented chords** are composed of the root note, the major third and the augmented fifth, 8 semitones above the root. Therefore, it is composed of two major thirds. See Figures 2.7d & 2.7h.

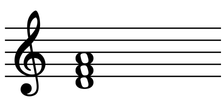

(a) C Major

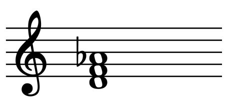

(b) C minor


(c) C diminished


(d) C augmented


(e) D Major


(f) D minor


(g) D diminished

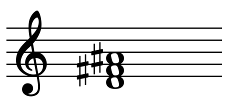
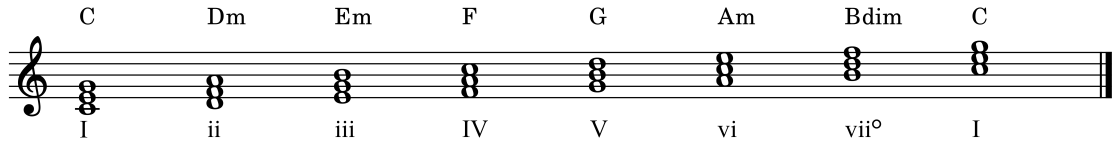

(h) D augmented

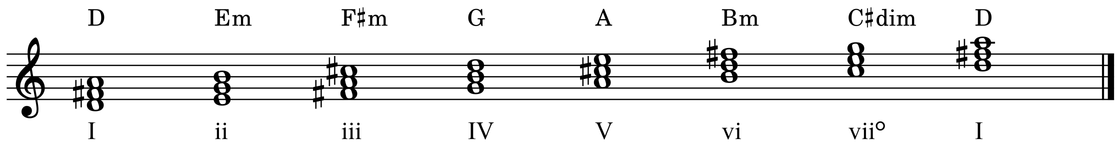
Figure 2.7: Chords examples

Variations of these chords exists, and some are explained in section 7.1.4. For example, a chord could be more than a triad of notes and include a seventh, or could be inverted, that is, include the same notes but with the root note transposed an octave higher.

Chords can be arranged to form a **chord progression**. Using the scale of the melody, if the chords are built with the notes of the scale, they are called **diatonic chords**. If it contains notes outside of the scale, they are **chromatic chords**. Each diatonic chord from a scale is named using roman numeral. Capitalised roman numerals represent the major chords, while lower-case numerals represent minor chords. Diminished chords are represented using the the symbol "°" and augmented chords use the symbol "+". The obtained chords for C major and D major are shown in Figure 2.8, while C and D minor chords are shown in Figure 2.9.

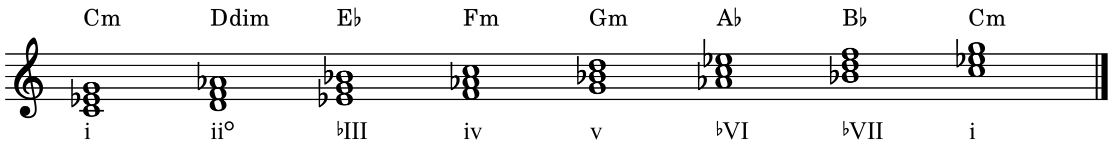


(a) C major diatonic chords

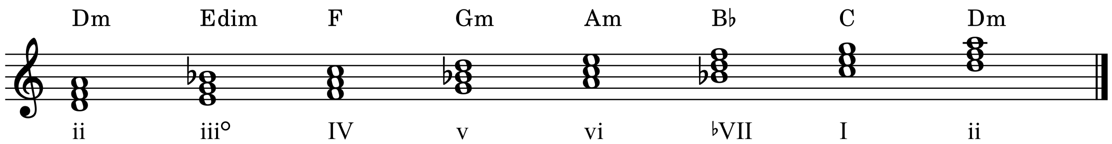


(b) D major diatonic chords

Figure 2.8: Diatonic major chords examples



(a) C minor diatonic chords



(b) D minor diatonic chords

Figure 2.9: Diatonic minor chords examples

2.2 Rock Music Composition

Structural understanding of rock music prior to Drew Nobile's work was incomplete. His thesis *A Structural Approach to the Analysis of Rock Music* [4] proposes three common full song forms used within the genre. The first one, which was used throughout Melodizer Rock is the *AABA* and *srdc* structure. The second expands this first form to a Verse-Prechorus-Chorus structure that is then developed into a Verse-Chorus form.

2.2.1 AABA and s r d c

Rock songs consist of verses and bridges, which correspond respectively to A and B sections. Each of those are themselves divided as 4 phrases: s, r, d and c. This structure is represented in the Figure 2.10.

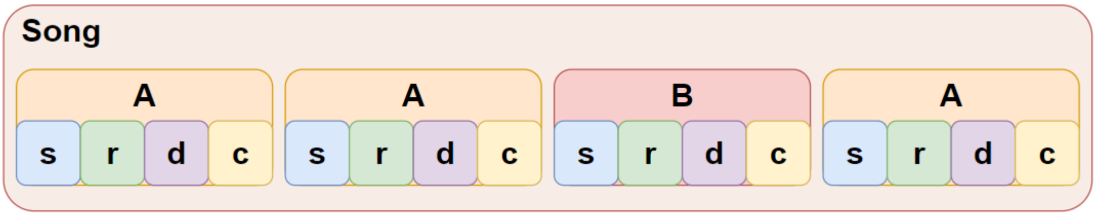


Figure 2.10: $AABA$ and *srdc* structure of a rock song described by Drew Nobile in [4]

$AABA$ and *srdc* forms, alongside their extended siblings (such as $AABABA$, $AABAABA$, etc.), were particularly popular during the 50's and 60's and in some way helped define the pre-psychedelic era of rock music. A straight-forward example analysis of The Beatles' *From me to you* given by D. Nobile [4] will help illustrate just how this form presents itself (Figure 2.11).

The *srdc* structure can be explained as such, in the s section a musical phrase is stated, then in r it is restated and might differ slightly, the third section is d and it acts as a disruption which departs from s and r whilst leading to the conclusive section that is c. The r phrase is similar to s, either by containing similar notes, by having the same note progression but transposed a few semitones, or both at the same time. The d phrase aims to disturb the emotions conveyed by the s phrase, and must thus differ from it. A representative example from *The Jackson 5* with the song *I'll Be There* is found in Figure 2.12.

Drew Nobile distinguished three models of *srdc* structures, which are described in Figure 2.13. In these models, T refers to the tonic, as explained in section 2.1.3, D to the dominant, PD to the pre-dominant and N refers to the off-tonic. Typically, each of those sections spans over two measures, leading to an eight-bar verse though a sixteen-bar verse is not uncommon. Using different models in a song allows to convey different emotions to the listener, mainly due to the tension that the difference in the d phrase communicates.

2.2.2 Cadence

Among the parts which form $AABA$ and *srdc* models, the cadence is most well defined. In Melodizer Rock, it was decided that within an *srdc* form the cadence will be included in c. This means the first model 2.13a described by Drew Nobile will not be suggested, but the composer might build it with constraints on the d phase. The c phase is the conclusion of this form, and must attempt to convey the final emotion that the composer wishes for. Different types of cadences are distinguishable from the chord progressions they use. Each of these cadences induces a different emotion

(Intro)

A	**s**	If there's anything that you want
	r	If there's anything I can do
	d	Just call on me, and I'll send it along
	c	With love, from me to you
A	**s**	I've got everything that you want
	r	Like a heart that's oh, so true
	d	Just call on me, and I'll send it along
	c	With love, from me to you
B		I got arms that long to hold you
		and keep you by my side
		I got lips that long to kiss you
		and keep you satisfied (ooh)
A	**s**	If there's anything that you want
	r	If there's anything I can do
	d	Just call on me, and I'll send it along
	c	With love, from me to you
A		*(harmonica solo)*
B		I got arms that long to hold you...
A		If there's anything that you want...

(Outro)

Figure 2.11: The Beatles' *From me to you* (1963): decomposition in *AABAABA* & *srdc* form [4]

or feeling while listening to a song. Therefore, some are more appropriate for certain uses. Below is a short description of various cadences' chord progressions [8].

Perfect cadences are very conclusive and typically used to announce some ending, although not necessarily the entire piece's ending. They are built from a succession of degrees **V** and **I** chords.

Plagal cadences are less conclusive and less frequently used. They are built from a succession of degrees **IV** and **I** chords.

Half or semi cadences are used to create tension, as the harmony isn't resolved and stays on hold. They are built from a succession of a chord of any degree followed by a chord of degree **V**.

Deceptive cadences create some sense of surprise, as usually it's a degree **I** chord that's played following a degree **V** chord. However, in a deceptive cadence a chord of degree **V** is either followed by a degree **VI** or **III** chord.

Figure 2.12: *The Jackson 5*, *I'll Be There* (1970): first verse with simplified accompaniment

s	**r**	**d**		**c**
T		PD	D	T

(a) Model 1

s	**r**	**d**	**c**	
T		PD	D	T

(b) Model 2

s	**r**	**d**		**c**	
T		N	T (PD)	D	T

(c) Model 3

Figure 2.13: The 3 harmonic models for the *srdc* structure [4]

Many more types of cadences exist, and the theory behind the concept of cadences can be expanded upon quite a bit. However, the knowledge brought by this section largely suffices to understand any further use of cadences throughout the thesis.

2.3 Constraint Programming

The previous sections showed that music theory actually uses a lot of mathematics to define its rules. For example, the time signature is a fraction, the pitches are frequencies, intervals between notes are differences in frequencies ... That being said, musical composition can be expressed as a Constraint Satisfaction Problem, where a song's notes might follow a given tonality or rhythm. Those constraints might differ according to the mood or emotions the composer wants to convey. For instance, a song with slower rhythm or longer notes gives a feeling of melancholy, while a faster pace might transmit happiness. All this can be expressed and used in a program using Constraint Programming to find scores corresponding to the criteria given by the composer.

This section is mainly based on the explanations of the basics of Constraint Programming by K. Apt [9] as well as the previous work done in Melodizer 2.0 [3]. As this thesis uses the Gecode library, explained is section 3.1, to implement the solver behind the program, this section also refers to some descriptions made in the Gecode modelling guide [10].

2.3.1 Definitions

Constraint Programming (CP) is defined by K. Apt [9] as an "alternative approach to programming which relies on techniques that deal with reasoning and computing". In this thesis, it will be used as a programming paradigm that solves problems, by narrowing down variables' domains using mathematical, logical and combinatorial constraints [3].

A **Constraint** on a sequence of variables is a relation on their domains, a requirement that states which combination of values from each variable domains are acceptable. The **domain** of a variable is the set of acceptable values for that variable.

A **Constraint Satisfaction Problem (CSP)** is an application of Constraint Programming composed of a finite set of constraints, each posed on a set of variables. It can be expressed as a tuple $P = (\mathcal{X}, \mathcal{D}, \mathcal{C})$ where

- $\mathcal{X} = \{i, j, ...\}$ is a set of n variables
- $\mathcal{D} = \{D_i, D_j, ...\}$ is a set of n domains for the variables
- $\mathcal{C}$ is a set of constraints imposing logical, arithmetic or combinatorial relations on one or more variables of $\mathcal{X}$.

A **solution** for P is a set of values $\{I_j\}$ such that, $\forall j \in \mathcal{X}$, $I_j \in D_j$ satisfies all the constraints in $\mathcal{C}$.

Constraint Programming can also be used to solve a **Constraint Optimisation Problem**. Those are CSPs where the quality of a solution is estimated with an objective function on the variables. The solver thus tries to minimise or maximise this function to obtain an optimal solution.

The **Search Space** is the set of all possible combinations for all variables of the CSP, represented as a tree (an example is showed in Figure 2.15). The **Search** explores this search space in an organised manner.

Backtracking Search is the simplest form of search, when it explores the space by travelling down a **Search Tree** with Depth First Search. A common way to organise this tree is to impose that each left branch is the assignation of one ore more variables to a value, and the right branch is the removal of those same values

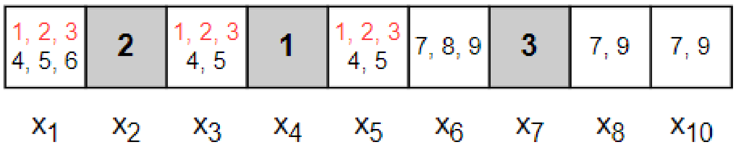


Figure 2.14: Example of propagation on a Sudoku line CSP

from the variables' domains. When updating the variables' domain by going down a branch, the solver must update the domain of other variables in the problem so that they still respect the constraints. This process is called **propagation**. When reaching a state where a variable's domain is empty, the search must backtrack to the previous state because it means no solution can be found with those assignments.

2.3.2 Constraint Propagation

There exists different types of propagation, which update variables' domains differently. Each achieve a different form of local consistency, attempting to approximate the notion of global consistency. A strong propagation prunes more values from the domain of the variables, and often leads to a smaller search tree. Whereas a weak propagation prunes less values, but is less computationally expensive. The propagation levels proposed by Gecode [10] are:

- **Value propagation:** the solver waits for a variable to be bound, then prunes the domain for other variables.
- **Bound propagation:** the solver achieves consistency by only considering minimal and maximal values of the variables' domains.
- **Domain propagation:** the solver propagates a constraint every time a variable's domain changes.

In the example of a Sudoku line where every square must contain a different value between 1 and 9, it means a `distinct` constraint is used on the variables that represent the squares. Given the line of Figure 2.14, as the values of x_2, x_4 and x_7 are fixed respectively to 2, 1 and 3, the three propagation algorithms will prune those same values from the domain of the other squares. For value propagation, only those values will be pruned. The bound propagation algorithm will also see that the values 4 and 5 can be pruned fron x_1's domain because of x_3 and x_5. The domain propagation algorithm will see even further, and will also prune 7 and 9 from x_6's domain because of x_8 and x_9.

It can be seen that domain propagation is the stronger algorithm, but it is also really computationally costly. This is due to it evaluating the constraint for each value in every variable's domain.

2.3.3 Branching Heuristics

Branching is what defines the tree's shape, based on the two-step decisions it takes. It requires to decide which variable to branch on and what values to bind it to at each branch.

Two different strategies could be to do a binary branching to bind a variable to a precise value of its domain, or to split its domain in two parts. Other strategies are possible on n-ary trees but will not be explored in this thesis. The heuristic chosen will determine the size of the search tree. It is therefore important to choose wisely.

Two logical branching heuristics exist for variable and value selection that are widely used in Constraint Programming:

- **First-fail**: when selecting a variable, if there is no solution under a node, the aim is to discover it as soon as possible and not spend too much time on impossible solutions.
- **First-success**: when selecting a value or a partition, if there is a solution under a node, the aim is to find it as soon as possible. Therefore, this strategy must determine the most promising value for the variable to branch on.

These branching heuristics are used together, first-fail for variable selection, and first-success for value selection. Gecode proposes several variable selection strategies from which one can chose from:

- Select the variable with the smallest domain
- Select the most constrained variable
- Select the variable that has failed the most
- Select the variable with highest ratio of degree of constraints over domain size

One must be careful when choosing a variable selection strategy as it could go against the first-fail principle. As for the value selection, choosing a strategy that follows the first-success principle is a more subtle task. Indeed, in the example of a strictly descending melody, then the most promising choice of value for the first variable would be the maximum value of its domain, which would respect the constraints but would not make for an original melody.

2.3.4 Tree Traversal Strategies

Now that how the tree is formed has been established, the decision of how it's explored is left to be made. Gecode proposes several strategies in the form of search engines, but only two were used for this thesis. The following explanations will discuss the Constraint Programming aspects of each strategy, and some of the advantages and disadvantages of each exploration method when applied to musical composition.

Depth-First Search (DFS)

Depth-First search is a well known strategy to explore a tree. Starting at the root, the algorithm goes down every left child until reaching the left-most leaf of the tree. It then goes up one node at a time and explores the right branch of this node. Figure 2.15 shows the exploration path for a tree of eleven nodes.

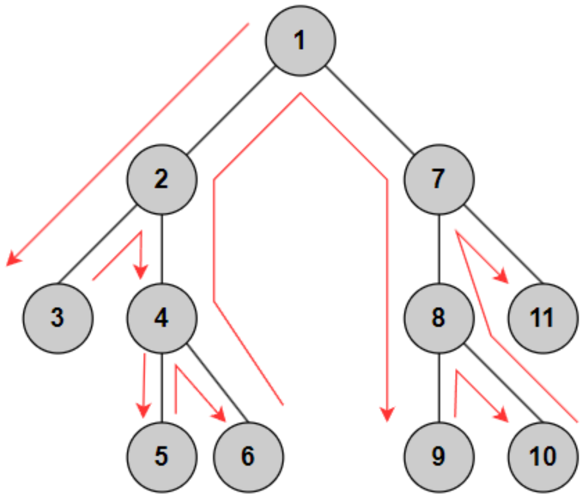


Figure 2.15: Example of a Depth-First exploration path in a tree of eleven nodes

From a composer's view point, this exploration strategy is not the most interesting. Indeed, two successive solutions given by this algorithm in the context of Constraint Programming will be successively explored leaves. As those two leaves are separated by only a few variable assignations, the different musical pieces obtained will differ by as many notes.

Furthermore, when the first left branch does not lead to any solution, a lot of time might be wasted by the search engine on exploring the left-most side of the tree. This is the reason why the branching strategy must be chosen wisely. A well chosen heuristic might lead, in the situation explained before, to explore the right-most side of the tree first, thus finding a solution faster.

Lastly, it is not possible to use a pure DFS search for a Constraint Optimisation Problem (COP). Indeed, this algorithm will explore the tree and give all the solutions found, regardless of any objective function.

Branch and Bound (BAB)

Branch and Bound is an interesting algorithm because it allows for more varied uses. It follows the same exploration principle as Depth-First search, with the subtlety that each time a solution is found, the solver adds new constraints. This makes it possible to solve a Constraint Optimisation Problem by imposing, every time a solution is found, that the next solution must give a better cost than the one found. Therefore, the last solution found will be the best solution, the one minimising or maximising the objective function. This strategy can be used in different cases, for example:

- In the specific domain of musical composition, a composer might want to minimise the **dissonance** between two instruments playing at different scales. This requires a COP rather than a CSP. Another example would be when working with chords. To avoid a chord progression to sound too disjointed, the composer might try to minimise the **span** of the chord progression (that is, the difference between highest and lowest pitch).

- Furthermore, it can be used in a case where there is no objective function to optimise, to impose a **difference** between two successive solutions. This allows for a larger variety of solutions. Indeed, by constraining a certain amount of variables to be different from the current solution, the solver will be forced to find another solution further in the tree.

- Lastly, BAB has a really important upside: **relaxation**. If the solver is not able to find a solution, when the solution space is empty, the problem can be relaxed. This is done by allowing some constraints to be violated. With BAB, the number of violated constraints can be minimised by using reified constraints (see section 3.1.3).

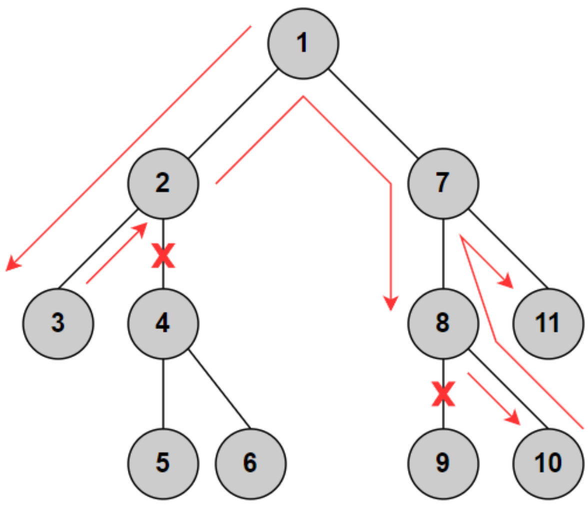


Figure 2.16: Example of Branch and Bound exploration path in a tree of eleven nodes

Figure 2.16 shows an example tree exploration using Branch and Bound. As can be seen, BAB prunes some branches of the tree. This is done either by forbidding the assignment of a variable to certain values, or by computing the objective function and deciding that no better solution can be found on that branch.

Chapter 3

Software Background

This chapter aims to convey sufficient software and tool-specific knowledge, which will prove necessary for the following chapters of this thesis. To this effect, section 3.1 goes over the Gecode constraint programming library, and the different parts that are used in Melodizer Rock. Section 3.2 gives an overview of important concepts within OpenMusic. Section 3.3 describes Damien Sprockeels' work on Melodizer 1.0 and GiL, alongside how it was used as a base for the following Melodizer iterations. Finally, section 3.4 describes Melodizer 2.0 concepts which Melodizer Rock built on.

3.1 Gecode

As described in its Modelling and Programming guide [10], Gecode is an "open, free, portable, accessible and efficient environment for developing constraint-based systems and application". It has been used since the start of Melodizer, in the work of D. Sprockeels, to model the Constraint Satisfaction Problem that is musical composition. This section aims to explain the concepts of Constraint Programming offered by Gecode which are used in the implementation of Melodizer Rock (see chapter 4).

3.1.1 Variables

Gecode offers different types of variables, each associated to its own set of constraints and uses. Three types of variables were used in Melodizer Rock:

- **IntVar:** a variable that can be bound to one integer value, and its domain is the set of integers it can possibly take.
- **BoolVar:** a variable that represents a boolean value. Its initialisation actually takes a domain as an argument but any attempt to create a BoolVar with values different from 0 and 1 will throw an exception.
- **SetVar:** a variable that can be bound to a set of integers. Its domain is also a set of integers, but SetVar variables can take multiple values from this set. A

problem can post a constraint restraining the cardinality of a SetVar, that is, the number of values it can or must be bound to at a time. An interesting set that exists in Gecode is the empty set `IntSet::empty`.

These different variables are initialised as follow:

```
// Creates an integer variable x and sets its domain to {l, ..., h}
IntVar x(home, l, h);
// Creates a Boolean variable y and sets its domain to {0,1}
BoolVar y(home, 0, 1);
// Creates a Set variable z and sets its domain to {{}, ..., {n1, ...,
↪  n2}} and its cardinality domain to [cl ... ch]
SetVar z(home, IntSet::empty, IntSet(n1, n2), cl, ch);
```

It is important to note that a BoolVar is not an IntVar with a domain of $\{0, 1\}$. The only possible way to get an Integer variable that is equal to a Boolean variable is through a channel constraint. When building a problem, it might be useful to use arrays of those variables. To that end, Gecode offers arrays of the aforementioned variables, which can be used like variables. For example:

```
// Initialise an array x of n IntVar variables with domain {l, ..., h}
IntVarArray x(home, n, l, h);
// Initialise an array y of n BoolVar variables with domain {0,1}
BoolVarArray y(home, n, 0, 1);
// Initialise an array z of n SetVar variables with domain {l, ..., h}
↪  and cardinality domain to [cl ... ch]
SetVarArray z(home, n, l, h, cl, ch);
```

It is also possible to instantiate an `IntVar` or a `BoolVar` using an expression of two other integer variables `x` and `y`:

```
IntVar z=expr(*this, a*x+b*y+c); // z = a * x + b * y +c
BoolVar bool=expr(*this, x <= s); // bool = (x <= y)
```

3.1.2 Constraints

Gecode offers a plethora of constraints for the aforementioned variable types. The following section will explain the ones used throughout Melodizer Rock. It is therefore not an exhaustive list of the constraints that Gecode proposes.

Domain Constraints

The **domain constraints** constrain the domain of a variable, or variable array, to a given set of values. They are written as follows:

- For a `IntVar` variable `x`

```
dom(*this, x, l, h); // l is a lower bound, h a higher bound
dom(*this, x, d); // d is an IntArgs, a set of int
```

- For a `BoolVar` variable, the `dom` constraint cannot be used, a relation constraint is used instead
- For a `SetVar` variable `y`, two domain constraints exists, `dom` modifies the values the variable can take, and `cardinality` modifies the number of values it can take, it uses relation types that are explained with the following constraints

```
dom(*this, y, REL_TYPE, l, h);// l and h are the domain bounds
dom(*this, y, REL_TYPE, s); // s is a set
dom(*this, y, d); // d is another variable set
cardinality(*this, x, l, h);// l is a lower bound, h a higher bound
```

- For an `IntVarArray`, `BoolVarArray` or a `SetVarArray` x

```
dom(*this, x, d);//d an array of integer, boolean or set variable
```

Relation Constraints

Relations constraints are the most used constraints, as they can express many different logical and arithmetic relations between variables. They represent a constraint that imposes a relation between two variables. For the different types of variables, there exists different types of relations:

- For `IntVars`:

IRT_EQ	equality ($=$)	IRT_NQ	inequality ($\neq$)
IRT_LE	strictly less ($<$)	IRT_LQ	less or equal ($\leq$)
IRT_GR	Strictly greater ($>$)	IRT_GQ	greater or equal ($\geq$)

- For `BoolVars`, they are more operation than relation types:

BOT_AND	conjuction ($\wedge$)	BOT_OR	disjuction ($\vee$)
BOT_IMP	implication ($\Rightarrow$)	BOT_EQV	equivalence ($\Leftrightarrow$)
BOT_XOR	exclusive or ($\not\Leftrightarrow$)		

- For `SetVars`:

SRT_EQ	equality ($=$)	SRT_NQ	inequality ($\neq$)
SRT_LE	strictly less ($<$)	SRT_LQ	less or equal ($\leq$)
SRT_GR	Strictly greater ($>$)	SRT_GQ	greater or equal ($\geq$)
SRT_SUB	subset ($\subseteq$)	SRT_SUP	superset ($\supseteq$)
SRT_DISJ	disjoint ($\|\|$)	SRT_CMLP	complement ($\bar{\cdot}$)

Relation constraints can be used in different ways with those three types of relations:

- With two variables x and y (that must be of the same type: integer, boolean, set, or arrays of any of those types), a relation can be expressed as follows:

```
rel(*this, x,  REL_TYPE, y);// x REL_TYPE y
```

- Three boolean variables x, y and z, can be constrained in a relation as follows:

```
rel(*this, x, REL_TYPE, y, z);// x REL_TYPE y = z
```

- With an array of integer variables x of size $k-1$, a relation can be imposed between x's elements:

```
rel(*this, x,  REL_TYPE);// x0 REL_TYPE x1 REL_TYPE ... REL_TYPE xk
```

- With a boolean variable x of size $k-1$ and a boolean variable y, a relation can be imposed as follows:

```
rel(*this, REL_TYPE, x, y);// (x0 REL_TYPE x1 ... REL_TYPE xk) = y
```

- With a set variable x and an integer variable y, a relation can be imposed between all of the set's values and the integer:

```
rel(*this, x,  REL_TYPE, y);// all values in x REL_TYPE y
```

- With three set variables x, y and z and a boolean variable b, an if-then-else constraint can be imposed:

```
ite(*this, b, x, y, z);// if b then z = x, else z = y
```

Arithmetic Constraints

Arithmetic constraints are only applicable to integer variables and their arrays. Melodizer Rock actually only uses the minimum and absolute constraints for intervals (as explained in section 4.3.2).

The **minimum constraint** imposes that, for an array of integer variables x and an integer variable y, y is the minimal value of x's variables:

```
min(*this, x, y); // y = min(x)
```

The **absolute constraint** imposes that, for two integer variables x and y, y is the absolute value of x:

```
abs(*this, x, y); // y = |x|
```

Counting Constraints

Counting constraints are quite frequent in Melodizer Rock's implementation. They count how often values are taken by an array of integer variables. This thesis used the simplest version of those constraints. Given and integer variable array x, and two integer variables y and z, it can imposed that

$$z = |\{x_i \in x \mid x_i\ REL\_TYPE\ y\}|$$

In other words, it counts the number of variables of x respecting the relation with y:

```
count(*this, x, y, REL_TYPE, z);
```

Set Operations

Set operations are relation constraints that perform operations on sets, according to the type in the following table:

SOT_UNION	union ($\cup$)	SOT_INTER	intersection ($\cap$)
SOT_DUNION	disjoint union ($\uplus$)	SOT_MINUS	set minus ($\setminus$)

It can be used with set variables x, y and z or an array of set variables s of size $k-1$:

```
rel(*this, x, OP_TYPE, y, REL_TYPE, z);// z REL_TYPE (x OP_TYPE y)
rel(*this, OP_TYPE, s, y);// y = (s0 OP_TYPE s1 ... OP_TYPE sk)
```

3.1.3 Reified Constraints

Reified constraints are a variant of generic constraints whose validity is reflected by a boolean control variable. There exists full and half reification. Full reification corresponds to a two-sided implication, for b a boolean variable and x and y integer variables:

$$b \Leftrightarrow x\ REL\_TYPE\ y$$

Which leads to different cases:

1. If b is assigned to 1, the constraint $x\ REL\_TYPE\ y$ is propagated

2. If b is assigned to 0, the constraint $\neg(x\ REL\_TYPE\ y)$ is propagated
3. If the constraint $x\ REL\_TYPE\ y$ holds, then $b = 1$ is propagated
4. If the constraint $\neg(x\ REL\_TYPE\ y)$ holds, then $b = 0$ is propagated

A half reification can be of different types, each implying some of the different cases above:

RM_IMP	implication ($b \Rightarrow x\ REL\_TYPE\ y$)	cases 1 and 4
RM_PMI	inverse implication ($b \Leftarrow x\ REL\_TYPE\ y$)	cases 2 and 3
RM_EQV	equivalence, full reification	all cases

As shown in this table, the full reification can be expressed with the type RM_EQV. Therefore, to use reification with two integer variables `x` and `y`, a boolean variable `b` and a reification variable `r` one can write:

```
Reify r(b, RM_TYPE);
rel(*this, x, REL_TYPE, y, r);
```

3.1.4 Branching

Gecode offers predefined variable-value branching by calling `branch(*this, x, var_selection, val_selection)`. This function's third argument corresponds to a variable selection strategy, while the fourth argument is for the value selection. The different variable selection strategies available for Melodizer Rock are:

- `INT_VAR_SIZE_MIN()`: selects the variable with the smallest domain size
- `INT_VAR_RND()`: selects the variable at random
- `INT_VAR_DEGREE_MAX()`: selects the variable with the highest propagator degree, the most constrained variable
- `INT_VAR_NONE()`: selects the first unassigned variable

For value selection, there are also several strategies available:

- `INT_VAL_MIN`: selects the smallest value of the domain
- `INT_VAL_RND`: selects a value of the domain at random
- `INT_VAL_SPLIT_MAX`: selects values not greater than $(min + max)/2$
- `INT_VAL_SPLIT_MIN`: selects values not smaller than $(min + max)/2$
- `INT_VAL_MED`: selects the greatest values not bigger than the median

These are non-exhaustive lists of the available strategies in Gecode, and are the ones that were explored for Melodizer Rock.

3.1.5 Search

As explained in the discussion about Constraint Programming 2.3, Gecode offers different **search engines** for different exploration methods. Those used in Melodizer Rock are Depth-first and Branch-and-Bound search engines. These engines possess some functions and attributes that can be used to optimise the search:

- `next()` is a function allowing to request the next solution the solver can find. If there is no more solution in the search space, this function returns NULL.
- `statistics()` is a function that gives statistical information about the search, such as the executed propagators, the number of failed or total nodes explored and the depth of the explored tree.
- `stopped()` is a function that queries whether the search engine has been stopped.
- A destructor deletes all resources used by the search engine.

A `BAB` search engine also has a `constrain()` function that allows to constrain the next solutions based on a obtained solution, as explained in section 2.3.4.

Search Options

A search engine can take options that define how to proceed with the search. The ones used in Melodizer Rock are the number of threads and the Stop objects.

Threads allow a program to run multiple computations in parallel. It allows for a more efficient program, as the complete computation is thus done faster. Imagine that the computer used has m threads, and that the value given in the options is n threads, different cases arise:

- $n = 0$ then m threads are used
- $n \geq 1$ then n threads are used
- $n \leq -1$ then $m + n$ threads are used
- $0 < n < 1$ then $n \cdot m$ threads are used
- $-1 < n < 0$ then $(1 + n) \cdot m$ threads are used

Stop objects implement a single function `stop()` that takes two arguments, a search statistic object and a search options object. This function returns `true` or `false`. A search object acts as a condition to stop the search. When a stop object is given to a search engine, the engine calls the `stop()` function before every exploration step, with the current statistics as argument, and stops the execution if it returns `true`. Once a search engine is stopped, its `next()` function will only return `NULL` as a solution.

3.2 OpenMusic

OpenMusic [11] is a musical composition oriented software built for composers, by researchers at IRCAM Paris. It uses a graphical data-flow approach to musical composition and aims to assist the composer in the creation of their complex musical idea.

3.2.1 Patches

When launching OpenMusic, the first window which a user is greeted by is the Workspace. This workspace allows for the creation of **patches** (cf. Figure 3.1), which correspond to the highest-level form of interactive element within OpenMusic. This concept was inspired by music synthesizers. Within a patch, a user can create their projects by utilising the capabilities of OpenMusic as well as the ones loaded from (user-defined) libraries.

Figure 3.1: Logo of the OpenMusic patch

Some of these capabilities involve creating instances of OpenMusic-specific classes such as voice and poly objects described in section 3.2.3. While others come from libraries such as GiL, described in section 3.3.1.

3.2.2 Editors

Editors are used hand-in-hand with most objects in OpenMusic, and a box's internal editor opens with a double click on said box (within a patch). An editor is typically composed of panels, buttons, check-boxes, sliders, drop-down menus. **Panels** are essentially regions of an arbitrary size which can contain any of the aforementioned elements, within an editor. Buttons, check-boxes, sliders and drop-down menus are all elements which are fairly explicit and whose functionalities won't be further described.

3.2.3 Voice and Poly Objects

Voice and Poly objects are two essential building blocks in OpenMusic, they are used to represent music in a conventional way, by displaying notes on staffs, separated by bars.

Voice objects are used when only one staff is needed to represent the music, as it is their limitation and how they are defined in OpenMusic. **Poly objects** however, are used when multiple staffs are needed to represent the music. Figures 3.2 and 3.3 below showcase the difference between the editors of voice and poly objects. Both of these objects allow the user to listen to the represented music, by connecting a synthesizer to OpenMusic and clicking the play button in the menu bar.

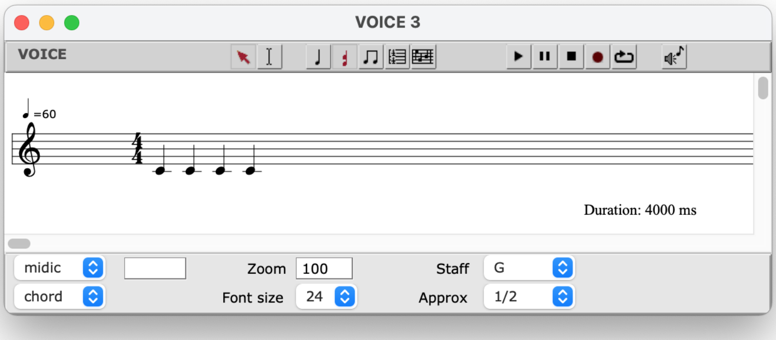


Figure 3.2: Example of a Voice object editor in Open Music

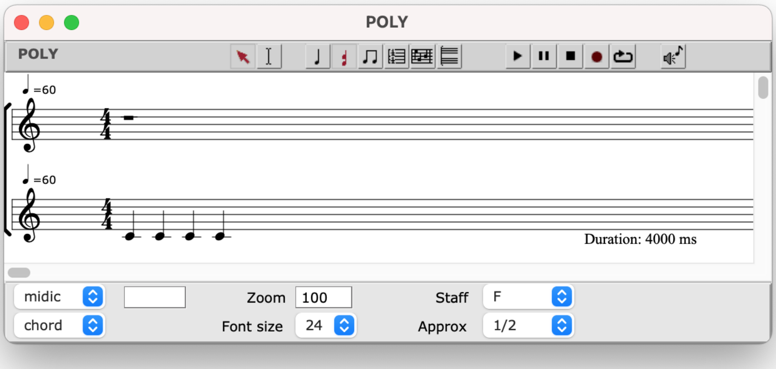


Figure 3.3: Example of a Poly object editor in Open Music

Rhythm trees are a concept used hand in hand with voice and poly objects. They correspond to a list describing the piece's rhythm, by indicating the number of measures, the time signature, and the rhythmic proportions for each measure. Figure 3.4 gives an example of a rhythm tree. In this example, there are two measures, each has a 4/4 time signature, and the second is comprised of 6 notes. The rhythmic proportion list indicates the proportional length of each note according to the total sum on the bar. The two-element sub-list in the second measure's rhythmic proportion list, represents a group of notes. The first element indicates the length duration of the group, and the second element is the rhythmic proportion within the group. Positive values represent notes, and negative values represent rest periods.

3.3 Melodizer 1.0

Melodizer 1.0 was built as an external library supplementing OpenMusic with capabilities beyond this software's initial scope. This is done by introducing Constraint Programming, to enforce musical rules through Constraint Satisfaction Problems,

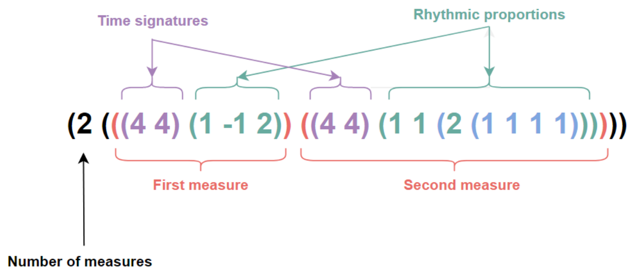


Figure 3.4: Example of a rhythm tree used in Open Music

aiming to provide interesting "out-of-the-box" musical ideas for the composer [2].

3.3.1 GiL

GiL was built as a solution to express Gecode CSPs in Common Lisp. It was initially developed by Baptiste Lapière in the context of his Master Thesis [1] and then further extended by Damien Sprockeels in the same context. It has now become a project with multiple contributors, and is used in every version of Melodizer including Melodizer Rock.

The Gecode concepts described in section 3.1 have analogous implementations in GiL, with almost identical nomenclature, and Melodizer Rock did not bring any significant contribution to GiL. For these reasons, and because GiL will likely be discontinued in future iterations of Melodizer, its implementation details won't be discussed. However, a general explanation of its structure is given below. An example of use is available in Appendix D.

Overview and Explanation

To create this interface between Gecode and Common Lisp, two main parts are needed. Since Gecode is a *C++* library, and Common Lisp can only call foreign *C* code, the Gecode functions used by GiL are first wrapped in *C* code which can then be called by the *Common Foreign Function Interface* (`cffi`) library in Common Lisp. This process is explicitly shown in figure 3.5. Each file in the pipeline has a box associated to it, and when a function in file *A* calls a function from file *B* then an arrow is drawn from file *A* to file *B*.

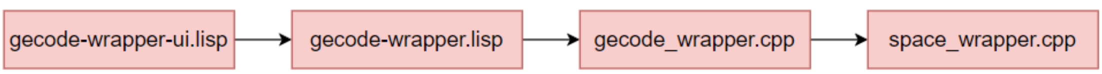


Figure 3.5: GiL function calls path through its various files

As this pipeline is best explained from Gecode to Common Lisp rather than following the actual function calls, let's discuss it in this order.

C Wrapper

The C wrapper is used to wrap the Gecode *C++* code into *C* code, such that Gecode functions could be called from *C*. It is decoupled into two files where `space_wrapper.cpp` wraps Gecode's space, and `gecode_wrapper.cpp` is the *C* library containing calls to the methods defined in the previous file.

Common Lisp Wrapper

This wrapper is defined by the two left-most boxes in figure 3.5 and is used to wrap the *C* library defined in `gecode_wrapper.cpp`. The first part of the Common Lisp wrapper happens in `gecode-wrapper.lisp`. It is where the *C* library is called from, and this is done using `cffi` foreign function calls. The second part happens in `gecode-wrapper-ui.lisp`, and is essentially used to clean up the signature of the functions to facilitate GiL's usage.

3.3.2 Search

The biggest novelty brought with Melodizer 1.0, which was a building block towards Melodizer Rock, is the search mechanism contained within the *Melodizer* object. It is this same mechanism which is used as base in Melodizer 2.0 and Melodizer Rock's search. This addition allowed users to create a CSP, start the search and explore the CSP's solutions, all through a user interface contained within an OpenMusic editor.

3.4 Melodizer 2.0

Melodizer 2.0 [3] extends Damien Sprockeels' Melodizer 1.0 with new capabilities. Two instantiable classes (*Blocks*, *Search*) and a new representation of music are the most notable additions. These additions were a great source of inspiration and essential building blocks in the making of Melodizer Rock. A concise description of the music representation, *Blocks*, and *Search* are provided in the following sections.

3.4.1 Music Representation

Along with Melodizer 2.0 came a new representation of music. This representation aims to create the rhythm of the melody in an intuitive and simple fashion, by using three different variables, `push`, `pull` and `play`. Each variable is an array containing a fixed number of entries, this number corresponds to the number of times the smallest possible note can be played in the musical piece. In their implementation, the smallest note can be played 192 times per measure and if a piece contains only one measure, then the `push` and `playing` arrays will contain 192 elements and `pull` 193. The one additional element in `pull` comes into play at the end of the piece, where all playing notes are pulled.

As this representation is what is used for Melodizer Rock, a more thorough explanation along with examples can be found in section 4.1. The addition of

`SetVarArrays` introduced in Melodizer 2.0, to represent multiple notes at the same time, was also used in Melodizer Rock. Indeed, this data structure was a brilliant way to represent chords being played in the accompaniment.

3.4.2 Blocks

Melodizer 2.0's *Blocks* are a class that can be instantiated within an OpenMusic patch, *Blocks* are used to represent a portion or totality of a musical piece with constraints. Each of these instances represent a CSP that can be solved individually. *Blocks* have multiple inputs and outputs, melodies under the form of voice objects can be taken as input and their content will be added to the CSP. *Blocks* can also be connected together to form one larger portion of the musical piece.

The addition of the *Blocks* class came with notable constraints and interface changes, which are a source of inspiration in Melodizer Rock. Such constraints covered different areas of the musical piece, general constraints relating to *Blocks*, rhythm constraints, and pitch constraints. A thorough explanation of how these constraints were implemented can be found in Melodizer 2.0's master thesis [3].

Among these constraints, several were used in Melodizer Rock: bar length, minimum/maximum pushed notes, minimum/maximum note length, chord key and quality selection, and minimum/maximum pitch. All of these constraints alongside their use within Melodizer Rock, is thoroughly explained in section 4.3. Additionally, the interface used in *Blocks* bridged the gap between the user and the aforementioned constraints, and is the base interface on which Melodizer Rock was built.

3.4.3 Search & Solver

The *Search* class contains the solver, and it is like *Blocks* in the sense that it can be instantiated within an OpenMusic patch. A *Search* object has to be connected to a *Block* instance, or some tree-like structure of interconnected *Blocks* instances, in order to solve the CSP which they are represented by.

The solver defined in Melodizer 2.0's *Search* object was used as inspiration for Melodizer Rock's own solver implementation. It follows the basic setup of a solver in Gecode, while being written in GiL, and interacts with OpenMusic.

First of all, branching and solution variables are picked (a combination of `push`, `pull` and `playing`), and search options are used to instantiate the search-engine. Once this search engine is created for the CSP, the search for solutions is done by interacting with GiL, and returning OpenMusic objects which represent the solution melodies.

The search for solutions is executed in a seperate thread to that of OpenMusic, so as to not hinder or block it during the search. It is an iterative process, where one solution is returned at a time, and the user must interact with the *Search* interface to obtain the next solution.

Chapter 4

Melodizer Rock : Implementation

This chapter describes Melodizer Rock's implementation. Section 4.1 discusses the chosen musical representation. Section 4.2 explains the implementation structure. Sections 4.3 and 4.4 go over the general and block-specific constraints defining Melodizer Rock's constraint satisfaction problem. Finally, section 4.5 describes the solver used to solve the aforementioned constraint satisfaction problem.

4.1 Music Representation

The approach chosen to represent music is to utilise three arrays: `push`, `pull`, `playing`. Each of these arrays has a number of elements equivalent to the maximum number of shortest notes per measure (16 here), multiplied by the number of total measures in the musical piece. These arrays are, for the melodic line, *Gecode* `IntVarArrays` as explained in section 3.1. For the accompaniment, as multiple notes can be pushed, pulled or played simultaneously, they are *Gecode* `SetVarArrays`.

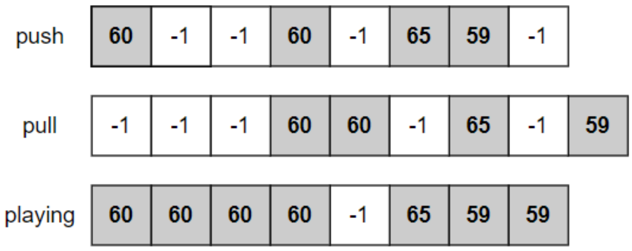
4.1.1 Melody Representation

Rock songs mostly use the melodic line as a singing line. As a human voice can only produce one note at a time, variables allowing one note per quantification were enough and the most practical to use. `IntVarArrays` are perfectly suited for the construction of a melody in Melodizer Rock, as they allow for exactly one integer at every i^{th} quantification.

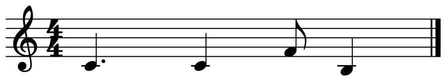
`push[i]` represents a note that is pushed at at the i^{th} quantification in the piece, `playing[i]` represents a note that is played at the i^{th} quantification in the piece, and `pull[i]` represents a note that stops being played (is pulled) at the i^{th} quantification in the piece. Since `IntVarArrays` are used, a pushed note is the only note playing, until it is pulled, which translates to the following implication:

$$
\begin{gathered}
push[i] = note \\
pull[j] = note \\
\Longleftrightarrow \\
\forall k \in [i, ..., j-1] \;\; playing[k] = note
\end{gathered} \tag{4.1}
$$

The figure 4.1a shows an example of a melody representation with 10 time slots. It can be seen that, when the previous playing note is pulled and no note is pushed, then no note is played at that time, leading to a -1 value in `playing`. Meanwhile, a note can be pushed and pulled at the same time, leading to the same note playing twice rather than this note linked to the other and playing once.



(a) Example of the melody representation with the three arrays



(b) Equivalence of the melody on a score

Figure 4.1: Melodic line representation used throughout Melodizer Rock and its equivalence on a score

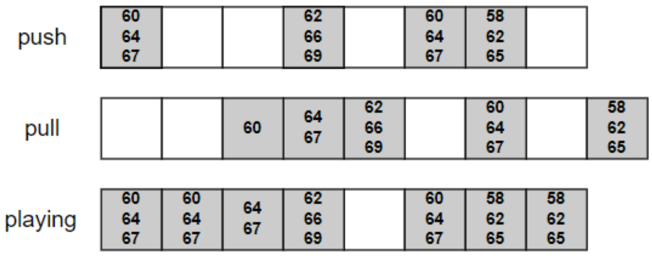
4.1.2 Accompaniment Representation

In rock music, accompaniments are typically guitar or piano chords. Therefore, multiple notes need to play simultaneously. This is not possible with `IntVarArrays`, as it only allows for one integer at the i^{th} quantification, thus only allowing one note to play at a time. This explains the need for these three arrays to be `SetVarArrays`.

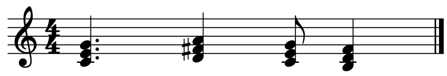
The relation between those arrays is a bit more complicated due to the use of `SetVars`. `push[i]` represents the set of notes that are pushed at the i^{th} quantification, `playing[i]` represents the set of notes playing at the i^{th} quantification, and `pull[i]` represents the set of notes being pulled at the i^{th} quantification. This means that a note pushed is not necessarily the only one playing until it is pulled. The relation from equation 4.1 is thus translated as follows:

$$
\begin{gathered}
note \in push[i] \\
note \in pull[j] \\
\Longleftrightarrow \\
\forall k \in [i, ..., j-1] \;\; note \in playing[k]
\end{gathered} \tag{4.2}
$$

Figure 4.2a shows an example of an accompaniment done using `SetVars`. As can be seen, some simultaneously pushed notes aren't necessarily pulled simultaneously. Those which aren't pulled keep playing.



(a) Example of the accompaniment representation with the three arrays



(b) Equivalence of the accompaniment on a score

Figure 4.2: Accompaniment representation used throughout Melodizer Rock and its equivalence on a score

4.2 Structure

Melodizer Rock follows a tree-like structure based on the rock music genre (explained in more details in section 2.1). The top level being the entire musical piece, it is built from a sequence of blocks A and B. Such blocks are themselves built from a sequence of blocks s, r, d and c, meaning that the structure of a typical $AABA$ sequence for the entire musical piece is represented as depicted in fig. 4.3

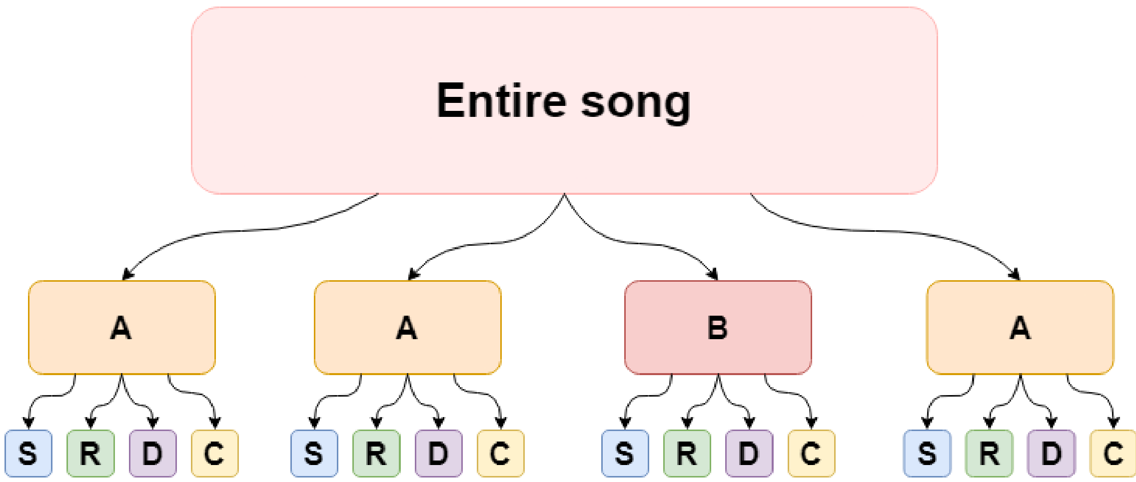


Figure 4.3: $AABA$ tree-like structure representation

In order to implement this structure, a class had to be created for each type of block which it is composed of. As can be expected many of these classes share attributes, and by default these shared attributes inherit values from their parent blocks (e.g. an s block will inherit values from its parent A block for the common attributes). The entire musical piece is represented by the *Rock* block, and is the root of the tree in fig. 4.3. It contains the values which will be inherited by the common attributes of its children blocks' (A and B). Blocks s, r, d and c also inherit values for their common attributes, from their parent block. There also exists some form of horizontal inheritance between A blocks, and between B blocks. This horizontal inheritance implies that the change of an attribute in a block of a certain type, will be propagated to other blocks of the same type. It is active when the `relative-to-same` flag (present in A and B blocks) is set to 1.

Values of many of these attributes can also be changed through the interface, which allows for overriding the vertically inherited values (further explained in section 5). The following sections will describe the aforementioned blocks, alongside their attributes and intricacies.

4.2.1 *Rock*

The *Rock* block contains information pertaining to the entire musical piece. Attributes in *Rock* such as `min-note-length`, `max-note-length`, `chord-key`, `chord--quality`, `min-pitch` and `max-pitch` contain values which will be used to constrain the whole musical piece. *Rock* also has various other attributes such as flags (i.e. to set values if a box in the interface is checked), `block-list` containing the blocks (A and B) which the global musical structure is made of, `solution` and `result` used to handle the current solution of the CSP, `percent-diff` containing the difference percentage to be imposed between successive solutions when using Branch And Bound.

Finally, *Rock* also has two attributes pertaining to the source melodies that can be given to the problem, `melody-source-A` and `melody-source-B`. These attributes are `nil` by default, and only have a value appointed to them when the composer chooses to pass a voice object as input to *Rock*. If source melodies have been given, `melody-source-A` will be used to set the structure's first A block's s phrase, and `melody-source-B` will be used to set the structure's first B block's s phrase.

The source melody given as input mustn't be longer than the default measure quantification used in Melodizer Rock (16 sixteenth notes) times the amount of measures used for the s phrase. If the source melody's length is equal to this value, then the aforementioned behaviour is applied. And if the source melody's length is less than this value, the first part of the s phrase is set to the source melody. After which the remainder of s is constrained as it would be if no source melody was given.

Default values given to the *Rock* block's attributes are typically `nil`, however some exceptions are made so that the software functions as smooth as possible out of the box. These default values are arbitrary and are listed in the following code snippet.

```
(min-note-length :accessor min-note-length :initform 1 :type integer)
(max-note-length :accessor max-note-length :initform 16 :type integer)
(chord-key :accessor chord-key :initform "C" :type string)
(chord-quality :accessor chord-quality :initform "Major" :type string)
```

```
(min-pitch :accessor min-pitch :initform 1 :type integer)
(max-pitch :accessor max-pitch :initform 127 :type integer)
(percent-diff :accessor percent-diff :initform 1 :type integer )
```

4.2.2 *A* and *B*

Unless explicitly stated, *A* and *B* blocks inherit the attributes from *Rock* and may add attributes onto this. Among the attributes which *Rock* has, *A* and *B* blocks don't share the following: `block-list`, `percent-diff`, `solution`, `result`. However, additional attributes they do contain are: `{s,r,d,c}-block`, `parent`, `block-position`, `block-position-A`, `block-position-B`, `similarity-percent-{A, B}0`, and finally `relative-to-{parent,same}`.

`{s,r,d,c}-block` attributes contain instances of s, r, d and c blocks which the current *A* or *B* block is composed of, `parent` is a reference to the current block's parent (a *Rock* block), `block-position` is used to keep track of the position of the current block within the overall structure of the music. `block-position-A` and `block-position-B` are used to keep track of the position of the current *A* or *B* block in relation to blocks of the same type, which form the song's overall structure. These last attributes are mainly used to set the source melodies only for the first *A* and *B* blocks. `similarity-percent-{A,B}0` hold the similarity percent values which are to be imposed on further *A* and *B* blocks, with respect to the first *A* and *B* blocks of the structure.

Finally, `relative-to-{parent,same}` are flags which are used for vertical and horizontal inheritance respectively. By default, vertical inheritance is active (`relative--to-parent` is set to 1) and horizontal inheritance isn't. Vertical inheritance functions in a straightforward manner, where *Rock* attribute values are propagated to its children *A* and *B* blocks. However, when using horizontal inheritance, these attribute values are propagated between *A* and *B* blocks of the same type.

As is the case for the *Rock* block's attributes, these attributes are typically `nil` by default, however there are some exceptions for some of the additional attributes, which are listed in the following code snippet:

```
(relative-to-parent :accessor relative-to-parent :initarg
↪  :relative-to-parent :initform 1 :type integer)
(block-position :accessor block-position :initform -1 :type integer)
(similarity-percent-A0 :accessor similarity-percent-A0 :initform 50 :type
↪  integer)
(similarity-percent-B0 :accessor similarity-percent-B0 :initform 50 :type
↪  integer)
(block-position-A :accessor block-position-A :initform -1 :type integer)
(block-position-B :accessor block-position-B :initform -1 :type integer)
```

```
(s-block :accessor s-block :initarg :s-block :initform (make-instance
↪  's))
(r-block :accessor r-block :initarg :r-block :initform (make-instance
↪  'r))
(d-block :accessor d-block :initarg :d-block :initform (make-instance
↪  'd))
(c-block :accessor c-block :initarg :c-block :initform (make-instance
↪  'c))
```

4.2.3 s, r, d, and c

Unless explicitly stated, s, r, d and c blocks inherit the attributes from A and B and add some new ones onto this. The attributes which they don't share with A and B are: `{s,r,d,c}-block`, `block-position`, `similarity-percent-{A,B}0` and `block-position-{A,B}`. Additional attributes used in s, r, d and c blocks include: `accomp`, `similarity-percent-s`, `difference-percent-s`, `cadence-type`, and `min-note-length-mult`. Their default values aren't `nil` and are listed in the code snippets below.

The `accomp` attribute is one that all s, r, d and c blocks have, and points to an instance of the *Accompaniment* block described in section 4.2.4.

```
(accomp :accessor accomp :initarg :accomp :initform (make-instance
↪  'accompaniment))
```

r Dependency with s

`similarity-percent-s` is an attribute of r blocks, and is a percent value describing the similarity that is to be imposed on the r block from it's sibling s block.

```
(similarity-percent-s :accessor similarity-percent-s :initform 50 :type
↪  integer)
```

d Dependency with s

`difference-percent-s` is an attribute of d blocks, and is a percent value describing the difference that is to be imposed on the d block from it's sibling s block.

```
(difference-percent-s :accessor difference-percent-s :initform 75 :type
↪  integer)
```

c Cadence-specific Attributes

`cadence-type`'s value represents the type of cadence that is used in the current block and is an attribute of c.

`min-note-length-mult`'s value represents the value by which the cadence's melody's minimum note length will be multiplied by. The aim is to improve the cadence's conclusive feeling and avoid abrupt endings.

```
(cadence-type :accessor cadence-type :initform "Perfect" :type string)
(min-note-length-mult :accessor min-note-length-mult :initform 2 :type
↪  integer)
```

4.2.4 *Accompaniment*

The *Accompaniment* block is a very bare-bones block and each s, r, d and c has an attribute pointing to one. Each block is then used in the `poly` object alongside the s, r, d and c blocks, in order to include the accompaniment in the music. By default the accompaniment has a note length equal to the quantification of a measure (16), and plays right at the beginning of each measure.

4.3 General constraints

Creating the constraint satisfaction problem as specified by the composer is done recursively, following the arborescent structure pictured in figure 4.3. A function aiming to constrain the *Rock* block, will call a function aiming to constrain each of the A and B blocks of the given structure. Each of these function calls will then call a function posting general and block-specific constraints, on each A or B block's children. This will be explained in more details in section 4.5.1.

As is implied in this short explanation, most constraints are set on the `push`, `pull`, and `playing` variables of leaves in figure 4.3. Building the problem this way rather than posting constraints on each level of the tree, aids in avoiding duplicate constraints. Even if the solver has efficient ways to handle this, it is avoidable and renders the implementation cleaner. The implementation of these constraints is quite lengthy, and of little aid when trying to understand the different links. Therefore, the `C++` code corresponding to the following constraints is available in appendix D.

4.3.1 Accompaniment Constraints

The accompaniment uses some of the constraints from the *Blocks* of Melodizer 2.0 explained in section 4.2. They allow to link the `push`, `pull` and `playing` arrays through set constraints. Constraints are also posted to restrain the number of notes that can play simultaneously. A last set of constraints is picked by the composer through the interface, and posted for every s, r, d and c *Accompaniment* block. All the implementation of the constraints explained hereafter are available in appendix C.1.1.

Link push pull and playing

The first thing to do is to make sure the problem is correctly stated, so that the variables of the problem are correctly linked to one another. Starting from equation 4.2 to derive the constraints for arrays `push`, `pull` and `playing` of size k, $\forall i \in [1, ..., k-1]$:

1. The notes playing at time i are the notes playing at time $i-1$ that weren't pulled, to which are added the notes pushed at time i:

$$playing[i] = playing[i-1] - pull[i] + push[i]$$

2. No note can be pulled at time i if it wasn't playing at time $i-1$:

$$pull[i] \subseteq playing[i-1]$$

3. A note cannot be pushed at time i if it was already playing at time $i-1$ and not pulled at time i:

$$push[i] \cap (playing[i-1] - pull[i]) = \emptyset$$

For the first index of the arrays, the constraints must be adapted:

1. No note can be pulled at the start as no note was playing:

$$pull[0] = \emptyset$$

2. The notes that are pushed at time 0 must play at time 0:

$$push[0] = playing[0]$$

Simultaneous Notes

Melodizer Rock allows for only three notes to play simultaneously as it correspond to the notes of a triad, as described in section 2.1.4. Those two constraints only modify the cardinality of the variables of the `playing` array, in the current implementation it is forced to 3. For all $i \in [0, ..., k-1]$ where k is the size of the array, `min-sim` and `max-sim` being respectively the minimum and maximum number of notes that can play simultaneously:

$$min - sim \leq |playing[i]| \leq max - sim$$

Constraints from the Interface

The interface allows the composer to personalise the accompaniment. Some of these criteria are common for every part of the accompaniment:

1. **Chord key and chord quality** defines the chord, as described in section 2.1.4, that will play in that part of the accompaniment. Melodizer Rock allows the notes playing in the accompaniment to be in any octave of the basic chord. If $octaves(chord, quality)$ provides the set of triads corresponding to the octaves of the chord, for an array `playing` of size k, $\forall i \in [0, ..., k-1]$:

$$playing[i] \in octave(chord, quality)$$

2. **Minimum and maximum note length** constrain a pushed note to be pulled after the minimum note length, and before the maximum note length. For `min-length`, the equation can be written as follows, for arrays `push` and `pull` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \not\subseteq pull[i+j] \ \forall j \in \{1, ..., min\_length - 1\}$$

   For `max-note-length`, the equation is for arrays `push` and `pull` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \in \bigcup_{j \in \{1,...,max\_length-1\}} pull[i+j]$$

3. **Minimum and maximum pitch** limits the values that the `SetVars` can contain. It corresponds to limiting the domain of the variables in `push`, to be contained between `min-pitch` and `max-pitch` for an array `push` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \subseteq \{min\_pitch, ..., max\_pitch\}$$

4.3.2 Melody Constraints

As the melody uses `IntVar` variables for all three arrays, the constraints of Melodizer 2.0 [3] had to be adapted. Therefore, a new set of constraints are set to link `push`, `pull` and `playing`, as well as another set for the constraints updated in the interface. A last constraint posted on the melody corresponds to the requirement on the intervals, explained in section 2.1.3. The implementation of the constraints explained hereafter are available in appendix C.1.2.

Link push pull and playing

As for the accompaniment, the first step was making sure the initial problem was correctly stated. This is done by linking the three arrays. From equation 4.1 the following constraints were derived for arrays `push`, `pull` and `playing` of size k, $\forall i \in [1, ..., k-1]$:

1. The note playing at time i is either the same note playing at time $i-1$ or a note pushed at time i:

$$playing[i] = playing[i-1] \mid\mid playing[i] = push[i]$$

2. Either the note pushed at time i is played a time i, or no note is pushed:

$$push[i] = playing[i] \mid\mid push[i] = -1$$

3. Either the note pulled at time i was playing at time $i-1$ or no note is pulled:

$$pull[i] = playing[i-1] \mid\mid pull[i] = -1$$

4. If a note is pushed at time i, the note playing at time $i-1$ must be pulled:

$$push[i] \neq -1 \Rightarrow pull[i] = playing[i-1]$$

5. If no note is playing at time i, no note can have been pushed at time i, and the note playing at time $i-1$ must have been pulled:

$$playing[i] = -1 \Rightarrow push[i] = -1 \;\&\&\; pull[i] = playing[i-1]$$

6. If the notes playing at time i and $i-1$ are identical, then either the same note has been pushed and pulled, or no note has been pushed and pulled:

$$playing[i] = playing[i-1] \Leftrightarrow push[i] = pull[i]$$

In addition to that, since the previous constraints don't constrain the first index of the arrays, two other constraints are posted to do so:

1. No note can be pulled in the first index, as no note was playing before:

$$pull[0] = -1$$

2. A note that is pushed at time 0 must play at time 0:

$$push[i] = playing[i]$$

Constraints from the Interface

Similarly to what is done for the accompaniment, the melody has constraints on each block based on values from the interface. They were inspired from the optional constraints on *Blocks* from 3.4, but had to be adapted to `IntVarArrays`.

- **Chord key and chord quality** define the scale on which the notes are played. Melodizer Rock forces every note to belong to the scale corresponding to the chord and quality given in the interface. Considering *scaleset(chord,quality)* as the set of notes of the scale, this can be written, for an array `playing` of size k, $\forall i \in [0, ..., k-1]$:

$$playing[i] \in scaleset(chord, quality) \ || \ playing[i] = -1$$

- **Minimum and maximum note length** like for the accompaniment, constrain the distance between the moment a note is pushed and pulled, but also the minimal length of a rest. For the minimum, it is imposed for arrays `push` and `pull` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \neq -1 \Rightarrow pull[i+j] = -1 \ \forall j \in \{1, ..., min\_length - 1\}$$

  For rests, $\forall i \in [1, ..., k-1], \forall j \in [1, ..., min\_length - 1]$:

$$playing[i-1] \neq -1 \ \&\& \ playing[i] = -1 \Rightarrow playing[i+j] = -1$$

  For the maximum note length, it is imposed for arrays `push` and `pull` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \neq -1 \Rightarrow push[i] \in \bigcup_{j \in \{1, ..., max\_length - 1\}} pull[i+j]$$

- **Minimum and maximum pitch** limits the values that the `IntVars` can contain. As for the accompaniment, this can be translated to limiting the domain of the variables in `push`, to be contained between `min-pitch` and `max-pitch` or -1. For an array `push` of size k, $\forall i \in [0, ..., k-1]$, this is written as:

$$push[i] \subseteq (\{min\_pitch, ..., max\_pitch\} \cup \{-1\})$$

Intervals

The simplified principles that were defined in section 2.1.3 can be turned into constraints. Melodizer Rock implements only the first: the interval between two notes cannot be larger than a perfect fifth, so for a `playing` array of size k, $\forall i \in [1, ..., k-1]$, one can write:

$$|playing[i] - playing[i-1]| \leq 7 \text{ if } playing[i] \neq -1$$

4.4 Block-specific Constraints

Given what different blocks represent, it is expected that each block might have some specific constraints posted on it. The following sections give in-depth descriptions and explanations of these block-specific constraints.

A	A	B	A	B	A
A_0	A_1	B_0	A_2	B_1	A_3

Figure 4.4: *AABABA* structure, with A and B indexes

4.4.1 A and B-specific Constraints

Both A and B have one specific constraint that is posted on them. This constraint aims to impose a similarity between blocks of the same type forming the musical piece's structure. For example in an *AABABA* structure (cf. Figure 4.4), where the first A block's melody is A_0 and the first B block's melody is B_0, all following A blocks' melodies will be constrained to be similar to A_0, and equivalently for B and B_0. By default these similarities are both set to 50%. This creates horizontal relations between A and B blocks of the same type, by posting constraints on their variables, and by having shared variables. Constraining these melodies to be similar is done by imposing a similarity metric between different *push* arrays. Given two arrays $push_x$ and $push_y$ with respectively i and j elements, their resemblance (in percent) *sim* is computed as such:

$$k = \min(i, j)$$

$$sim = |\{push_x[l] : push_x[l] = push_y[l] \mid l \in [0, k-1]\}|/k$$

Constraining an A block to be at least `similarity-percent-A0` similar to A_0 is done with the use of the `cst-common-vars` function (cf. appendix C.2.2), and analogously for B with `similarity-percent-B0` and B_0. Considering the previous definition of k, $push_x$, $push_y$, and given a similarity *minsim*, this function posts the following constraints:

$$count = |\{push_x[l] : push_x[l] = push_y[l] \mid l \in [0, k-1]\}|$$

$$count >= \lceil minsim * k \rceil$$

This similarity constraint is then applied to all blocks A_m for $m > 0$ within the structure, and analogously for B_m blocks within the structure (in this case, constraining the similarity to B_0).

This similarity can also be done on a transposed piece of music. As A and B block don't yet allow for a transposition of a certain amount of semitones, it is imposed using the scale. It is done by constraining that a note on the scale of the initial melody, the one coming from A_0 or B_0, is transposed to the note at the same place on the scale of the block we want to constrain. Given the same x and i as before, $index_{scale}(chord, quality, note)$ is the index of a note on the scale defined by chord and quality. Then $chord_x$ and $quality_x$ are the chord and quality in which the melody of x is set. Finally t is the transposed melody with same length as x, and $chord_t$ and $quality_t$ define the scale to transpose to, it can be written $\forall j \in [0, ..., i]$:

$$index_{scale}(chord_x, quality_x, x[j]) = index_{scale}(chord_t, quality_t, t[j])$$

The similarity defined above is then posted on t rather than on x directly.

4.4.2 s r d and c-specific Constraints

s-specific Constraints

Constraints that are applied specifically to s blocks only include those which pertain to the source melody. The source melody or melodies can be given as voice object inputs to the *Rock* block, and are consequently used as source to set the notes in the intended s phrases. There are up to two potential s phrases in the *Rock* musical structure which can be set to a source melody. Each one corresponds to the s phrase of the first A or B block within the structure.

In order to set the `push`, `pull` and `playing` arrays of the s block to the notes represented by a voice object, said voice object must first be converted to an equivalent representation. That is done through the `create-push-pull-int` utility function (cf. appendix D.4.1), which takes a voice object as input and returns it in a `push`, `pull` and `playing` format.

Constraining s to these notes is fairly straightforward and is done as follows. Let the source melody be represented by $\{push, pull, playing\}_{source}$ arrays of i elements, and s by $push, pull, playing$ arrays of j elements. The constraints can then be written $\forall k \in [0, \min(i, j) - 1]$ as:

$$push[k] = push_{source}[k]$$
$$pull[k] = pull_{source}[k]$$
$$playing[k] = playing_{source}[k]$$

Which are written in Gecode in appendix C.2.1.

r-specific Constraints

The specificity of r lies in its similarity with s. The chosen similarity metric for musical phrases is based on how close their `push` arrays are. The similarity between two `push` arrays is computed in almost the exact same way as it is done for A and B blocks' `push` similarity (cf. section 4.4.1).

Constraining r to be at least `similarity-percent-s` similar to s is done with the use of the `cst-common-vars` function (cf. appendix C.2.2). The difference with the constraints defined in section 4.4.1 is posted on the transposition of the `push` array of s. A r block can be transposed according to a number of semitones. Given the x the melody to be transposed, and i its length, t the transposed melody of same length as x, and s the number of semitones to transpose, the transposition constraint becomes:

$$\forall j \in [0, ..., i],\ t[j] = x[j] + s$$

d-specific Constraints

In a similar fashion as is done for r, d also relies on the notion of similarity (or rather dissimilarity) with s. The dissimilarity metric between musical phrases is also computed based on their `push` arrays, and uses the same function `cst-common-vars` described in section 4.4.1. To impose a dissimilarity of *dissim* between two arrays, it was chosen to impose a similarity of $1 - dissim$.

c-specific Constraints

These are the constraints which are only applied to c blocks, also known as cadences in the context of Melodizer Rock. A cadence is defined by a chord progression, implying that such constraints are not only applied to the melody representing c, but also to its accompaniment. Several important things come into play when setting constraints for a cadence. The chord key, chord quality, and cadence choice (which is made through the interface, and is discussed in section 5.3.4) are what is needed to post constraints, in accordance with the musical definition of cadences.

Starting with the chord key, its importance is that it is the root note on which a degree **I** chord is built. Since chords of any degree require this information to be built, a succession of chords (as is done for cadences) evidently requires it too. The chord quality's necessity in posting cadence constraints comes into play when considering how triads in each quality are built. A detailed explanation of this construction can be found in section 2.1.4. As for cadence choice, its importance in posting cadence constraints is rather straightforward. Indeed, different cadences are defined by different successions of chord degrees. See section 2.2.2.

Now that the dependency between cadences and these three variables is clear, the actual constraints which have been implemented can be discussed further. As a generalisation, cadences in Melodizer Rock are only a succession of two chords. Depending on the value contained in the `cadence-type` attribute of c, `constrain-c` will impose the correct succession of chords on the accompaniment's `push` array.

The constraints posted to impose this chord succession are described mathematically below. Where the cadence is defined by a succession of chord degrees *succession* (array of two distances from the root note, in semitones), $push_{acc}$ is the accompaniment's `push` array of i elements, and *chords* is an array of two elements. Each of these elements is a set of notes representing a chord to be played . Note that i is a multiple of 16, therefore $push_{acc}$ always has an even number of elements.

$$push_{acc}[0] = chords[0]$$
$$push_{acc}[i/2] = chords[1]$$

Which are written in Gecode in appendix C.2.4

The elements in *chords* are built by formalising the theory explained in section 2.1.4, defining the triads to be played based on the chord's quality. These triads to be played are then represented by a succession of distances (in semitones) from the root note:

$$triad_{major} = [0, 4, 7], triad_{minor} = [0, 3, 7]$$
$$triad_{augmented} = [0, 4, 8], triad_{diminished} = [0, 3, 6]$$

Considering the root note's midi value is *root*, defining *chords* is done as follows

$$chord_0 = root + succession[0], chord_1 = root + succession[1]$$

$$chords[0] = [chord_0 + triad_{quality}[0], chord_0 + triad_{quality}[1], chord_0 + triad_{quality}[2]]$$
$$chords[1] = [chord_1 + triad_{quality}[0], chord_1 + triad_{quality}[1], chord_1 + triad_{quality}[2]]$$

Other cadence-specific changes have been added to c's melody to improve the overall conclusive feeling of cadences. These changes consist in multiplying the melody's `min-note-length` by its `min-note-length-mult` attribute, as well as ending the melody on the tonic. Given a *playing* array of i elements, and *tonic*. Given a function $octaves(tonic)$ which returns the tonic in all possible octaves, the constraint can be expressed as follows:

$$playing[i-1] \in octaves(tonic)$$

4.5 Solver

Melodizer Rock's solver was developed following the ideas of Melodizer 2.0 [3]. Melodizer objects contain a specification of the constraint problem, from which a Gecode CSP is created and used to create a melody. All of the constraints are then posted, following the recursive structure explained in section 4.2. After which the branching heuristic is defined, and finally, the search engine is built.

4.5.1 Constraint Satisfaction Problem

It was briefly explained in section 4.3 how the constraints are posted recursively. To be more precise, the function `constrain-rock` is called when starting the solver. It initialises the arrays `push`, `pull` and `playing`, as well as `push-acc`, `pull-acc` and `playing-acc` for the accompaniment. It then posts the constraints linking them, as explained in section 4.3.

This function then loops on the list of blocks forming the structure, and calls the `constrain-srdc-from-parent` function with parts of the six arrays (`{push,pull,-playing}-acc, push, pull, playing`) corresponding to the song's block.

This next function then posts constraints on the A or B block it was given as argument, then calls the functions `constrain-{s,r,d,c}`. These functions post the constraints explained in section 4.4.2, on s, r, d, and c. Figure 4.5 illustrates the followed path, when the solver is started.

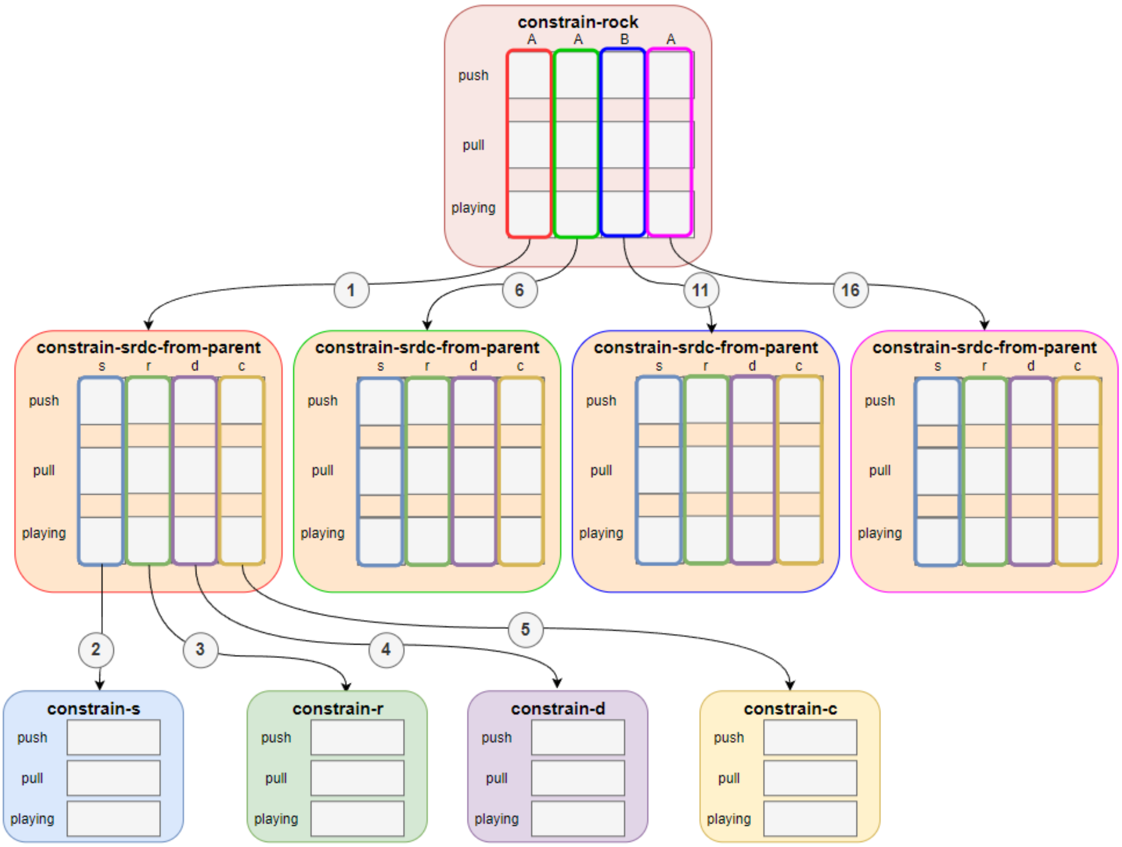


Figure 4.5: Followed path to construct the Constraint Satisfaction Problem

4.5.2 Search Engine

After the Constraint Satisfaction Problem is constructed, the search engine has to be built. The first step in building the search engine is to determine the branching heuristic that will be used. Then, the search engine's options have to be decided, as described in section 3.1.5. Finally, in order to have more varied results, the Branch and Bound algorithm explained in section 2.3.4 looked to be the most interesting exploration algorithm.

Branching Heuristic

The best branching heuristic was chosen through exploratory testing of the interface. As seen in section 2.3.3 and 3.1.4, different strategies are available and each has its advantages and drawbacks. During this testing, the strategy that seemed to come up with original solutions in least time functioned as follows:

1. Branch on the `push` array, as it is the most constrained of the three main variable arrays, by choosing the variable with the smallest domain and branching on a random value.

2. Using the same heuristics, branch on the `pull` array.

3. As now the push and pull arrays must be fixed, branch on the potential

remaining unfixed variables of `playing` with the same heuristics.

Branch and Bound

Now that the branching is decided, the used exploration method has to be determined. As previously explained, BAB allows for more varied solutions, by imposing a difference between two solutions. In order to achieve this, Melodizer Rock's solver receives a `percent-diff` parameter through the interface, representing the percentage of difference to be imposed between successive solutions.

It then uses the `constrain()` function of BAB (in Gecode), to impose that the number of variables with the same values as in the previous solution, has to be lower than 100−`percent-diff`. This algorithm was inspired by the one proposed in Melodizer 1.0 [3], by Damien Sprockeels, but adapted to Melodizer Rock's problem. This difference constraint is imposed on `playing`, as it regroups (in a sense) constraints posted on both `push` and `pull`, and is therefore more representative of the problem.

```
void WSpace::constrain(const Space& _b) {
    const WSpace& b = static_cast<const WSpace&>(_b);
    IntArgs bvars(b.var_sol_size);
    for(int i = 0; i < b.var_sol_size; i++)
        bvars[i]=(b.int_vars).at((b.solution_variable_indexes)[i]).val();
    IntVarArgs vars(b.var_sol_size);
    for(int i = 0; i < b.var_sol_size; i++)
        vars[i] = (int_vars).at((solution_variable_indexes)[i]);
    IntVar c(*this, 0, b.var_sol_size);
    count(*this, vars, bvars, IRT_EQ, c);
    rel(*this, c, IRT_LQ, b.var_sol_size * (100-b.percent_diff));
}
```

4.5.3 Search

After determining both the branching and exploration algorithm, the search options are given to the search engine, as explained in 3.1.5. Melodizer Rock imposes the search to use only one thread. Then, the composer can request a solution through the interface. The search engine will then explore the tree in search of the next existing solution. If it doesn't find one, or if the search was stopped through the interface, the search engine will return a `NULL` solution and the search won't be able to be continued. Otherwise, it will convert the obtained solutions for the six arrays into two voice objects, which in turn are combined into a poly object, as described in section 3.2.3.

Chapter 5

Melodizer Rock : User Interface

The user interface developed for Melodizer Rock is intended to be used by composers with minimal IT knowledge, therefore it aims to be as straightforward and intuitive as can be. In order to achieve this, the interface was built following a structure that closely resembles the hierarchical structure of rock music.

Each window the user has access to represents the editor of the block they are currently in. Editors are composed of various different panels, each serving a different purpose, and with which the user can interact with. Typically these panels aim to bundle actions related to each other, making navigating the interface intuitive.

In this chapter, panels will be referred to with Figure-specific regions containing roman numerals. Whereas interactive elements on these same Figures will be referred to with red numbers.

5.1 *Rock* Editor

The *Rock* editor (Figure 5.1) is composed of several different panels, firstly panel **I** which contains three different buttons the user can interact with. These buttons are used to define the structure of the music which the composer wishes to create.

Meaning that they can choose to build a structure based on blocks A and B such as $AABA$ or any extension of it. Adding an A block is done through button 1, and B through button 2. Clearing the structure that has been built by the composer can be done by interacting with button 3. This will allow the user to input a new musical structure.

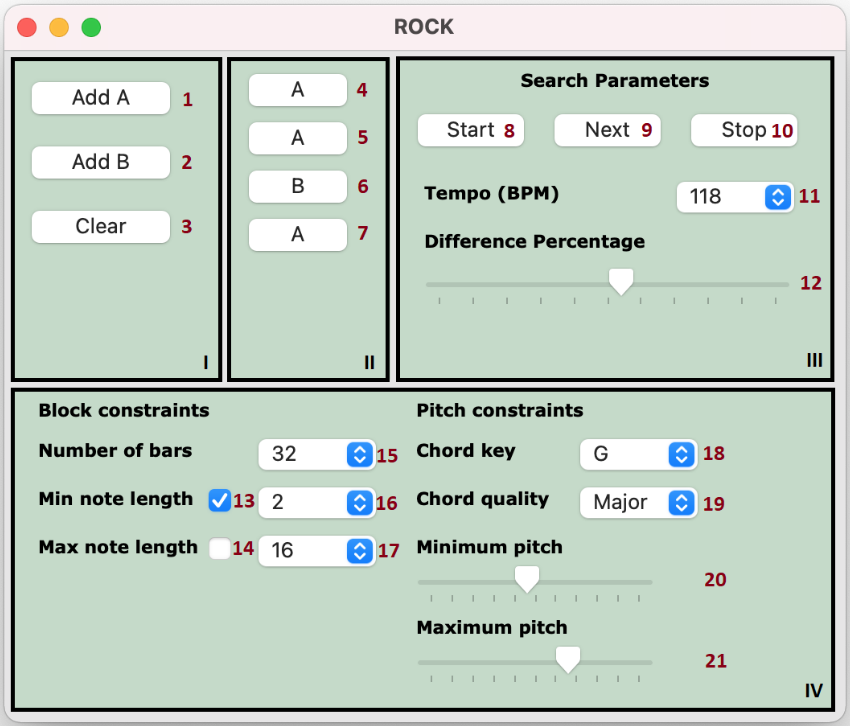


Figure 5.1: *Rock* editor, split into its various panels

Next panel contained within the *Rock* editor is panel **II**. This panel displays the structure that has been created by the user through interaction with panel **I** and buttons 1 through 3. Each block of the considered musical structure has an associated interactive button. Interacting with any of these buttons in the panel will open up their respective editor. Buttons $4, 5, 7$ open their *A* editor, and button 6 opens the *B* editor. These opened editors are further described in the upcoming section 5.2.

The remaining two panels contained in the *Rock* editor are panels **III** and **IV**. In order to define the musical piece and define its constraint problem, there is some initial information which has to be given by the composer. All this information is what is managed in panel **IV**.

The composer has control over the information relating to the musical piece they want to create. The information which has to be given by the composer, in order for the solver to function as intended, is: the *Chord key*, *Chord quality*, *Number of bars*. These can be set by interacting with elements $18, 19, 15$. All the other information, which can be modified through interaction with the various elements of the panel, will be translated into additional constraints (e.g. interacting with elements $20, 21$ will constrain the pitch range of the entire musical piece to fall within the specified values). All of the values chosen by the composer within this panel will constrain the entire musical piece.

Finally, panel **III** is the way the composer can interact with the solver. This panel contains three buttons named after their actions, 8 creates the problem and sets the constraints based on the information given in the panel **IV**, then starts the search. Button 9 gives the search's next solution, and 10 stops the search. Element 11 modifies the tempo of the solutions, meaning that the generated `poly` object will have this tempo. And element 12 constrains the search's next solution to differ from the previous solution by at least the specified percentage.

5.2 A and B Editors

Both A and B editors have identical interfaces and functionalities, thus only one figure (Figure 5.2) is provided as reference throughout this section.

Panel **I** serves as an interface to interact with the *srdc* defining the current block, it is in a way similar to the *Rock* editor's panel **II**. Where by interacting with buttons $1, 2, 3, 4$ contained within the panel, the user can open the respective s, r, d, and c editors.

Panel **II** is used to indicate whether changes done in the *Rock* editor, and in other editors of the same block type, should change values in the current A or B editor. Meaning that checking the check-box of element 5, any changes to *Rock* (representing the whole music) will be propagated to the A or B block the composer is currently in (representing this *srdc* portion of the music). Checking element 6 implies that if the composer is in an A block, then any change in other A blocks of the structure, will be propagated to the current block. This is done analogously with B, if the composer is in the editor of a B block.

Panel **III** is effectively the same panel as *Rock*'s panel **IV**. With the difference being, any constraint set in this panel will by default only affect the current block and it's children from figure 4.3 (i.e. constraints in panel **III** of an A or B editor will be propagated to the s, r, d, and c which it represents). This panel is slightly different depending on the place of the block in the overall structure. If it isn't the first of its type, then the elements 7 through 11 are replaced by a slider, controlling the resemblance with the first block of the same type in the structure.

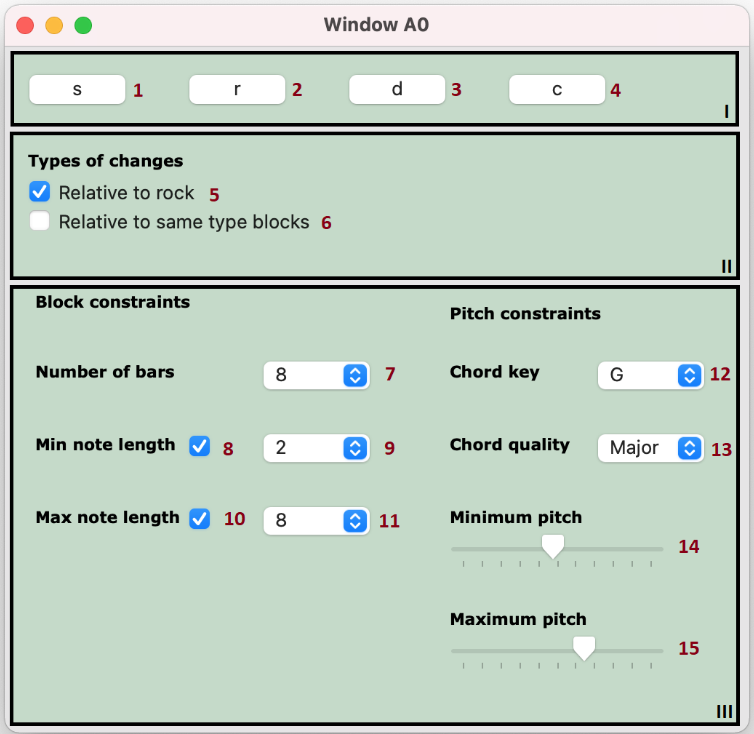


Figure 5.2: A editor, split into its various panels

5.3 s, r, d, and c Editors

Editors s, r, d, and c (Figures 5.3, 5.4, 5.5, 5.6) are all quite different but have one common panel. Each of the s, r, d, and c editors' panel **I** is identical. Element 1 allows the composer to choose the number of measures which the current block will be made of. Elements 2 and 3 are used to set the minimum note length for the current block, whereas elements 4 and 5 are used to set its maximum note length. Elements 6 and 7 are used to restrict the current block's pitch range. All additional panels contained in these editors are further described in the following sections.

5.3.1 s Editor

The s editor is shown in Figure 5.3, and gives the user control of the accompaniment in panel **II**. This panel allows the composer to modify s's accompaniment note length through elements $8, 9, 10, 11$. It also allows for modifying chord key and chord quality independently from s itself, through elements 12 and 13.

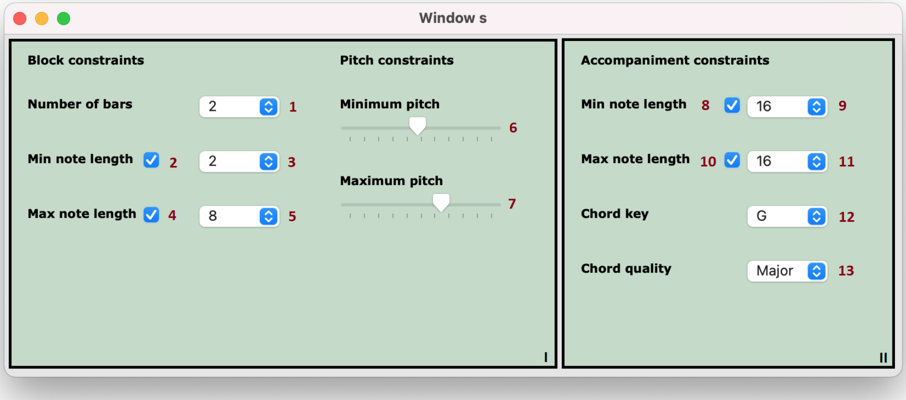


Figure 5.3: s editor, split into its various panels

5.3.2 r Editor

The r editor is shown in Figure 5.4, and contains the same panels as s, to which it adds panel **III**. The value set by the *"Similarity with s block"* slider (element 14), constrains r to resemble s. The value on the far right of the slider implies a 100% similarity, meaning r will be the same as s, and the far left is 0%. As for element 15, it allows the composer to choose r's semitone transposition from s.

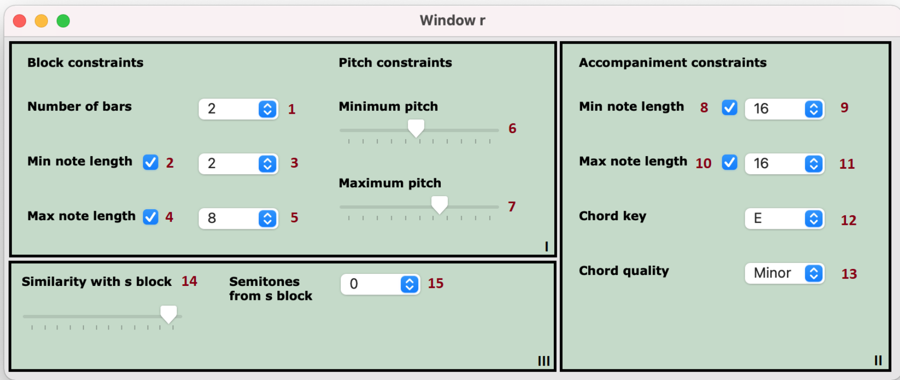


Figure 5.4: r editor, split into its various panels

5.3.3 d Editor

The d editor is shown in Figure 5.5. As can be seen, it is very similar to r's editor and only differs in panel **III**. The value set by the *"Difference with s block"* slider (element 14), constrains d to be different from s. The value on the far right of the slider implies a 100% difference, meaning d will be completely different to s, and the far left is 0% meaning they are the same. As for element 15, it allows the composer to choose d's semitone transposition from s.

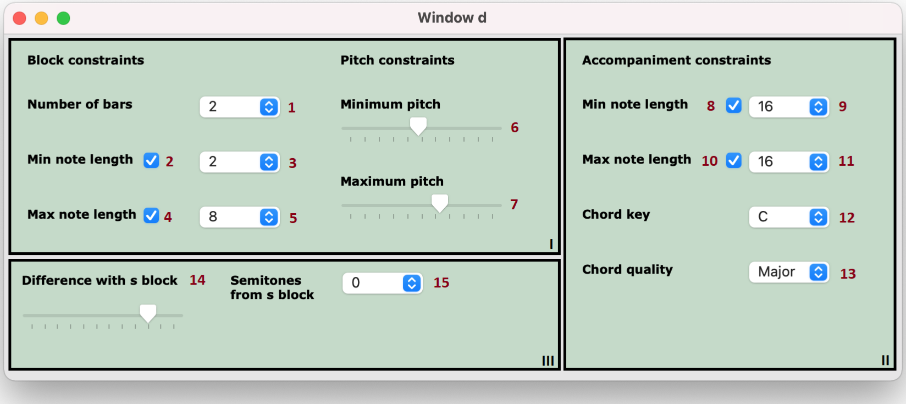


Figure 5.5: d editor, split into its various panels

5.3.4 c Editor

The c editor is rather bare-bones, as can be seen in Figure 5.6. Panel **II** contains a *"Cadence choice"* (element 8) drop-down menu giving the composer a choice between multiple cadence types.

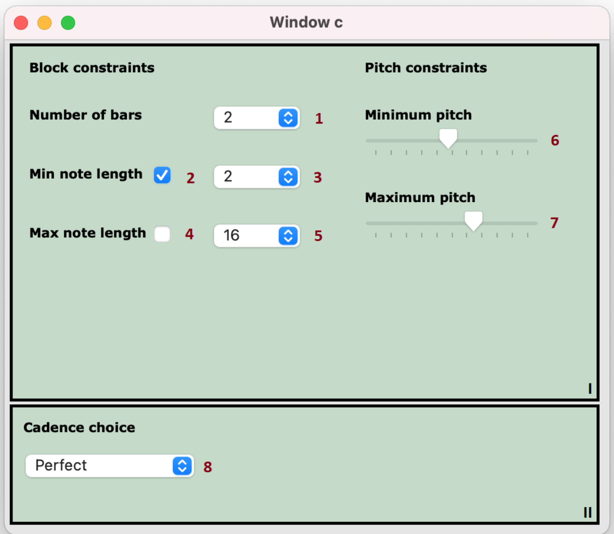


Figure 5.6: c editor, split into its various panels

Chapter 6

Composing with Melodizer Rock

This chapter is first and foremost destined for composers, and aims to give examples on how to use Melodizer rock to compose Rock music. To this effect the terminology used will be musical rather than scientific when possible. Before reading this chapter, one should have followed the steps to install all the necessary tools as explained in appendix A, and familiarised themselves with the appendix B tutorial.

As a composer using Melodizer Rock, your creativity and decisions occur through the interface described thoroughly in chapter 5. The first step is deciding on the structure to be used for the musical piece that you wish to create, typically this would be $AABA$ but could be extended to some of its variations such as $AABABA$ (discussed in depth in section 2.2). Then by selecting the number of measures for this musical piece. This two-step process is the strict minimum that must be done in order to compose music with Melodizer Rock.

In the following sections, several progressive examples and use-cases of Melodizer Rock will be presented and go over the composition process from a user's standpoint.

6.1 A Simple A Block

The first example will explore the solutions found with a single and simple A block. It doesn't take a source melody as input, and will only have a few simple constraints. On the *Rock* interface, the constraints will be:

- **Number of bars:** 4
- **Min note length:** not checked, allows the shortest note possible
- **Max note length:** also not checked, allows the longest note possible
- **Chord key:** default, C key
- **Chord quality:** default, Major

- **Minimum pitch:** increased to slightly below half of the slider
- **Maximum pitch:** lowered to slightly above half of the slider

After setting the search to a tempo of 100 and the slider of difference percentage to the maximum (far right), the interface should look like this:

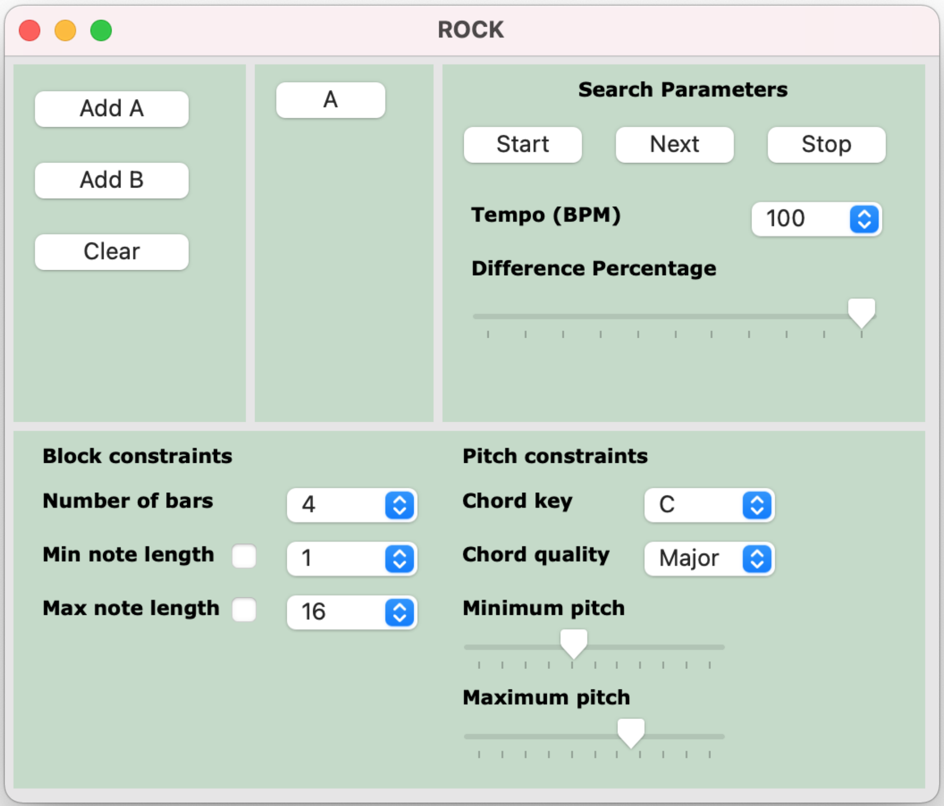


Figure 6.1: Rock interface of an example with a single A block

No other block is changed for this example. The solver can now be started. This is done through the interface by pressing the start button, that builds the CSP, then the next button, that searches for the next solution. Melodizer Rock then displays a first solution, shown in figure 6.2. Another press of the next button gives another solution shown in figure 6.3.

As can be seen, the solutions which were found use the shortest possible note first, and use rests to allow large leaps in the song. Imposing more constraints might help with rendering more harmonious results.

Figure 6.2: First solution to an example with a single *A* block

Figure 6.3: Second solution to an example with a single *A* block

6.2 An A Block and a B Block

What about a longer song using both *A* and *B* block types? The structure is cleared and both an *A* and *B* blocks are added. The constraints in the *Rock* editor are set to:

- **Number of bars:** 16
- **Minimum note length:** checked and set to 2, which corresponds to an eighth note

- **Maximum note length:** not checked
- **Chord key**: E
- **Chord quality**: Minor
- **Minimum pitch**: as for the previous example, the slider is set slightly below half
- **Maximum pitch**: as for the previous example, the slider is set slightly above half

The search parameters from the previous example are maintained. The *Rock* editor should now look like in figure 6.4.

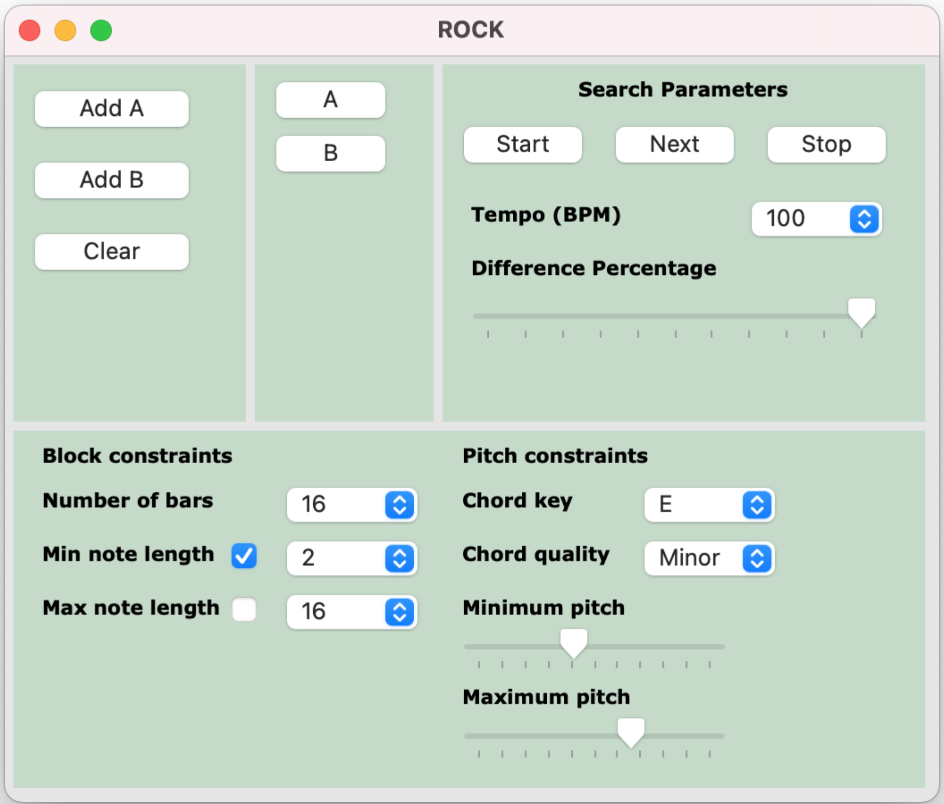


Figure 6.4: Rock editor of an example with an A block and a B block

Let's also set more constraints in the blocks. Starting with the A block, its r block is changed such that its similarity with s is 100%, and its editor should now look like figure 6.5

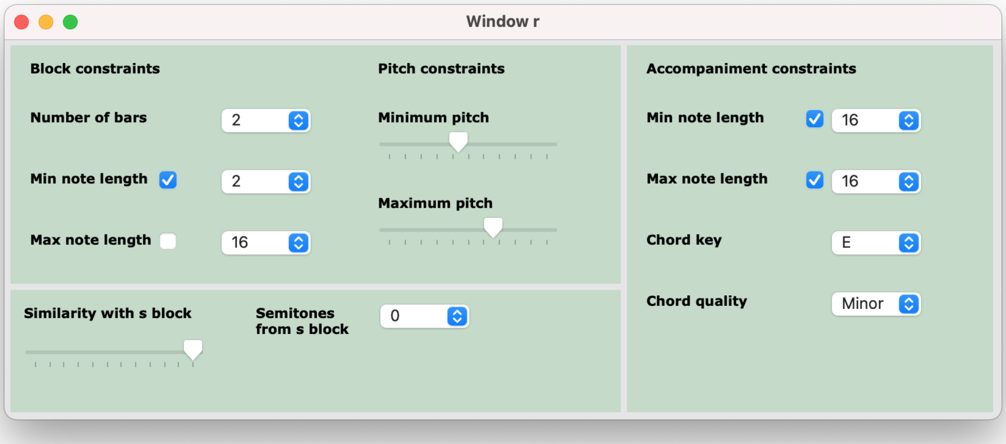


Figure 6.5: r editor of an example with an A block and a B block

The d block is also changed to impose more disruption in the song. The accompaniment's minimum note length is set to 4, which corresponds to a quarter note, and its chord will be set to a G Major. The editor is as shown in figure 6.6.

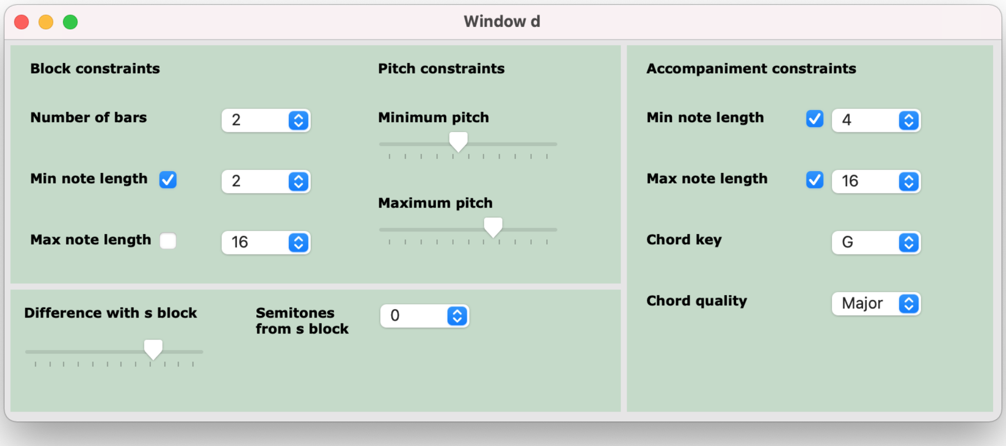


Figure 6.6: d editor of an example with an A block and a B block

For the B block, its d sub-block will also be updated to allow more disruption. This time, the accompaniment is set to have a minimum note length of 8, which corresponds to a half note, and its key is set to a D Major. Its slider of difference with s is also lowered to around half way. B's d editor now looks like figure 6.7.

The search can now be launched, the same way it was done in the first example. Solutions can be obtained with 100% of difference, the first one being shown in figure 6.8.

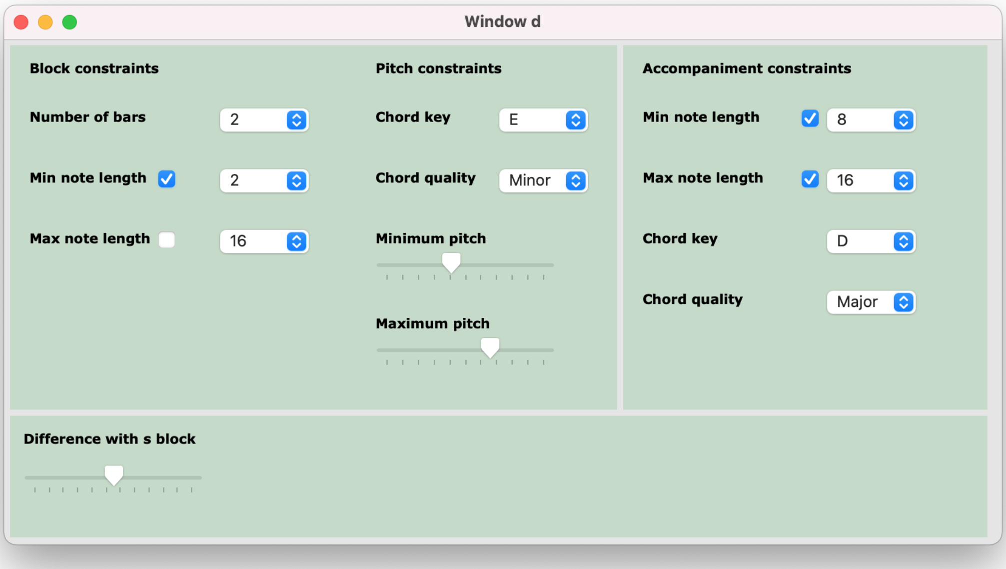


Figure 6.7: d editor of an example with a block A and a block B

6.3 A Source Melody on Two *A* Blocks

It is clear given previous examples that, without input melodies, Melodizer Rock does not always produce harmonious songs. But as was explained in section 4.4.2, two source melodies can be taken by a *Rock* block, to set the *s* phrase of the first *A* and *B* blocks. As other blocks of the same type have a similarity percentage with this block, they also use this source melody.

Let's try it out with two *A* blocks, and an input melody for the first *A* block. The existing song in Figure 2.12 was reproduced as a voice object in OpenMusic, and given as a source melody to the *Rock* block through its third input (starting on the left). The result that should be observed in the corresponding patch is shown in Figure 6.9.

The interface must now be modified to approach the score's constraints. It has to be noted that an exact reproduction of the song won't be possible, as Melodizer Rock only accommodates a subset of all possible constraints. And as only the *s* block of the score is given in input. The *Rock* block is set up with the following parameters. Its editor will not be shown, as it is similar to the previous examples.

- **Number of bars:** 16, as blocks *s*, *r*, *d* and *c* are each two measures long.
- **Minimum note length:** checked and set to two
- **Maximum note length:** not checked
- **Chord key:** F, as the key signature indicate the scale to be a F Major

Figure 6.8: First solution of an example with an A block and a B block

- **Chord quality:** Major
- **Minimum pitch:** the slider is set just below half way
- **Maximum pitch:** the slider is set just above half way
- **Tempo:** 96, as indicated on the score
- **Difference percentage:** around 50% as the use of a source melody prevents the 100% difference between solutions.

The sub-blocks can now be set up. It is important to note that the chords available for the accompaniment are quite limited, and that this song uses other types that will be explained in section 7.1.4. In the first A block, the r sub-block is

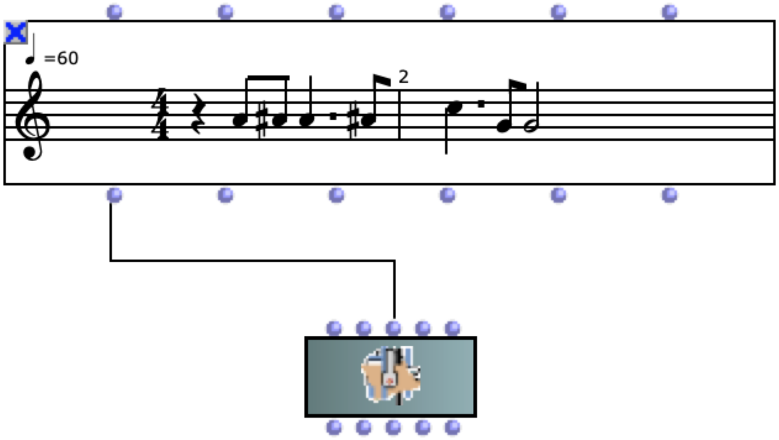


Figure 6.9: Connection of A's source melody to the *Rock* block

modified to have a similarity with the s block of 100%, which does not correspond exactly to the score, but is approximated out of convenience. The score also has some sort of transposition 4 semitones lower, which is set in the *"semitones from s"* block parameter. Finally, the accompaniment has a minimum note length of 8. The r editor should now look like Figure 6.10.

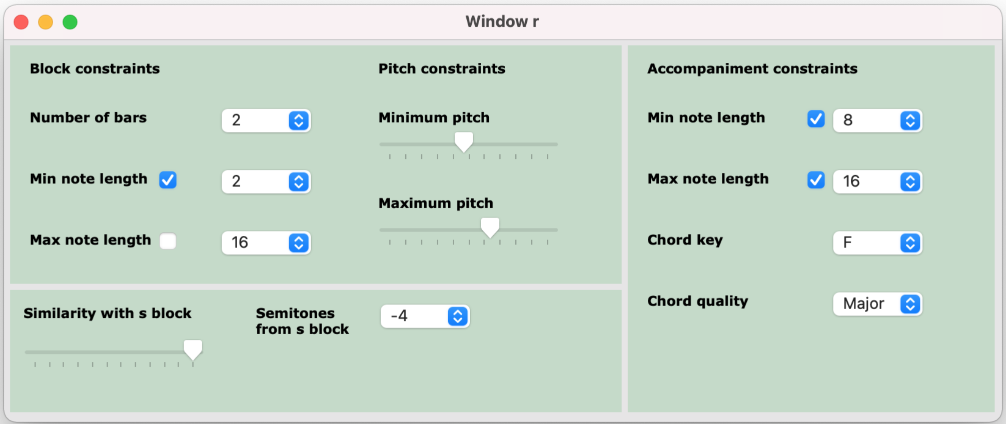


Figure 6.10: r block of the first A block, for an example with two A blocks and a source melody

The d block of that same A block is also changed to respect the score. Its accompaniment is set to have a minimum note length of 8, the other parameters stay unchanged. The resulting editor is shown in Figure 6.11.

Lastly, the second A block is updated to have a similarity of around 65% with the first A block, as shown in Figure 6.12.

After launching the search, two successive solutions can be obtained. The first

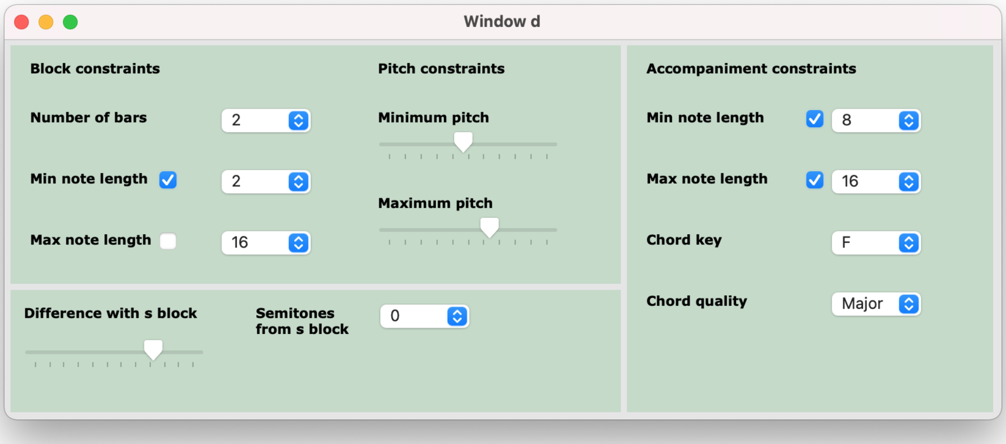


Figure 6.11: *d* block of the first *A* block, for an example with two *A* blocks and a source melody

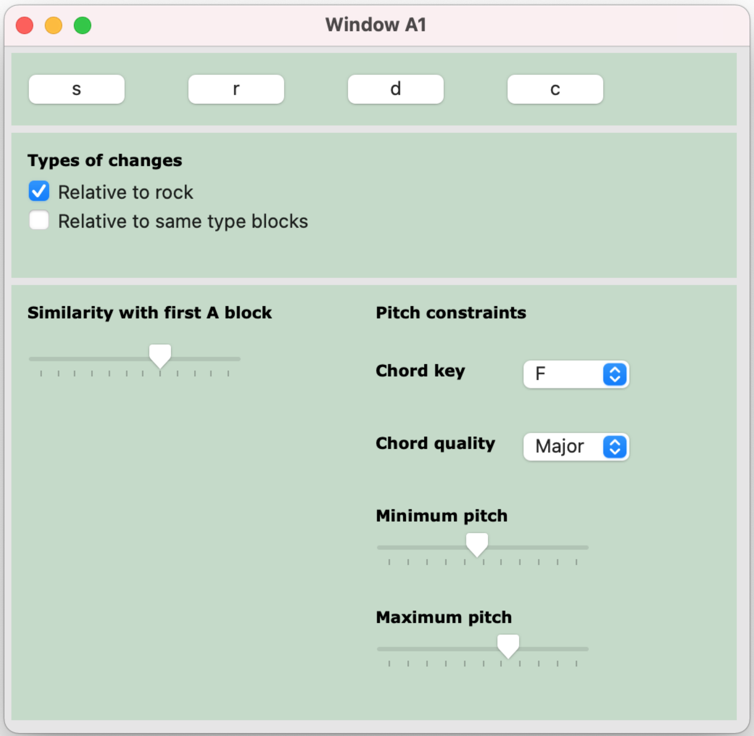


Figure 6.12: Second *A* block for an example with two *A* blocks and a source melody

one is showed in Figure 6.13.

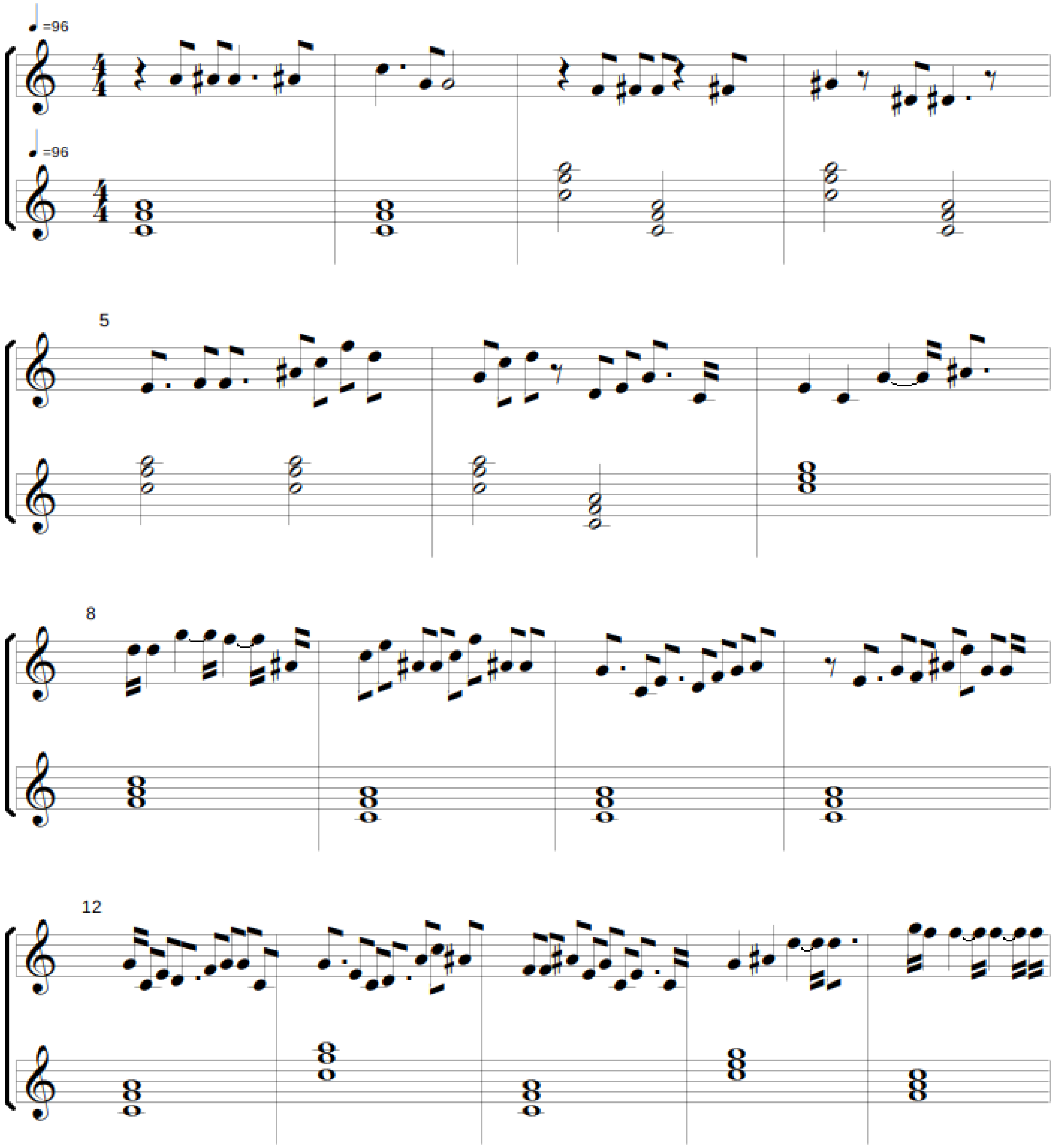


Figure 6.13: First solution of an example with two A blocks and a source melody

6.4 A Full Song Form

Now that Melodizer Rock showed what it was capable of with rather simple song structures, it can be tested to produce a full song on its own. As a full song implies a lot more variables, the problem will be further constrained to obtain a solution in order to obtain solution within a couple seconds.

The solver will be run on a classic *AABA* form, with one bar per s, r, d, and c block which is low compared to a real rock song. But it implies less variables to branch on. Therefore, the parameters of *Rock*'s editor are the following. Again, its editor is not shown as it is similar to the previous examples.

- **Number of bars:** 16

- **Minimum note length:** checked and set to four, to simplify the search
- **Maximum note length:** not checked
- **Chord key:** G
- **Chord quality:** Major
- **Minimum pitch:** the slider is set above a third of the way
- **Maximum pitch:** the slider is set below two thirds of the way
- **Tempo:** 100
- **Difference percentage:** 100%

It is the perfect occasion to have a little fun with the constraints proposed by Melodizer Rock. Starting with the first A's r sub-block, the slider for its similarity with the s block is set to 100%, and the transposition from s is set to two semitones. The accompaniment's minimum note length is also changed and set to 8. r's editor should now look like Figure 6.14.

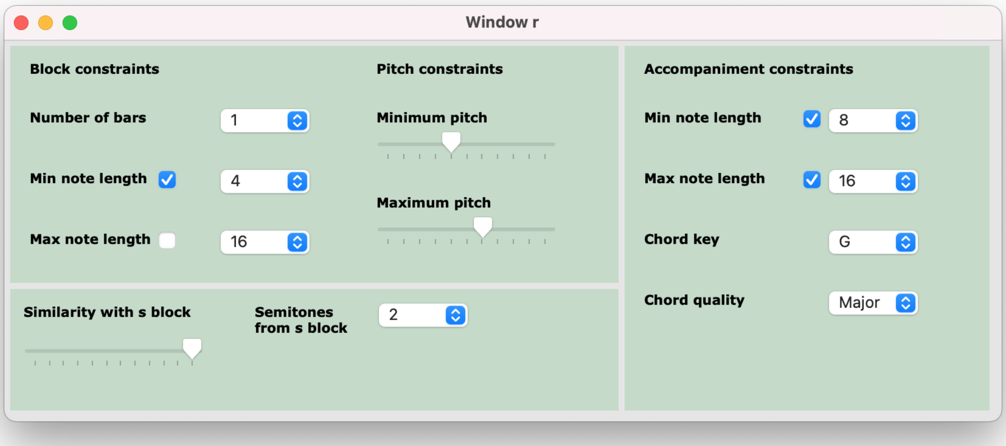


Figure 6.14: First A block's r editor, in an example with an $AABA$ structure

In the same A block, its d sub-block is changed to sound more disruptive. The accompaniment's minimum note length is set to 4, and the slider for its difference with the s phrase is set around 50%. Its editor is not shown.

As this is only the first block of the structure, a perfect cadence to end it might sound too definitive. Therefore, the *"cadence choice"* in the c sub-block of the first A block is set to Plagal. The c editor should now look like Figure 6.15.

Now that the first A block is set, the second can be changed based on the first. To avoid too long of a search, and because a rock song usually repeat its first A

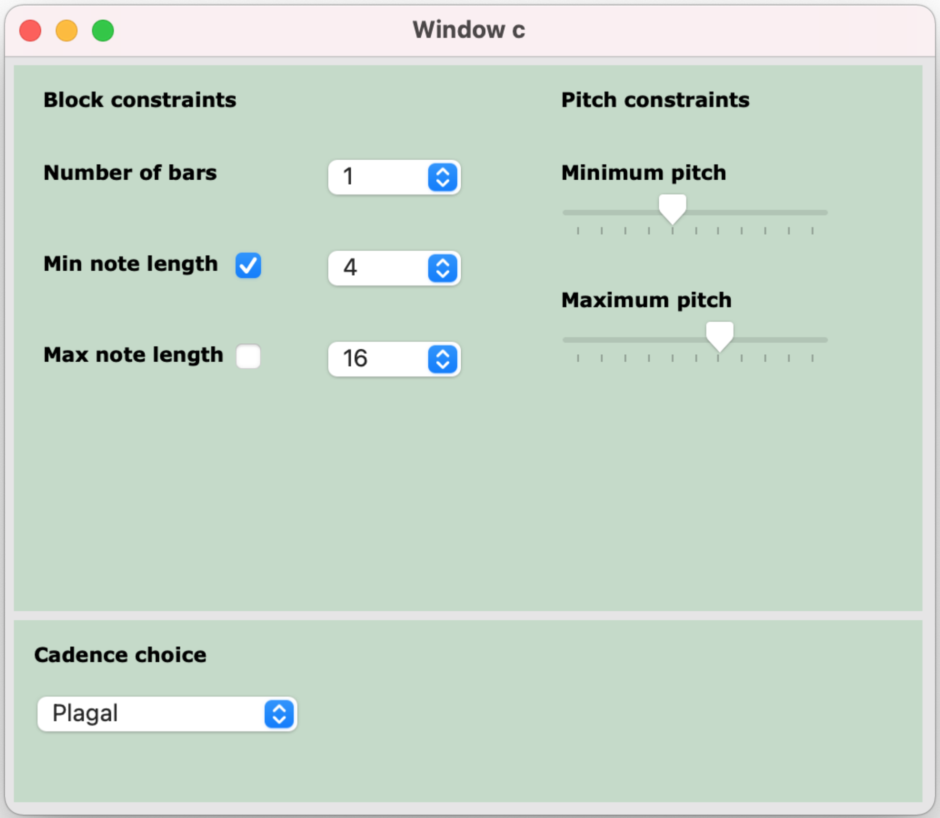


Figure 6.15: c editor of the first A block, in an example with an $AABA$ structure

almost exactly, the slider of resemblance with the first A is set to 100%. The only other thing that can be, and that is changed for that block, is the cadence choice. As it is imposed on the accompaniment, and the resemblance is imposed on the melody, it will not cause any conflicting constraints. This A block's ending being the middle of the piece, a semi cadence might be appropriate.

The B block can have different constraints, as it has no resemblance with another block, and is thus at no risk of causing conflicting constraints. This part of the song corresponds to a bridge, which is a part supposedly quite different from the rest of the song. Therefore, it will be allowed to go faster by imposing a minimum note length of 2. The resulting editor is shown in Figure 6.16.

Its r sub-block will not be changed, keeping a slider of around 50% of resemblance. On the contrary, its d sub-block will be slightly more varied. First, it will impose a transposition from the s sub-block of -2 semitones, that is, it imposes the difference to be set with a melody two semitones lower. Then its accompaniment is set to have a minimum note length of 8, and to be in D major. Its editor should now look like Figure 6.17. The c sub-block will also be changed to impose a Plagal cadence, as the

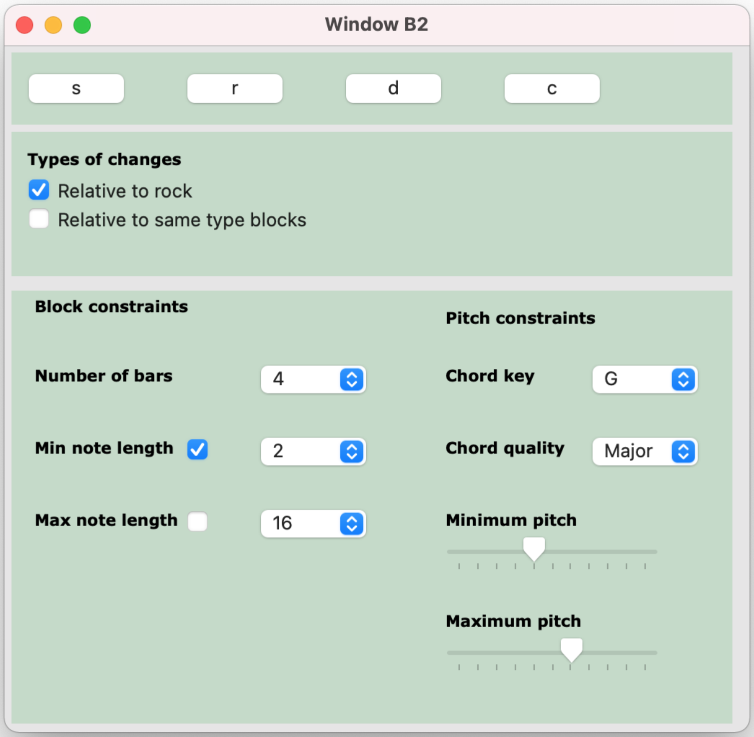


Figure 6.16: Editor of the B block in an example with an $AABA$ structure

song is not yet ended.

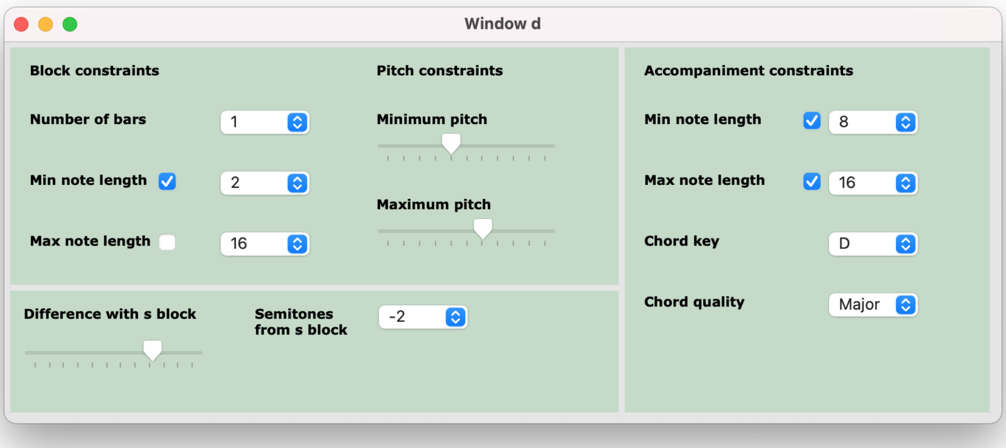


Figure 6.17: B's d editor, in an example with an $AABA$ structure

Finally, the last A block will be slightly modified. The only change from the default parameters imposed on this block is on the r sub-block, where its resemblance with the s sub-block is set to 100%.

After launching the search, two solutions can be obtained quite fast, the first one appearing within 5 seconds after the press of the next button. The most interesting solution for this example is the second, showed in Figure 6.18.

Figure 6.18: Second solution of an example with an *AABA* structure

6.5 A Full Song Form with Two Source Melodies

For this last example, Melodizer Rock is put to the test with a full song, with two source melodies given as input. A song which suits the *AABA* structure nicely, is *Every Breath You Take* by The Police [12]. The full score of that song is available in appendix E. The source melodies are connected to the *Rock* block, through its third and fourth input (from the left), as shown in figure 6.19.

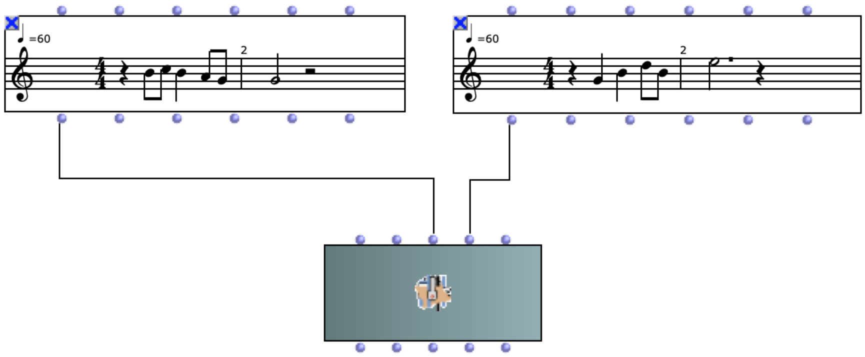


Figure 6.19: Connection of the source melodies, for an example with an $AABA$ structure

The *Rock* block is set to the following parameters, which leads to the editor shown in figure 6.20.

- **Number of bars:** 16
- **Minimum note length:** checked and set to two, as it is the shortest note seen in the score
- **Maximum note length:** not checked
- **Chord key:** G, as it corresponds to the key signature on the score
- **Chord quality:** Major
- **Minimum pitch:** the slider is set just below the half
- **Maximum pitch:** the slider is set just above the half
- **Tempo:** 118, because it corresponds to the tempo given by Drew Nobile [4] for this same song
- **Difference percentage:** 50%, because of the source melodies

The first A block will also constrain the maximum note length to be 8. Then its sub-blocks are changed, such that the r sub-block resembles the s sub-block slightly less than 100%. And its accompaniment is set to be an E Minor chord. The d sub-block is also changed so that the accompaniment is a C Major chord. The two other A block are set to have a 100% resemblance with the first A, and the accompaniment constraints for both r and d sub-blocks are the same as for the first A.

The B block can now be changed. As is done in the A blocks, it will impose a

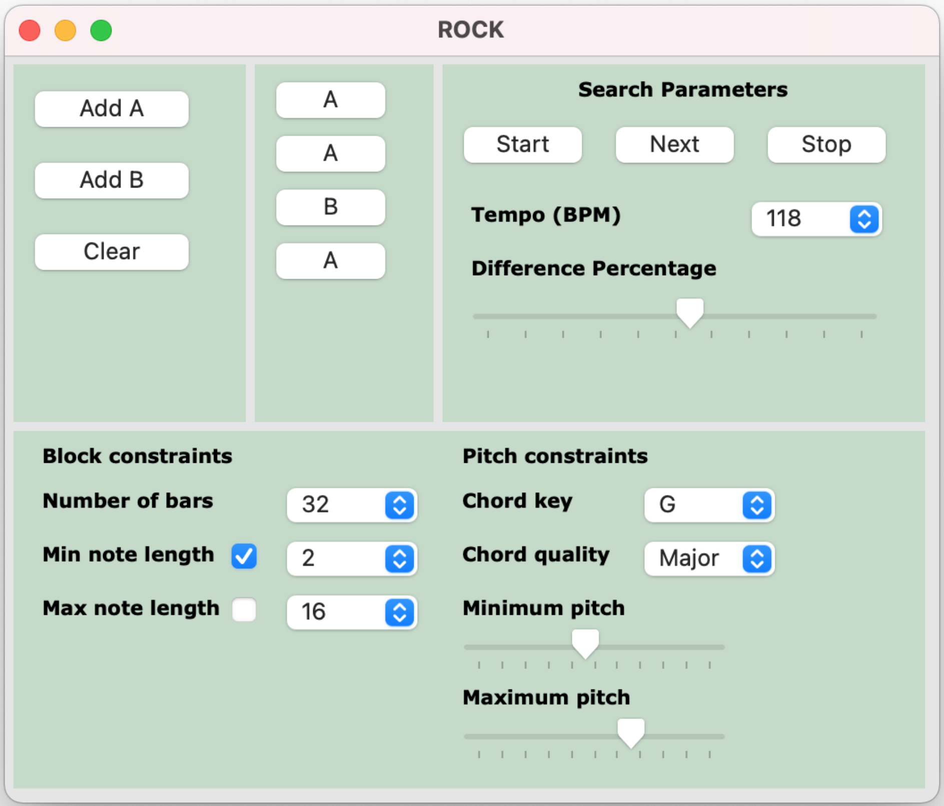


Figure 6.20: *Rock* editor, for an example with an $AABA$ structure and two source melodies

maximum note length of 8. The r sub-block is set to have a similarity with the s sub-block of around 95%, and its accompaniment is constrained to be in A Minor. The d sub-block is set to have a dissimilarity with the s phrase of around 50%, as well as a transposition of two semitones, and an accompaniment in A Major.

Now that all blocks have been set to parameters resembling those of the song, the search can be launched. It can obtain multiple solution, the first one being shown in Figure 6.21 and 6.22.

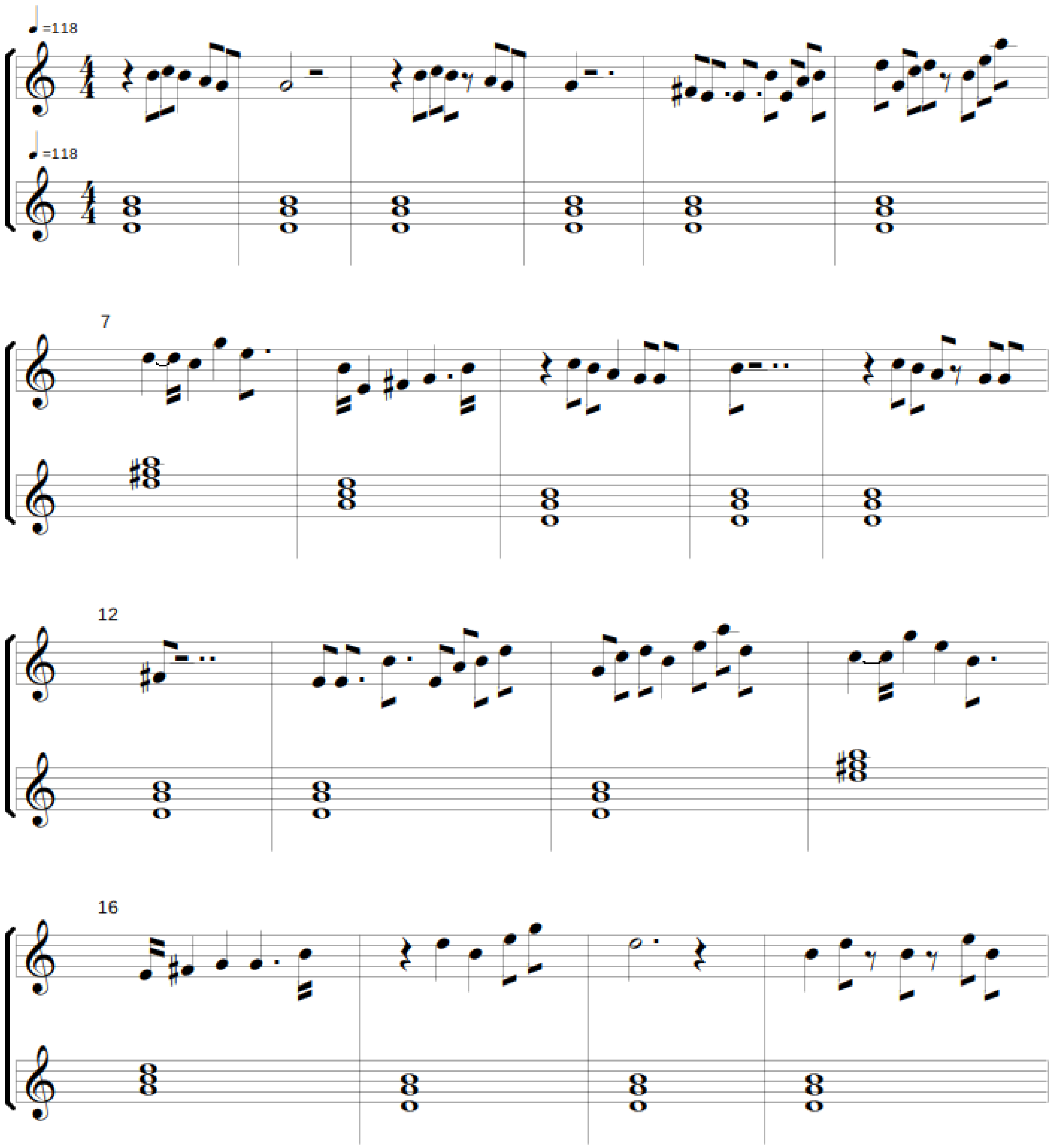


Figure 6.21: First page of the first solution given by Melodizer Rock, with the inputs of *Every Breath You take* [12] for an *AABA* structure

Figure 6.22: Second page of the first solution given by Melodizer Rock, with the inputs of *Every Breath You take* [12] for an $AABA$ structure

Chapter 7

Future Works

This chapter aims to discuss potential improvements which could be made to Melodizer Rock. Several approaches are suggested, the first of which is diving deeper within the rock genre, the second is to expand Melodizer Rock to new musical genres, and finally, using Gecode without going through GiL.

7.1 Diving Deeper Within Rock

Diving deeper within this genre can be done in many different manors. One could explore alternative structures to $AABA$ and its extended forms, or constrain and build s, r, d, and c blocks differently. Additionally, the overall melodic line and accompaniment could both be improved.

7.1.1 Other Structures than $AABA$

Much of the discussion and insight given in this chapter stems from Drew Nobile's Thesis [4]. Three song structures are present in his thesis, $AABA$ and *srdc*, Verse-Prechorus-Chorus, and Verse-Chorus. $AABA$ and *srdc* is present in Melodizer Rock, and exploring other forms might be interesting leads.

Expansion of srdc into Verse-Prechorus-Chorus

Expanding the *srdc* phrase structure into Verse-Prechorus-Chorus was thought of during the early 1960's when the presence of a chorus grew ever so popular. A general *srdc* structure typically spans over one verse, whereas the expanded Verse-Prechorus-Chorus spans over three verses. By expanding *srdc* as such, the verse now comprises s and r, the prechorus corresponds to d, and c is the chorus. It is also typical for the verse, prechorus and chorus to have roughly equal lengths.

This model became even more popular in the 1980's, and implementing the Verse-Prechorus-Chorus structure within Melodizer Rock would enable the creation of musical pieces following this style. Drew Nobile discusses this structure very thoroughly in his thesis [4], and should be taken as reference if this improvement suggestion is pursued.

Verse-Chorus

Verse-Chorus forms give rise to different harmonic-melodic layouts, and might be an interesting area to explore. Drew Nobile discusses this structure very thoroughly in his thesis [4], and should be taken as reference if this improvement suggestion is pursued.

7.1.2 Alternative Take on *srdc*

As the way *srdc* was implemented in Melodizer Rock only represented one possible vision, potentially richer and more interesting implementations exist for composers. For example, one of the simplified assumptions made in this implementation is that cadences only span over c. However, even though c contains the cadence, often times this cadence starts in d. The following sections contain a short discussion over some of the possibly interesting variants for each s, r, d, and c block.

Source Melody

A possibly interesting suggestion to explore could be to use source melodies as inspiration rather than just copying them. Meaning that a source melody could inspire an s phrase, and not set all of its notes to it, for example by setting half of the notes to be from the source melody. Another suggestion would be to use source melodies as rhythm or pitch-setting tools. A composer could give a source melody as input, and choose for its rhythm or pitch sequence to be used instead of the whole melody.

Resemblance

Some songs might use different variations on the s phrase of a block to obtain the r or d blocks. Someone wanting to improve Melodizer Rock could study a larger range of rock songs to propose more variations on the stated phrase.

Disruption

Improving the composer's control over the disruption (d), and making this disruption lead into the cadence better, could be worthwhile additions to Melodizer Rock.

New and Improved Cadences

Melodizer Rock has a rather primitive range of cadences available to the composer, which could be expanded. As mentioned in section 2.2.2, the cadence choices are: Perfect, Plagal, and Half. This could be improved by adding cadences such as Deceptive, Evaded, Imperfect, Burgundian, Lydian, Inverted, etc. to Melodizer Rock's capabilities. Another improvement which can be made to cadences within Melodizer Rock, is allowing progressions of more than two chords for the accompaniment.

Cadences in Melodizer Rock use the simplified assumption that they occur entirely in c, however some models discussed in Figure 2.13 show that the cadence is sometimes already present in d. Implementing these various models and therefore considering that the cadence could be present over multiple blocks, could lead to interesting results.

Another change that could be made to Melodizer Rock regarding cadences, is to select default cadences which are appropriate for the position of the considered block within the music. For example, a perfect cadence might not be suited to an early portion of a song, but might be good to end the song.

7.1.3 Improve the Melodic Line

The melodic line obtained when using Melodizer Rock does not always sound harmonious. This was thoroughly shown with the examples of Chapter 6. Different improvements can be made on this melody.

Contour

Music theory for Dummies [5] describes contours often used in the composition of the melody. This contour is the shape of the pitch's travels, its upwards and downwards flow. Different contours can make the song sound more tense or more lively, more melancholic or happy.

- The arch: the melody' pitch increases from a low point, to a high point, then gradually goes back down. The pitch increase results in an increase in tension, therefore when the pitch goes down the tension releases.
- The wave: it can be considered as small consecutive arches. The melody repeatedly goes up and down.
- The inverted arch: as its name suggest, this contour starts by going from a high point to a low point and then back up again. Therefore, it starts by sounding relaxed and then increases the tension.
- The pivotal: a pivotal melody line mainly pivots around the central note of the piece. It acts much like a wave, except that the movement is minimal and returns to the central note.

Handling Rests

The examples of Chapter 6 made it clear that Melodizer Rock's search engine tends to favour rests, as small as possible, to allow for greater intervals. Singing those intervals might not be a realistic expectation. Therefore, some constraints could be added to smooth over those imperfections:

- One could try to limit the interval of notes surrounding the rest. The problem with this idea is that it requires knowledge of start and end of a note, to be able to point out which note precedes and succeeds a rest.

- Melodizer 2.0 [3] had introduced an interesting constraint to quantify the number of rests in a block and their distribution. This could be reused and adapted for the melodic line of Melodizer Rock, allowing the composer to have more control over the amount of rests they want, as well as their location in the song.

Scales

As explained in section 2.1.3, Melodizer Rock only offers four scales, the diminished and augmented being quite uncommon in rock music. Other scales such as the harmonic minor scale or the melodic minor scale could be added to allow for more choices in Melodizer Rock. Many scales were actually implemented with Melodizer 2.0 and could easily be integrated in Melodizer Rock.

Further more, Melodizer Rock merged the notion of chord key and quality, with the key and mode that form a scale. The melodic line should actually propose the key and modes while the accompaniment should propose chord key and qualities. But the a link between the two should be made, as most of the accompaniment is often set in the **I** chord corresponding to the scale.

Other Constraints for Melodizer 2.0

When implementing Melodizer Rock, some constraints from the previous work done in Melodizer 2.0 [3] had to be set aside. Many of those constraints could actually be reintegrated and would allow the composer to have more control on the melodic line. Some example of those constraints are:

- **Minimum and maximum notes:** limiting the number of pushed notes throughout a phrase or a block of the song might allow for longer notes or rests that will allow the listener to relax between parts of the song.

- **Rhythm repetition:** in the song *Every Breath You Take*, some measures repeat themselves on a single phrase of the song. It might be interesting to allow the composer to ask for a rhythm to be repeated throughout a block or phrase.

- **Note repetition:** in rock songs, a note is often repeated. Very often this note is the tonic of the song's scale. Therefore, constraining the number of times it is repeated throughout a part of the song might lead to more recognisable melodies.

7.1.4 Improve the Musical Accompaniment

The accompaniment proposed by Melodizer Rock is quite simple, and does not give that much control to the composer. It can be improved in several ways.

More Chord Qualities

Melodizer Rock currently proposes only four types of chords, as described in section 2.1.4. But many variations of those chords exists, and existed in Melodizer 2.0:

- **Seventh chords:** they consist of a the classic triads with an added note which is a seventh above the root. For a major chord, it is thus eleven semitones above the root, and ten for a minor chord.
- **Ninth chords:** similarly as the previous chords, they add a ninth note to the initial triad. This corresponds to a second after the next octave, thus 14 semitones above the root for a major chord, and 13 for a minor chord.
- **Inverted chords:** they are triads of chords where the root note is transposed of an octave and thus end up higher than the two other notes.

Many variations of the classic chords used in Melodizer exist and would be interesting to add, as they are common in rock music.

Chord per Measure

Currently, Melodizer Rock lets the composer choose the accompaniment's chord for a complete block. Thus the corresponding chord will often span two measure, even if a different octave is played at each time. An interesting variation that could be added would be to allow a change of chord per measure in a same block.

Non-simultaneous Notes of the Chord

The current offered accompaniments only allow for triads to be played simultaneously. However, a variation could be to play the root note from the start, then play the other notes of the triads, along with their octaves, in a certain rhythm. This would allow for more a varied accompaniment. Other plays on the note of a chord, such as arpeggios, might be interesting to explore.

7.2 Explore Other Musical Genres

The constraints and structure concepts used throughout Melodizer Rock could be easily adapted to other music genres. The following list suggests some non-exhaustive genre examples that could be explored:

- **Ragtime:** a musical genre that originated from African-American communities, close in genre to a march and using poly-rhythm.[13]
- **Jazz:** a musical genre rooted in Ragtime that is characterised by some particular notes, chords and movement of the melody it uses.[14]
- **Alternative Rock:** founded on the rock genre explored by Melodizer Rock, it is focused on the use of guitars, their chords and riffs.[15]

- **Heavy Metal:** another genre based on the rock music explored in this thesis, that is characterised by the distorted sound of guitars, the guitar solos and its loudness.[16]
- **Country:** a genre that originated from the American working class. It is recognisable by its dance tunes of simple form, its harmonies and the used instruments.[17]
- **Reggae:** a music genre coming from Jamaica, recognisable by the counterpoint between its bass and drums downbeat, as well as the offbeat rhythm sections.[18]

7.3 GiL Overhead

GiL has many limitations and problems which could be solved by finding another way to run Gecode code directly in Common Lisp. A few examples of these limitations are listed below:

- GiL's performance is significantly worse than Gecode's standalone performance, due to the way it is built
- GiL is built on a specific version of Gecode, which might become obsolete, or have changes in method signatures.
- Each Gecode function must have its interface implemented manually in GiL, which is very inconvenient as not all Gecode functions are present within GiL.
- Readability of GiL code might not be as good as Gecode (`C++`) code.

Chapter 8

Conclusion

Melodizer Rock is a tool whose goal is to provide rock music scores meant to inspire the composer. This objective encompass many things, from allowing the user to interact with the solver, to actually computing solutions. Melodizer Rock's mission can be split into two halves: the development of an intuitive user interface and the process of constructing the corresponding problem.

8.1 An Interactive Interface

The visible part of Melodizer Rock is quite obviously its interface. It was built with an intended user in mind, the composer. Therefore, this interface had to be extremely straight forward and not require any technical knowledge much beyond the basic use of a computer.

First of all, allowing to see the structure of the song in an editor rather than by connecting blocks to one another was an important task. Melodizer Rock allows a composer to build their own structure by clicking a few buttons, and shows each change in a hierarchical representation.

Then, Melodizer Rock had to allow the composer to give specifications to the music, depending on the location in the song. With this objective in mind, the previously built structure had to be shown, and be editable. To this end, different objects were created, one for each part of the $AABA$ and *srdc* structure analysed by Drew Nobile [4]. Then, an interface for each of those phrases was developed to allow the modification of a specific part of the song, by going down into the hierarchy with a few button clicks.

Finally, an improvement made by Melodizer Rock over the previous works is the merging of the object representing the song, with the object representing the search. The *Rock* editor now proposes the necessary tools to launch the search and obtain the next solution by pressing a few buttons rather than connecting blocks in a patch.

8.2 A Specific CSP for Rock Music

Building a representation of the structure is important, but using it to develop a rock specific problem is even more essential. What makes a song belong to the rock genre will be the links between the different levels of the hierarchy, both vertical and horizontal ones. Those links could be expressed mathematically, and thus as constraints for a Constraint Satisfaction Problem.

The first step was to constrain the song in its entirety. Different variables for the melodic line and the accompaniment were created, with their own type-specific constraints. Then constraints are posted by going down in the hierarchy of the structure, linking each block to the others. The final step posts the specifications given by the composer through the different interfaces, on the smallest division of the song, that is in s, r, d, and c blocks.

This development lead to a whole new CSP, inspired from the one proposed in Melodizer 2.0 [3], but for songs with a hierarchical structure. It also added to the previous works by combining multiple voices, a monophonic one for the melody, and a polyphonic one for the accompaniment. Furthermore, it focused on creating songs that are singable, whether that be by constraining the intervals, or the pitches themselves.

8.3 An Impressive Tool for Composing

All this development is interesting, but what makes it important? This new tool is a great basis to build CSPs for any music genre with a hierarchical structure. It could easily be adapted to Jazz, Ragtime, Metal ... and is therefore a necessary step towards a larger tool for the creation of music scores.

But one could still wonder what its benefit is when compared to the use of generative Artificial Intelligence models such as *ChatGPT-4*. Both tools are important and impressive, but have their difference and specific uses. The main difference is in the obtained results, while generative models will generate answers to prompts based on the data they're trained on, Melodizer Rock will find a solution from scratch. As a result, generative models might produce songs which are more enjoyable to the listener, but they will sound similar to existing songs. On the other hand, a song produced by Melodizer Rock might sound less harmonious, but won't sound as similar to existing songs and will give the composer more control and creativity. Melodizer Rock can inspire composers through these original solutions in ways that generative models can't.

It is quite clear that Melodizer Rock is far from having reached its full potential. Many improvements can be added to give even more control to the composer. Improving the base melodic line, and allowing more variations to the accompaniment, are great first leads towards improving Melodizer Rock. One could also specify the CSP towards a more rock sounding problem by adding constraints on the blocks of the structure.

Bibliography

[1] B. Lapière, "Computer-aided musical composition," https://www.info.ucl.ac.be/~pvr/LAPIERE_2020.pdf, M.S. thesis, Université Catholique de Louvain, 2020.

[2] D. Sprockeels, "Melodizer : A constraint programming tool for computer-aided musical composition," https://www.info.ucl.ac.be/~pvr/SPROCKEELS_68641400_2022.pdf, M.S. thesis, Université Catholique de Louvain, 2021-2022.

[3] C. Chardon, A. Diels, and F. Gobbi, "Melodizer 2.0 : A constraint programming tool for computer-aided musical composition," https://www.info.ucl.ac.be/~pvr/Chardon_55411600_Diels_22601600_Gobbi_12201500.pdf, M.S. thesis, Université Catholique de Louvain, 2021-2022.

[4] D. F. Nobile, "A structural approach to the analysis of rock music," https://academicworks.cuny.edu/cgi/viewcontent.cgi?article=1082&context=gc_etds, Ph.D. dissertation, The City University of New York, 2014.

[5] M. Pilhofer and H. Day, *Music Theory for dummies*. John Wiley & Sons, Inc., 2019.

[6] R. Gauldin, "Harmonic practice in tonal music," in W.W. Norton & Company, Inc., 2004, pp. 3–66.

[7] R. Gauldin, "Harmonic practice in tonal music," in W.W. Norton & Company, Inc., 2004, pp. 94–115.

[8] R. Gauldin, "Harmonic practice in tonal music," in W.W. Norton & Company, Inc., 2004, pp. 126–143.

[9] K. Apt, *Principles of Constraint Programming*. Cambridge University Press, 2003, https://books.google.be/books?id=HEta789MatkC&printsec=frontcover&hl=fr#v=onepage&q&f=false.

[10] C. Schulte, G. Tack, and M. Z. Lagerkvist, *Modeling and Programming with Gecode*. 2019, https://www.gecode.org/doc-latest/MPG.pdf.

[11] *OpenMusic Documentation*, visited on 25-05-2021, Ircam - Centre Pompidou.

[12] G. M. Sumner, *Every breath you take*, produced by The Police, 1983.

[13] Wikipedia contributors, *Ragtime — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Ragtime&oldid=1156437541, [Online; accessed 2-June-2023], 2023.

[14] Wikipedia contributors, *Jazz — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Jazz&oldid=1145468925, [Online; accessed 2-June-2023], 2023.

[15] Wikipedia contributors, *Alternative rock — Wikipedia, the free encyclopedia*, https://en.wikipedia.org/w/index.php?title=Alternative_rock&oldid=1155616759, [Online; accessed 2-June-2023], 2023.

[16] Wikipedia contributors, *Heavy metal music — Wikipedia, the free encyclopedia*, [Online; accessed 2-June-2023], 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Heavy_metal_music&oldid=1157601281.

[17] Wikipedia contributors, *Country music — Wikipedia, the free encyclopedia*, [Online; accessed 2-June-2023], 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Country_music&oldid=1157495684.

[18] Wikipedia contributors, *Reggae — Wikipedia, the free encyclopedia*, [Online; accessed 2-June-2023], 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Reggae&oldid=1157601424.

Appendix A

Installation and Setup

This appendix gives instructions on how to install Melodizer Rock. As a disclaimer, Melodizer Rock can not be installed on Windows systems. GiL will not function properly as the Lisp version used by OpenMusic is a 32bit version and the Windows version of Gecode is 64bit.

A.1 Download and Installation

Melodizer Rock is dependant on the following tools, which have to be downloaded and installed according to their respective instructions:

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

As a reminder to the user, if any problem occurs during the installation, please refer to these tools' installation instructions and `README`s as they contain all the necessary troubleshooting information.

Once these tools have been downloaded and installed properly, the following *GitHub* repositories have to be cloned:

- GiL: `https://github.com/sprockeelsd/GiL`
- Melodizer Rock: `https://github.com/felixlepeltier/Melodizer-Rock`

After which, GiL's branch has to be switched to `melodizer-rock-bab`, as this is the version needed to use Melodizer Rock as intended.

A.2 Setup

Melodizer Rock and GiL are both libraries which are used inside OpenMusic. Therefore, they must be imported within this software. The following steps explain how to load these libraries in OpenMusic:

1. Launch OpenMusic
2. Enter an existing workspace, or create a new workspace
3. In the taskbar, click on "Windows" and then "Library", or simply press `Shift+Ctrl+P`
4. In the taskbar, click on "File" and then "Add Remote User Library"
5. Navigate to both GiL and Melodizer Rock's folders and add them

Both GiL and Melodizer Rock are now loaded into OpenMusic.

If you wish to load these libraries by default into OpenMusic, to avoid this tedious library loading process each time you launch OpenMusic, then follow these steps:

1. Launch OpenMusic
2. Enter an existing workspace, or create a new workspace
3. In the taskbar, click on "OM 7.1" and then "Preferences", or simply press `Ctrl+,`
4. In the pop-up, click on the "Libraries" tab
5. Click on the folder icon
6. Navigate to both GiL and Melodizer Rock's folders and add them
7. Click on "Apply"
8. Check the boxes next to both GiL and Melodizer in the "Auto Load" box

Appendix B

Tutorial for Melodizer Rock

Below is a basic step-by-step tutorial aiming to give explanations on how to go from an empty OpenMusic workspace to composing with Melodizer Rock, from which the user will have more than enough knowledge to reproduce the examples in chapter 6.

1. Launch OpenMusic
2. Enter an existing workspace, or create a new workspace
3. Create a new patch by right clicking on your workspace, and then on "New Patch", or just press `Ctrl+1`
4. Double click this patch
5. Then in the taskbar click "Classes" then "Libraries > Melodizer > ALL > ROCK" and click your patch interface to add the *Rock* object

Melodizer Rock is now ready to be used if your patch looks like Figure B.1. By double clicking on this *Rock* object and interacting with it, you can start creating music scores.

Now to create a basic example with Melodizer Rock, follow these steps

1. Double click the *Rock* object
2. Click the "Add A" button
3. Set "Number of bars" to 4 via the drop-down menu
4. Set "Min note length" to 4 via the drop-down menu, and then check the check-box on its left
5. Set "Minimum pitch" to slightly below half
6. Set "Maximum pitch" to slightly above half

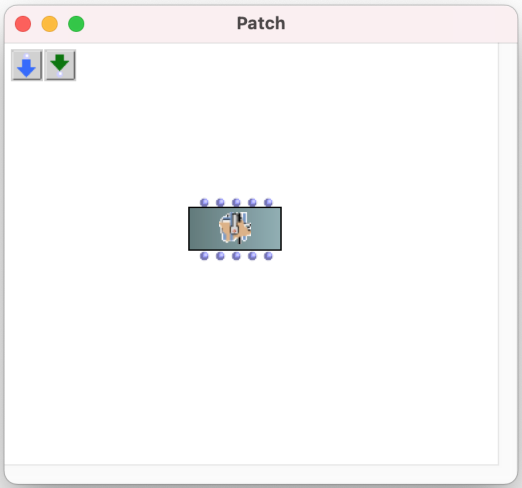


Figure B.1: OpenMusic patch with a *Rock* object instance

7. Click Start

8. Click Next

The *Rock* object interface should look like Figure B.2, and a "current solution" window should pop-up like Figure B.3. Now let's start using more of Melodizer Rock's capabilities, and create an example that is based on a source melody. To do so, follow these steps:

1. Go to the patch's interface

2. Then in the taskbar click "Classes" then "Score > VOICE" and click your patch interface to add the *Voice* object

3. Double click the *Voice* object

4. You can now modify this *Voice* object so that it contains your input melody, which can be done by using the commands explained in the taskbar's "Help > Editor Command Keys..." menu (or just press `Shift+Ctrl+H`)

5. Close this *Voice* object interface, and press `b` to block it if it is not already marked with a cross

6. Connect the *Voice* object to the *Rock* object, by linking the first output of this *Voice* object to the third input of the *Rock* object, the patch should look like Figure B.4

7. Click once on the *Rock* object and press `V` on your keyboard, this will run the *Rock* object and will process the input voice object

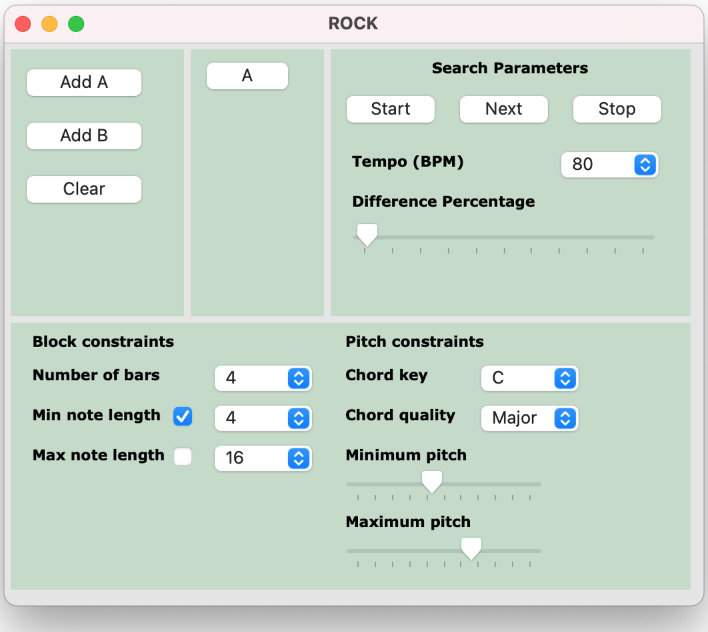


Figure B.2: *Rock* object interface, with one *A* block

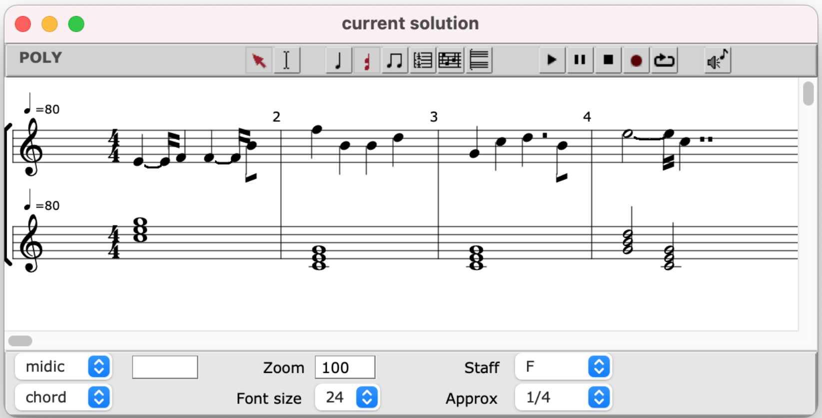


Figure B.3: Example solution that is obtained with one *A* block

8. *Rock* is now ready to create an example using this source melody, you can follow the same steps as done previously and will obtain a solution using your source melody and a single *A* block

If you wish to use a source melody for *B*, then the input in the *Rock* object that will be able to process it is the fourth one from the left.

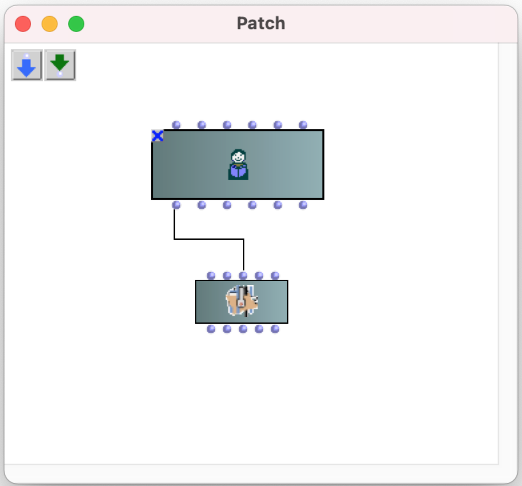


Figure B.4: Patch with a *Voice* object given as input to *Rock*'s first *A* block

Appendix C

Constraints

This chapter will recall the different constraints used through Melodizer Rock and give their implementation in `C++` using Gecode.

C.1 General Constraints

This section comport the constraints explained in section 4.3.

C.1.1 Accompaniment General Constraints

This section refers to the constraints explained in 4.3.1

Link push pull and playing

1. 1 $playing[i] = playing[i-1] - pull[i] + push[i]$
2. 2 $pull[i] \subseteq playing[i-1]$
3. 3 $push[i] \cap (playing[i-1] - pull[i]) = \emptyset$

```
for(int i = 1; i < playing.size(); i++){
    SetVar temp(*this, 0, max_pitch, 0, max_simultaneous_notes);
    rel(*this, playing[i-1], SOT_SUB, pull[i], temp);
    //Constraint 1
    rel(*this, temp, SOT_UNION, push[i], playing[i]);
    //Constraint 2
    rel(*this, pull[i], SRT_SUB, playing[i-1]);
    //Constraint 3
    rel(*this, playing[i-1], SOT_MINUS, pull[i], SRT_DISJ, push[i]);
}
```

And the constraints for the first index of the arrays:

1. 1 $pull[0] = \emptyset$
2. 2 $push[0] = playing[0]$

```
// Constraint 1
dom(*this, pull[0], SRT_EQ, IntSet::empty);
// Constraint 2
rel(*this, push[0], SRT_EQ, playing[0]);
```

Simultaneous Notes

For all $i \in [0, ..., k-1]$ where k is the size of the array, `min-sim` and `max-sim` being respectively the minimum and maximum number of notes that can play simultaneously:

$$min - sim \leq |playing[i]| \leq max - sim$$

```
for(int i = 0; i < k; i++){
    cardinality(*this, playing[k], min_sim, max_sim);
}
```

Chord Key and Quality

This is the implementation of the constraint of 1. For a `playing` array of size k, $\forall i \in [0, ..., k-1]$:

$$playing[i] \in octave(chord, quality)$$

```
for(int i = 0; i < k; i++){
    // Octave is the list of octaves of the chord
    BoolVarArray bool_array(*this, octaves.size(), 0, 1);
    for(int j = 0; j < octaves.size(); j++){
    // triad is the set of three notes corresponding to the chord and
    ↪ quality
        Reify r(bool_array[j], RM_IMP);
        rel(*this, playing[i], SRT_EQ, octaves[i], r);
    }
    rel(*this, BOT_XOR, bool_array); // One of the triads must be played
}
```

Minimum Note Length

This is the implementation of the first constraint of 2. For arrays `push` and `pull` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \nsubseteq \; pull[i+j] \; \forall j \in \{1, ..., min\_length - 1\}$$

```
for(int i = 0; i < k; i++){
    for(int j = 0; j < min_note_length && i+j < k; j++){
        rel(*this, pull[i+j], SRT_DISJ, push[i]);
    }
}
```

Maximum Note Length

This is the implementation of the second constraint of 2. For `max-note-length`, the equation is for arrays `push` and `pull` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \in \bigcup_{j \in \{1,...,max\_length-1\}} pull[i+j]$$

```
for(int i = 0; i < k; i++){
    SetVarArray l_pull(*this, max_length, 0, 127, 0, 127);
    SetVar l_pull_union(*this, 0, 127, 0, 127);
    //union of all pulled notes during max_length

    for(int k = 0; k < max_length; k++){
        rel(*this, l_pull[k], SRT_EQ, pull[i+k+1]);
    }
    rel(*this, SOT_UNION, l_pull, l_pull_union);
    // push[i] included in l-pull-union
    rel(*this, push[i], SRT_SUB, l_pull_union);
}
```

Maximum and Minimum Pitch

This is the implementation of the constraint 3. For a `push` array of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \subseteq \{min\_pitch, ..., max\_pitch\}$$

```
for(int i = 0; i < k; i++){
    dom(*this, push[i], SRT_SUB, min_pitch, max_pitch);
}
```

C.1.2 Melody General Constraints

This section develops the implementation of the constraints explained in 4.3.2

Link push pull and playing

1. 1 $playing[i] = playing[i-1] \mid\mid playing[i] = push[i]$
2. 2 $push[i] = playing[i] \mid\mid push[i] = -1$
3. 3 $pull[i] = playing[i-1] \mid\mid pull[i] = -1$
4. 4 $push[i] \neq -1 \Rightarrow pull[i] = playing[i-1]$
5. 5 $playing[i] = -1 \Rightarrow push[i] = -1 \; \&\& \; pull[i] = playing[i-1]$
6. 6 $playing[i] = playing[i-1] \Leftrightarrow push[i] = pull[i]$

```
for(int i = 1; i < push.size(); i++){
    BoolVar playing_i_playing_i_one = expr(*this, playing[i] == playing[i-1]);
    BoolVar push_i_playing_i = expr(*this, push[i] == playing[i]);
    // Constraint 1
    rel(*this, playing_i_playing_i_one, BOT_OR, push_i_playing_i, 1);

    BoolVar push_i_one = expr(*this, push[i] == -1);
    // Constraint 2
    rel(*this, push_i_playing_i, BOT_OR, push_i_one, 1);

    BoolVar pull_i_playing_i_one = expr(*this, pull[i] == playing[i-1]);
    BoolVar pull_i_one = expr(*this, pull[i] == -1);
    // Constraint 3
    rel(*this, pull_i_playing_i_one, BOT_OR, pull_i_one, 1);

    BoolVar push_i_nq_one = expr(*this, push[i] != -1);
    // Constraint 4
    rel(*this, push_i_nq_one, BOT_IMP, pull_i_playing_i_one, 1);

    BoolVar playing_i_one = expr(*this, playing[i] = -1);
    // Constraint 5
    rel(*this, playing_i_one, BOT_IMP, push_i_one, 1);
    rel(*this, playing_i_one, BOT_IMP, pull_i_playing_i_one, 1);

    BoolVar push_i_pull_i = expr(*this, push[i] == pull[i]);
    // Constraint 6
```

```
    rel(*this, playing_i_playing_i_one, BOT_IMP, push_i_pull_i, 1);
    rel(*this, push_i_pull_i, BOT_IMP, playing_i_playing_i_one, 1);
}
```

And the two constraints for the first index of the arrays are:

1. 1 $pull[0] = -1$

2. 2 $push[i] = playing[i]$

```
// Constraint 1
rel(*this, pull[0], IRT_EQ, -1);
// Constraint 2
rel(*this, push[0], IRT_EQ, playing[0]);
```

Chord Key and Quality

This section develops the implementation for the constraint of 4.3.2. For a `playing` array of size k, $\forall i \in [0, ..., k-1]$:

$$playing[i] \in scaleset(chord, quality) \mid\mid playing[i] = -1$$

```
for(int i = 0; i < k; i++){
    int * chordset = scaleset(chord, quality);
    BoolVarArray boolArray(*this, chordset.size()+1, 0, 1);

    for(int j = 0; j < chordset.size(); j++){
        BoolVar isNote = expr(*this, playing[i] == chordset[j]);
        rel(*this, boolArray[i], IRT_EQ, isNote);
    }

    BoolVar isMinusOne = expr(*this, playing[i] == -1);
    rel(*this, boolArray[chordset.size()], IRT_EQ, isMinusOne);
    // The note is one of the note of chordset or is equal to -1
    rel(*this, BOT_OR, boolArray, 1);
}
```

Minimum Note Length

This section refers to the first constraint of 4.3.2. For arrays `push` and `pull` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \neq -1 \Rightarrow pull[i+j] = -1 \ \forall j \in \{1, ..., min\_length - 1\}$$

$\forall i \in [1, ..., k-1]$:

$$playing[i-1] \neq -1 \ \&\& \ playing[i] = -1 \Rightarrow playing[i+j] = -1$$

```
for(int j = 0; j < k; j++){
    for(int n = 1; n < min_length; n++){
        // If a note is pushed, can't be pulled before min_length
        BoolVar pushed = expr(*this, push[j] != -1);
        BoolVar pulled = expr(*this, pull[j+n] == -1);
        rel(*this, pushed, BOT_IMP, pulled, 1);

        //If no note is playing, no note can play before min_length
        if(j > 0){
            BoolVar playing_j = expr(*this, playing[j] == -1);
            BoolVar playing_j_1 = expr(*this, playing[j-1] != -1);
            BoolVar playing_j_n = expr(*this, playing[j+n] == -1);
            BoolVar rest(*this, 0, 1);

            rel(*this, playing_j, BOT_AND, playing_j_1, rest);
            rel(*this, rest, BOT_IMP, playing_j_n, 1);
        }else{
            BoolVar playing_j = expr(*this, playing[j] == -1);
            BoolVar playing_j_n = expr(*this, playing[j+n] == -1);
            rel(*this, playing_j, BOT_IMP, playing_j_n, 1);
        }
    }
}
```

Maximum Note Length

This section refers to the second constraint of 4.3.2. For arrays `push` and `pull` of size k, $\forall i \in [0, ..., k-1]$:

$$push[i] \neq -1 \Rightarrow push[i] \in \bigcup_{j \in \{1, ..., max\_length - 1\}} pull[i+j]$$

```
for(int j = 0; j < push.size() - max_length; j++){
    IntVar count(*this, 0, max_length);
    IntVarArray int_array(*this, max_length, 0, max_length);

    for(int k = 0; k < max_length; k++){
        int_array[k] = expr(*this, push[j] - pull[j+k+1]);
    }
    //The pushed note must have appeared at least once
    count(*this, int_array, 0, IRT_EQ, count);
    rel(*this, count, IRT_GQ, 1);
}
```

Maximum and Minimum Pitch

This section shows the implementation of the constraint 4.3.2. For an array `push` of size k, $\forall i \in [0, ..., k-1]$, this is written as:

$$push[i] \subseteq (\{min\_pitch, ..., max\_pitch\} \cup \{-1\})$$

```
for(int j = 0; j < k; j++){
    BoolVar bool_one = expr(*this, push[j] == -1);
    BoolVar bool_min = expr(*this, push[j] >= min_pitch);
    BoolVar bool_max = expr(*this, push[j] <= max_mitch);
    BoolVar temp(*this, 0, 1);

    // Either the note is between the bounds, or it is equal to -1
    rel(*this, bool_min, BOT_AND, bool_max, temp);
    rel(*this, temp, BOT_OR, bool_one, 1);
}
```

Intervals

This section implements the last constraint of section 4.3.2. For an array `playing` of size k, $\forall i \in [1, ..., k-1]$, one can write:

$$|playing[i] - playing[i-1]| \leq 7 \text{ if } playing[i] \neq -1$$

```
for(int i = 1; i < k; i++){
    BoolVar playing_i = expr(*this, playing[i] == -1);
    BoolVar playing_i_one = expr(*this, playing[i-1] == -1);
```

```

    IntVar interval = expr(*this, playing[i] - playing[i-1]);
    IntVar interval_abs(*this, 0, 127);
    abs(*this, interval, interval_abs);

    BoolVar interval_max = expr(*this, interval_abs <= max_interval);
    BoolVar temp(*this, 0, 1);

    //Either one of the note is a rest, or the interval is respected
    rel(*this, playing_i, BOT_OR, playing_i_one, temp);
    rel(*this, temp, BOT_OR, interval_max, 1);
}
```

C.2 Block Specific Constraints

This section refers to the constraints explained in section 4.4.

C.2.1 Melody Source Constraints

This section describes the implementation of the constraints described in the first part of section 4.4.2. Let the source melody be represented by $\{push, pull, playing\}_{source}$ arrays of i elements, and s by $push, pull, playing$ arrays of j elements. The constraints can then be written $\forall k \in [0, \min(i, j) - 1]$ as:

$$push[k] = push_{source}[k]$$
$$pull[k] = pull_{source}[k]$$
$$playing[k] = playing_{source}[k]$$

```
for(int j = 0; j < i; j++){
    rel(*this, push[i], IRT_EQ, push-source[i]);
    rel(*this, playing[i], IRT_EQ, playing-source[i]);
}
for(int j = 0; j < i - 1; j++){
    rel(*this, pull[i], IRT_EQ, pull-source[i]);
}
```

```
rel(*this, push-acc[0], IRT_EQ, notes-to-play[0]);
rel(*this, push-acc[push-acc.size()/2], IRT_EQ, notes-to-play[1]);
```

C.2.2 Similarity Constraint Between IntVarArrays

This section describes the constraint explained in 4.4.1 for similarity between arrays. Given two arrays x and y with respectively i and j elements, their resemblance (in percent) sim is computed as such:

$$k = \min(i, j)$$

$$sim = |\, \{x[l] : x[l] = y[l] \mid l \in [0, k-1]\} \,|/k$$

Given $minsim$ the minimal similarity in percent, the resemblance is computed as:

$$count = |\, \{x[l] : x[l] = y[l] \mid l \in [0, k-1]\} \,|$$

$$count >= \lceil minsim * k \rceil$$

```
IntVar count(*this, 0, k);
IntVarArray int_array(*this, k, -127, 127);

for(int i = 0; i < k; i++){
    int_array[i] = expr(*this, x[i] - y[i]);
}
// The number of similar note must be greater or equal
// to the minsim*k
count(*this, int_array, 0, IRT_EQ, count);
rel(*this, count, IRT_GQ, ceil(minsim*k));
```

C.2.3 Transposition of an IntVarArray

Two types of transpositions were explained in section 4.4. The first one defines the transposition from one scale to another. Given the same x and i as before, $index_{scale}(chord, quality, note)$ is the index of a note on the scale defined by chord and quality. Then $chord_x$ and $quality_x$ are the chord and quality in which the melody of x is set. Finally t is the transposed melody with same length as x, and $chord_t$ and $quality_t$ define the scale to transpose to, it can be written $\forall j \in [0, ..., i]$:

$$index_{scale}(chord_x, quality_x, x[j]) = index_{scale}(chord_t, quality_t, t[j])$$

In Gecode, given `scaleset(chord,quality)` a function providing the array of notes of the scale in order, it is implemented as:

```
int notes[] = scaleset(chord_x, quality_x);
int new_notes[] = scaleset(chord_t, quality_y);

```

```
4  IntVarArray t(*this, i, -1, 127);
5
6  for(int j = 0; j < i; j++){
7      BoolVarArray bool_array(*this, notes.size(), 0, 1);
8
9      for(int k = 0; k < min(notes.size(), new_notes.size()) k++){
10         BoolVar x_n = expr(*this, x[j] == notes[k]);
11         BoolVar t_n = expr(*this, n[j] == new_notes[k]);
12         rel(*this, x_n, BOT_IMP, t_n, 1);
13     }
14 }
```

The second implementation is equivalent with just the line 2 replaced by:

```
new_notes[notes.size()];
for(int n = 0; n < notes.size(); n++){
    new_notes[n] = notes[n] + s;
}
```

C.2.4 c-specific Constraints

This section describes the last part of the constraints explained in 4.4. The cadence is defined by a succession of chord degrees *succession* (array of two distances from the root note, in semitones), $push_{acc}$ is the accompaniment's `push` array of i elements, and *chords* is an array of two elements. Each of these elements is a set of notes representing a chord to be played . Note that i is a multiple of 16, therefore $push_{acc}$ always has an even number of elements.

$$push_{acc}[0] = chords[0]$$
$$push_{acc}[i/2] = chords[1]$$

```
rel(*this, push_acc[0], SRT_EQ, chords[0]);
rel(*this, push_acc[i/2], SRT_EQ, chords[1]);
```

`octaves(tonic)` is a function returning a list of notes corresponding to all the possible octaves of the tonic. The last index of `playing` is forced to belong to this list:

$$playing[i-1] \in octave(tonic)$$

```
dom(*this, playing[i-1], octaves(tonic))
```

Appendix D

Melodizer Rock Code

This section shows the code that was explained and not shown in the main part of the thesis. It can be divided into four main parts:

1. The **package** definition of Melodizer Rock that allows to import it into Open Music
2. The definition of the **objects and interfaces** that compose Melodizer Rock's structure
3. The construction of the **CSP** specific to Melodizer Rock
4. The **utility** files that contain mostly useful functions used in the other three categories.

D.1 Package Setup

To be able to load the Melodizer Rock package into OpenMusic, two files are necessary:

- **Melodizer.lisp:** Contains the definitions of the files and objects to be loaded. It is located outside of a sources folder in which all the source code is located.
- **package.lisp:** defines the code as a package for Open Music. Located in the source folder.

D.1.1 Melodizer.lisp

```
(in-package :om)

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


(mapc 'compile&load (list
                     (make-pathname :directory (append (pathname-directory *load-pathname*)
                     ↪  (list "sources")) :name "package" :type "lisp")
```

```
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "melodizer-utils" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "melodizer-csp" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "melodizer-csts" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "block" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "rock-utils" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "rock" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "rock-AB" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "rock-srdc" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "rock-accompaniment" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "rock-csp" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "rock-csts" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "dummy-problem" :type "lisp")
                            (make-pathname :directory (pathname-directory *melodizer-sources-dir*)
                            ↪  :name "golomb-ruler" :type "lisp")
                            ))


;; remplir à la fin
(fill-library '(("ALL" nil (mldz::melodizer mldz::block mldz::search mldz::rock) nil)
                ("UTILS" Nil Nil (mldz::get-voice mldz::to-midicent) nil)
))

(print "Melodizer Loaded")
```

D.1.2 sources/package.lisp

```
(in-package :om)

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

```

```
(require-library "GIL")

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

D.2 Objects

Different objects and their interfaces, as explained in Chapter 4, were implemented into Melodizer Rock. They are all located in the sources folder. Their implementation respects Figure 2.10 where a greater block contains its sub-blocks.

D.2.1 sources/rock.lisp

This file contains the Rock object, describing the whole song.

```
(in-package :mldz)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          ROCK CLASS                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Define a rock object containing the constraints
;; and attributes necessary for the search
(om::defclass! rock ()
    (
      (block-list
          :accessor block-list :initarg :block-list :initform nil
          :documentation "Block list containing the global musical structure")
      (melody-source-A
          :accessor melody-source-A :initarg :melody-source-A :initform nil
          :documentation "Source melody for s of the first A block")
      (melody-source-B
          :accessor melody-source-B :initarg :melody-source-B :initform nil
          :documentation "Source melody for s of the first B block")
      (bar-length
          :accessor bar-length :initform 0 :type integer
           :documentation "Number of bars contained in the block")
      (nb-a
          :accessor nb-a  :initform 0 :type integer
          :documentation "number of block A in the structure")
```

```lisp
        (nb-b
            :accessor nb-b  :initform 0 :type integer
            :documentation "number of block B in the structure")
        (idx-first-a
            :accessor idx-first-a :initform 0 :type integer
            :documentation "index of the first block A in the structure")
        (idx-first-b
            :accessor idx-first-b  :initform 0 :type integer
            :documentation "index of the first block B in the structure")
        (min-note-length-flag
            :accessor min-note-length-flag :initform nil :type integer
            :documentation "Flag stating if the note-min-length constrain must be posted")
        (min-note-length
            :accessor min-note-length :initform 1 :type integer
            :documentation "Minimum note length value")
        (max-note-length-flag
            :accessor max-note-length-flag :initform nil :type integer
            :documentation "Flag stating if the note-max-length constrain must be posted")
        (max-note-length
            :accessor max-note-length :initform 16 :type integer
            :documentation "Maximum note length value")
        (chord-key
            :accessor chord-key :initform "C" :type string
            :documentation "Chord key to set the scale in")
        (chord-quality
            :accessor chord-quality :initform "Major" :type string
            :documentation "Quality to set the scale in")
        (min-pitch
            :accessor min-pitch :initform 1 :type integer
            :documentation "Minimum pitch value")
        (max-pitch
            :accessor max-pitch :initform 127 :type integer
            :documentation "Maximum pitch value")
        (solution
            :accessor solution :initarg :solution :initform nil
            :documentation "The current solution of the CSP in the form of a voice object.")
        (result :accessor result
            :result :initform (list)
            :documentation "A list holder to store the result of the call to the CSPs")
        (stop-search
            :accessor stop-search :stop-search :initform nil
            :documentation "booleanto tell if the user wishes to stop the search or not.")
        (input-rhythm
            :accessor input-rhythm :input-rhythm :initform (make-instance 'voice)
            :documentation "The rhythm of the melody or a melody in the form of a voice
            ↪  object. ")
```

```lisp
      (tempo
          :accessor tempo :initform 80 :type integer
          :documentation "The tempo (BPM) of the project")
      (branching
          :accessor branching :initform "Top down" :type string
          :documentation "The tempo (BPM) of the project")
      (percent-diff
          :accessor percent-diff :initform 1 :type integer
          :documentation "The minimum difference percentage between solutions")
    )
)


(defclass rock-editor (om::editorview) ())

(defmethod om::class-has-editor-p ((self rock)) t)
(defmethod om::get-editor-class ((self rock)) 'rock-editor)
(defmethod om::om-draw-contents ((view rock-editor))
  (let* ((object (om::object view)))
    (om::om-with-focused-view
      view
    )
  )
)

(defmethod initialize-instance ((self rock-editor) &rest args)
  ;;; do what needs to be done by default
  (call-next-method) ; start the search by default?
  (make-my-interface self)
)

(defmethod make-my-interface ((self rock-editor))

  ; create the main view of the object
  (make-main-view self)
  (let*
    (
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different regions of the tool ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      (rock-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 130 200)
        :position (om::om-make-point 5 5)
        :bg-color om::*azulito*)
      )
```

```lisp
      (constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 510 200)
        :position (om::om-make-point 5 210)
        :bg-color om::*azulito*)
      )
      (structure-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 100 200)
        :position (om::om-make-point 140 5)
        :bg-color om::*azulito*)
      )
      (search-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 270 200)
        :position (om::om-make-point 245 5)
        :bg-color om::*azulito*)
      )
    )

    (setf elements-rock-panel (make-rock-panel self rock-panel))
    (setf elements-constraints-panel (make-constraints-panel self constraints-panel))
    (setf elements-structure-panel (make-structure-panel self structure-panel))
    (setf elements-search-panel (make-rock-search-panel self search-panel))

    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      rock-panel
      constraints-panel
      structure-panel
      search-panel
    )
  )
  ; return the editor
  self
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                  INTERFACE CONSTRUCTION                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ;;;;;;;;;;;;;;;;;
    ;;; main view ;;;
    ;;;;;;;;;;;;;;;;;
```

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


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         ROCK PANEL                         ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defun make-rock-panel (editor rock-panel)
  (om::om-add-subviews
    rock-panel

    ;; Button to add a block A at the end of the current block-list
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 5 10) ; position (horizontal, vertical)
        (om::om-make-point 100 20) ; size (horizontal, vertical)
        "Add A"
        :di-action #'(lambda (b)
          (print "Added A to structure")
          ;;Create the block and set its values
          (let ((bar-length 0) (new-block (make-instance 'A :parent (om::object editor) (om::object editor))))
            (setf (block-position new-block) (length (block-list (om::object editor))))
            (setf (block-position-A new-block) (count-A-block-list (block-list (parent new-block))))
            (setf (block-list (om::object editor)) (append (block-list (om::object editor)) (list new-block)))
            (if (= (length (block-list (om::object editor))) 1)
              (setq bar-length 0)
              (setq bar-length (bar-length (first (block-list (om::object editor)))))
            )
            (if (= (nb-a (om::object editor)) 0)
              (setf (idx-first-a (om::object editor)) (block-position new-block))
            )
            (setf (nb-a (om::object editor)) (+ (nb-a (om::object editor)) 1))
            (setf (bar-length (om::object editor)) (+ bar-length (bar-length (om::object editor))))
            ;; Update the constraints values based on the Rock block
```

```
          (change-subblocks-values (om::object editor)
                                   :bar-length (bar-length (om::object editor))
                                   :chord-key (chord-key (om::object editor))
                                   :min-pitch (min-pitch (om::object editor))
                                   :max-pitch (max-pitch (om::object editor))
                                   :min-note-length-flag (min-note-length-flag (om::object editor))
                                   :min-note-length (min-note-length (om::object editor))
                                   :max-note-length-flag (max-note-length-flag (om::object editor))
                                   :max-note-length (max-note-length (om::object editor))
                                   :chord-quality (chord-quality (om::object editor))
          )
        )
        ;; (om::om-remove-subviews rock-panel)
        (make-my-interface editor)
      )
    )


    ;; Button to add a block B at the end of the current block-list
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 5 50) ; position (horizontal, vertical)
        (om::om-make-point 100 20) ; size (horizontal, vertical)
        "Add B"
        :di-action #'(lambda (b)
          (print "Added B to structure")
          ;;Create the block and set its values
          (let ((bar-length 0) (new-block (make-instance 'B :parent (om::object editor) (om::object editor))))
            (setf (block-position new-block) (length (block-list (om::object editor))))
            (setf (block-position-B new-block) (count-B-block-list (block-list (parent new-block))))
            (setf (block-list (om::object editor)) (append (block-list (om::object editor)) (list new-block)))
            (if (= (length (block-list (om::object editor))) 1)
              (setq bar-length 0)
              (setq bar-length (bar-length (first (block-list (om::object editor)))))
            )
            (if (= (nb-b (om::object editor)) 0)
              (setf (idx-first-b (om::object editor)) (block-position new-block))
            )
            (setf (nb-b (om::object editor)) (+ (nb-b (om::object editor)) 1))
```

```
            (setf (bar-length (om::object editor)) (+ bar-length (bar-length (om::object editor))))
            ;; Update the constraints values based on the Rock block
            (change-subblocks-values (om::object editor)
                                     :bar-length (bar-length (om::object editor))
                                     :chord-key (chord-key (om::object editor))
                                     :min-pitch (min-pitch (om::object editor))
                                     :max-pitch (max-pitch (om::object editor))
                                     :min-note-length-flag (min-note-length-flag (om::object editor))
                                     :min-note-length (min-note-length (om::object editor))
                                     :max-note-length-flag (max-note-length-flag (om::object editor))
                                     :max-note-length (max-note-length (om::object editor))
                                     :chord-quality (chord-quality (om::object editor))
            )
          )
          ;; (om::om-remove-subviews rock-panel)
          (make-my-interface editor)
        )
    )

    ;; Buton to erase every bit of the current structure
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 5 90) ; position (horizontal, vertical)
        (om::om-make-point 100 20) ; size (horizontal, vertical)
        "Clear"
        :di-action #'(lambda (b)
          (print "Cleared structure")
          (mp:process-run-function ; start a new thread for the execution of the next method
            "clear struct" ; name of the thread, not necessary but useful for debugging
            nil ; process initialization keywords, not needed here
            (lambda () ; function to call
              (setf (bar-length (om::object editor)) 0)
              (setf (block-list (om::object editor)) nil)
              (setf (nb-a (om::object editor)) 0)
              (setf (nb-b (om::object editor)) 0)
              (om::om-remove-subviews rock-panel)
              (make-my-interface editor)
            )
          )
        )
```

```
    )
  )
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      STRUCTURE PANEL                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-structure-panel (editor structure-panel)

  (let ((loop-index 0) (subview-list '()))
  ;; Loop on the block-list and create buttons for every block of the structure
  ;; that open the corresponding editor
  (loop for x in (block-list (om::object editor))
    do
      (if (typep x 'mldz::a)
        (setf subview-list (append subview-list (list (om::om-make-dialog-item
          'om::om-button
          (om::om-make-point 5 (+ 5 (* 30 loop-index))) ; position (horizontal, vertical)
          (om::om-make-point 75 20) ; size (horizontal, vertical)
          "A"
          :di-action #'(lambda (b)

            (print "Selected A")
            (mp:process-run-function ; start a new thread for the execution of the next method
              "next thread" ; name of the thread, not necessary but useful for debugging
              nil ; process initialization keywords, not needed here
              #'(lambda () ; function to call
                (om::openeditorframe ; open a window displaying the editor of the A block
                  (om::omNG-make-new-instance (nth (position b subview-list)
                  (block-list (om::object editor)))
                  (concatenate 'string "Window A" (write-to-string (position b subview-list))))
                )
              )
            )
          )
        ))))
      )

      (if (typep x 'mldz::b)
        (setf subview-list (append subview-list (list (om::om-make-dialog-item
```

```
          'om::om-button
          (om::om-make-point 5 (+ 5 (* 30 loop-index))) ; position (horizontal, vertical)
          (om::om-make-point 75 20) ; size (horizontal, vertical)
          "B"
          :di-action #'(lambda (b)
            (print "Selected B")
            (mp:process-run-function ; start a new thread for the execution of the next method
              "next thread" ; name of the thread, not necessary but useful for debugging
              nil ; process initialization keywords, not needed here
              #'(lambda () ; function to call
                (om::openeditorframe ; open a window displaying the editor of the B block
                  (om::omNG-make-new-instance (nth (position b subview-list)
                  (block-list (om::object editor)))
                  (concatenate 'string "Window B" (write-to-string (position b subview-list))))
                )
              )
            )
          )
        ))))
      )
      (setq loop-index (+ loop-index 1))
  )


  (if (not subview-list)
    (om::om-add-subviews
      structure-panel
    )
    (loop for x in subview-list
      do
        (om::om-add-subviews
          structure-panel
          x
        )
    )

  )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      CONSTRAINTS PANEL                     ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-constraints-panel (editor panel)
  (om::om-add-subviews
    panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 5)
      (om::om-make-point 120 20)
      "Block constraints"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 30)
      (om::om-make-point 100 20)
      "Number of bars"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 150 30)
      (om::om-make-point 80 20)
      "Bar length"
      :range (bar-length-range (om::object editor))
      :value (number-to-string (bar-length (om::object editor)))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (setf (bar-length (om::object editor)) (string-to-number check))
        (change-subblocks-values (om::object editor) :bar-length (bar-length (om::object
        ↪  editor)))
        (if (not (typep (om::object editor) 'mldz::rock))
          (progn
            (propagate-bar-length-srdc (om::object editor))
            (set-bar-length-up (om::object editor))
          )
        )
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 60)
      (om::om-make-point 100 20)
```

```
      "Min note length"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 120 60)
      (om::om-make-point 20 20)
      ""
      :checked-p (min-note-length-flag (om::object editor))
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (setf (min-note-length-flag (om::object editor)) 1)
                       (setf (min-note-length-flag (om::object editor)) nil)
                     )
                     (change-subblocks-values (om::object editor)
                                  :min-note-length-flag (min-note-length-flag (om::object
                                  ↪  editor))
                                  :min-note-length (min-note-length (om::object editor)))
      )
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 150 60)
      (om::om-make-point 80 20); size
      "Minimum note length"
      :range  (loop :for n :from 0 :upto 4 :collect (number-to-string (expt 2 n)))
      :value (number-to-string (min-note-length (om::object editor)))
      :di-action #'(lambda (m)
          (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
          (setf (min-note-length (om::object editor)) (string-to-number check))
          (change-subblocks-values (om::object editor)
                                  :min-note-length-flag (min-note-length-flag (om::object
                                  ↪  editor))
                                  :min-note-length (min-note-length (om::object editor)))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 90)
      (om::om-make-point 100 20)
      "Max note length"
      :font om::*om-default-font1b*
    )
```

```

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 120 90)
      (om::om-make-point 20 20)
      ""
      :checked-p (max-note-length-flag (om::object editor))
      :di-action #'(lambda (c)
                    (if (om::om-checked-p c)
                      (setf (max-note-length-flag (om::object editor)) 1)
                      (setf (max-note-length-flag (om::object editor)) nil)
                    )
                    (change-subblocks-values (om::object editor)
                                :max-note-length-flag (max-note-length-flag (om::object
                                ↪  editor))
                                :max-note-length (max-note-length (om::object editor)))
      )
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 150 90)
      (om::om-make-point 80 20); size
      "Maximum note length"
      :range  (loop :for n :from 0 :upto 4 :collect (number-to-string (expt 2 n)))
      :value (number-to-string (max-note-length (om::object editor)))
      :di-action #'(lambda (m)
          (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
          (setf (max-note-length (om::object editor)) (string-to-number check))
          (change-subblocks-values (om::object editor)
                                 :max-note-length-flag (max-note-length-flag (om::object
                                 ↪  editor))
                                 :max-note-length (max-note-length (om::object editor)))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 250 5)
      (om::om-make-point 200 20)
      "Pitch constraints"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-static-text
```

```
      (om::om-make-point 250 30)
      (om::om-make-point 100 20)
      "Chord key"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 350 30)
      (om::om-make-point 80 20)
      "Chord key"
      :range '("C" "C#" "D" "Eb" "E" "F" "F#" "G" "Ab" "A" "Bb" "B")
      :value (chord-key (om::object editor))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-key (om::object editor)) nil)
          (setf (chord-key (om::object editor)) check)
        )

        (change-subblocks-values (om::object editor) :chord-key check)
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 250 60)
      (om::om-make-point 100 20)
      "Chord quality"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 350 60)
      (om::om-make-point 80 20)
      "Chord quality"
      :value (chord-quality (om::object editor))
      :range '("Major" "Minor" "Augmented" "Diminished")
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-quality (om::object editor)) nil)
          (setf (chord-quality (om::object editor)) check))
        (change-subblocks-values (om::object editor) :chord-quality check)

```

```
      )
    )

     (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 250 90)
      (om::om-make-point 100 20)
      "Minimum pitch"
      :font om::*om-default-font1b*
    )


    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 250 110)
      (om::om-make-point 150 20)
      "Minimum pitch"
      :range '(1 127)
      :increment 1
      :value (min-pitch (om::object editor))
      :di-action #'(lambda (s)
        (setf (min-pitch (om::object editor)) (om::om-slider-value s))
        (change-subblocks-values (om::object editor)
                                 :min-pitch (min-pitch (om::object editor)))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 250 140)
      (om::om-make-point 100 20)
      "Maximum pitch"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 250 160)
      (om::om-make-point 150 20)
      "Maximum pitch"
      :range '(1 127)
      :increment 1
      :value (max-pitch (om::object editor))
      :di-action #'(lambda (s)
        (setf (max-pitch (om::object editor)) (om::om-slider-value s))
        (change-subblocks-values (om::object editor)
```

```
                                    :max-pitch (max-pitch (om::object editor)))
      )
    )
  )

)



;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                         SEARCH PANEL                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-rock-search-panel (editor search-panel)
  (om::om-add-subviews
    search-panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 75 5)
      (om::om-make-point 120 20)
      "Search Parameters"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 5 30) ; position (horizontal, vertical)
      (om::om-make-point 80 20) ; size (horizontal, vertical)
      "Start"
      :di-action #'(lambda (b)
          (setf (result (om::object editor))
                (rock-solver (om::object editor)
                             (percent-diff (om::object editor))
                             (branching (om::object editor))))
      )
    )

    (om::om-make-dialog-item
      'om::om-button
      (om::om-make-point 90 30) ; position
      (om::om-make-point 80 20) ; size
      "Next"
      :di-action #'(lambda (b)
        (if (typep (result (om::object editor)) 'null); if the problem is not initialized
```

```
          (error "The problem has not been initialized. Please set the input and press
          ↪  Start.")
        )
        (print "Searching for the next solution")
        ;reset the boolean because we want to continue the search
        (setf (stop-search (om::object editor)) nil)
        ;get the next solution
        (mp:process-run-function ; start a new thread for the execution of the next method
          "next thread" ; name of the thread, not necessary but useful for debugging
          nil ; process initialization keywords, not needed here
          (lambda () ; function to call
            (let ((res (new-rock-next (result (om::object editor)) (om::object editor))))
              (setf (solution (om::object editor)) (first res) (result (om::object editor))
              ↪  (cdr res))
              (om::openeditorframe ; open a voice window displaying the solution
                (om::omNG-make-new-instance (solution (om::object editor)) "current
                ↪  solution")
              )
            )
          )
        )
      )
    )

    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 175 30) ; position (horizontal, vertical)
        (om::om-make-point 80 20) ; size (horizontal, vertical)
        "Stop"
        :di-action #'(lambda (b)
          (setf (stop-search (om::object editor)) t)
        )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 75)
      (om::om-make-point 100 20)
      "Tempo (BPM)"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 75)
      (om::om-make-point 80 20)
```

```
      "Tempo"
      :range (loop :for n :from 30 :upto 200 :collect (number-to-string n))
      :value (number-to-string (tempo (om::object editor)))
      :di-action #'(lambda (m)
        (setf (tempo (om::object editor)) (string-to-number (nth
        ↪  (om::om-get-selected-item-index m) (om::om-get-item-list m))))
      )
    )


    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 105)
      (om::om-make-point 200 20)
      "Difference Percentage"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 15 130)
      (om::om-make-point 230 20)
      "Difference Percentage"
      :range '(0 100)
      :increment 1
      :value (percent-diff (om::object editor))
      :di-action #'(lambda (s)
        (setf (percent-diff (om::object editor)) (om::om-slider-value s))
      )
    )
  )

)
```

D.2.2 sources/rock-AB.lisp

This file contains the A and B objects. First by defining the objects and their attributes.

```
(in-package :mldz)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          A CLASS                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(om::defclass! A ()
    (
      (s-block
          :accessor s-block :initarg :s-block :initform (make-instance 's)
          :documentation "s sub-block, first few bars of the block")
      (r-block
          :accessor r-block :initarg :r-block :initform (make-instance 'r)
          :documentation "r sub-block, bars after s")
      (d-block
          :accessor d-block :initarg :d-block :initform (make-instance 'd)
          :documentation "d sub-blocks, bars after r")
      (c-block
          :accessor c-block :initarg :c-block :initform (make-instance 'c)
          :documentation "c sub-block, last few bars")
      (parent
          :accessor parent :initarg :parent :initform nil
          :documentation "parent block containing the instance of this block")
      (relative-to-parent
          :accessor relative-to-parent :initarg :relative-to-parent :initform 1 :type integer
          :documentation "Flag to now if the block attributes are reltive to its parent's")
      (relative-to-same
          :accessor relative-to-same :initarg :relative-to-same :initform nil :type integer
          :documentation "Flag to now if the block attributes are reltive to similar blocks")
      (bar-length
          :accessor bar-length :initform 0 :type integer
          :documentation "Number of bars of the block")
      (min-note-length-flag
          :accessor min-note-length-flag :initform nil :type integer
          :documentation "Flag stating if the note-min-length constrain must be posted")
      (min-note-length
          :accessor min-note-length :initform 1 :type integer
          :documentation "Minimum note length value")
      (diff-min-length
          :accessor diff-min-length :initform 0 :type integer
          :documentation "Difference for relative changes")
      (max-note-length-flag
          :accessor max-note-length-flag :initform nil :type integer
          :documentation "Flag stating if the note-max-length constrain must be posted")
      (max-note-length
```

```
            :accessor max-note-length :initform 16 :type integer
            :documentation "Maximum note length value")
        (diff-max-length
            :accessor diff-max-length :initform 0 :type integer
            :documentation "Difference for relative changes")
        (chord-key
            :accessor chord-key :initform "C" :type string
            :documentation "Chord key to set the scale in")
        (diff-chord-key
            :accessor diff-chord-key :initform 0 :type integer
            :documentation "Difference for relative changes")
        (chord-quality
            :accessor chord-quality :initform "Major" :type string
            :documentation "Quality to set the scale in")
        (diff-chord-quality
            :accessor diff-chord-quality :initform 0 :type integer
            :documentation "Difference for relative changes")
        (min-pitch
            :accessor min-pitch :initform 1 :type integer
            :documentation "Minimum pitch value")
        (diff-min-pitch
            :accessor diff-min-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (max-pitch
            :accessor max-pitch :initform 127 :type integer
            :documentation "Maximum pitch value")
        (diff-max-pitch
            :accessor diff-max-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (block-position
            :accessor block-position :initform -1 :type integer
            :documentation "Index of the A or B block within the global structure")
        (similarity-percent-A0
            :accessor similarity-percent-A0 :initform 50 :type integer
            :documentation "Percentage of resemblance with first A")
        (block-position-A
            :accessor block-position-A :initform -1 :type integer
            :documentation "Index of this block relative to other A blocks within the global
            ↪  structure")
        (block-position-B
            :accessor block-position-B :initform -1 :type integer
            :documentation  "Index of this block relative to other B blocks within the
            ↪  global structure")
        (semitones
            :accessor semitones :initform 0 :type integer
            :documentation "Semitones of transposition from key")
```

```lisp
    )
)

(defclass A-editor (om::editorview) ())

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

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

(defmethod initialize-instance ((self A-editor) &rest args)
  ;;; do what needs to be done by default
  (call-next-method) ; start the search by default?
  (make-my-interface self)
)



(defmethod make-my-interface ((self A-editor))

  ; create the main view of the object
  (make-main-view self)

  (let*
    (
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different regions of the tool ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      (A-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 50)
        :position (om::om-make-point 5 5)
        :bg-color om::*azulito*)
      )
      (changes-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 100)
        :position (om::om-make-point 5 60)
        :bg-color om::*azulito*)
      )
      (constraints-panel (om::om-make-view 'om::om-view
```

```lisp
        :size (om::om-make-point 500 300)
        :position (om::om-make-point 5 165)
        :bg-color om::*azulito*)
      )

    )

    (setf elements-A-panel (make-A-panel self A-panel))
    (if (= (block-position-A (om::object self)) (idx-first-a (parent (om::object self))))
      (setf elements-constraints-panel (make-constraints-AB-panel self constraints-panel))
      (setf elements-constraints-panel (make-constraints-not-first-panel self
      ↪  constraints-panel))
    )

    (setf elements-changes-panel (make-changes-panel self changes-panel))

    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      A-panel
      changes-panel
      constraints-panel
    )
  )
  ; return the editor
  self
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          B CLASS                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(om::defclass! B ()
    (
      (s-block
          :accessor s-block :initarg :s-block :initform (make-instance 's)
          :documentation "s sub-block, first few bars of the block")
      (r-block
          :accessor r-block :initarg :r-block :initform (make-instance 'r)
          :documentation "r sub-block, bars after s")
      (d-block
          :accessor d-block :initarg :d-block :initform (make-instance 'd)
```

```
        :documentation "d sub-blocks, bars after r")
    (c-block
        :accessor c-block :initarg :c-block :initform (make-instance 'c)
        :documentation "c sub-block, last few bars")
    (parent
        :accessor parent :initarg :parent :initform nil
        :documentation "parent block containing the instance of this block")
    (relative-to-parent
        :accessor relative-to-parent :initarg :relative-to-parent :initform 1 :type integer
        :documentation "Flag to now if the block attributes are reltive to its parent's")
    (relative-to-same
        :accessor relative-to-same :initarg :relative-to-same :initform nil :type integer
        :documentation "Flag to now if the block attributes are reltive to similar blocks")
    (bar-length
        :accessor bar-length :initform 0 :type integer
        :documentation "Number of bars of the block")
    (min-note-length-flag
        :accessor min-note-length-flag :initform nil :type integer
        :documentation "Flag stating if the note-min-length constrain must be posted")
    (min-note-length
        :accessor min-note-length :initform 1 :type integer
        :documentation "Minimum note length value")
    (diff-min-length
        :accessor diff-min-length :initform 0 :type integer
        :documentation "Difference for relative changes")
    (max-note-length-flag
        :accessor max-note-length-flag :initform nil :type integer
        :documentation "Flag stating if the note-max-length constrain must be posted")
    (max-note-length
        :accessor max-note-length :initform 16 :type integer
        :documentation "Maximum note length value")
    (diff-max-length
        :accessor diff-max-length :initform 0 :type integer
        :documentation "Difference for relative changes")
    (chord-key
        :accessor chord-key :initform "C" :type string
        :documentation "Chord key to set the scale in")
    (diff-chord-key
        :accessor diff-chord-key :initform 0 :type integer
        :documentation "Difference for relative changes")
    (chord-quality
        :accessor chord-quality :initform "Major" :type string
```

```
            :documentation "Quality to set the scale in")
        (diff-chord-quality
            :accessor diff-chord-quality :initform 0 :type integer
            :documentation "Difference for relative changes")
        (min-pitch
            :accessor min-pitch :initform 1 :type integer
            :documentation "Minimum pitch value")
        (diff-min-pitch
            :accessor diff-min-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (max-pitch
            :accessor max-pitch :initform 127 :type integer
            :documentation "Maximum pitch value")
        (diff-max-pitch
            :accessor diff-max-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (block-position
            :accessor block-position :initform -1 :type integer
            :documentation "Index of the A or B block within the global structure")
        (similarity-percent-A0
            :accessor similarity-percent-B0 :initform 50 :type integer
            :documentation "Percentage of resemblance with first A")
        (block-position-A
            :accessor block-position-A :initform -1 :type integer
            :documentation "Index of this block relative to other A blocks within the global structure")
        (block-position-B
            :accessor block-position-B :initform -1 :type integer
            :documentation  "Index of this block relative to other B blocks within the global structure")
        (semitones
            :accessor semitones :initform 0 :type integer
            :documentation "Semitones of transposition from key")
    )
)

(defclass B-editor (om::editorview) ())

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

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

```lisp
  )
)

(defmethod initialize-instance ((self B-editor) &rest args)
  ;;; do what needs to be done by default
  (call-next-method) ; start the search by default?
  (make-my-interface self)
)


(defmethod make-my-interface ((self B-editor))

  ; create the main view of the object
  (make-main-view self)

  (let*
    (
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different regions of the tool ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      (B-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 50)
        :position (om::om-make-point 5 5)
        :bg-color om::*azulito*)
      )
      (changes-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 100)
        :position (om::om-make-point 5 60)
        :bg-color om::*azulito*)
      )
      (constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 300)
        :position (om::om-make-point 5 170)
        :bg-color om::*azulito*)
      )

    )

    (setf elements-B-panel (make-B-panel self B-panel))
    (if (= (block-position (om::object self)) (idx-first-b (parent (om::object self))))
      (setf elements-constraints-panel (make-constraints-AB-panel self constraints-panel))
      (setf elements-constraints-panel (make-constraints-not-first-panel self
      ↪  constraints-panel))
    )
    (setf elements-changes-panel (make-changes-panel self changes-panel))
```

```

    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      B-panel
      changes-panel
      constraints-panel
    )
  )
  ; return the editor
  self
)

```

Then by defining the interfaces.

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          A PANEL                             ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



(defun make-A-panel (editor A-panel)

  (om::om-add-subviews
    A-panel
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 5 10) ; position (horizontal, vertical)
        (om::om-make-point 80 25) ; size (horizontal, vertical)
        "s"
        :di-action #'(lambda (b)
          (print "Selected s")
          (mp:process-run-function ; start a new thread for the execution of the next
          ↪  method
            "next thread" ; name of the thread, not necessary but useful for debugging
            nil ; process initialization keywords, not needed here
            (lambda () ; function to call
                (setf (parent (s-block (om::object editor))) (om::object editor))
              ;; (setf (s-block (om::object editor)) (make-instance 's :parent (om::object
              ↪  editor) (om::object editor)))
              (om::openeditorframe ; open a window displaying the editor of the first A
              ↪  block
```

```
                (om::omNG-make-new-instance (s-block (om::object editor)) "Window s")
              )
            )
          )
        )
    )
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 115 10) ; position (horizontal, vertical)
        (om::om-make-point 80 25) ; size (horizontal, vertical)
        "r"
        :di-action #'(lambda (b)
          (print "Selected r")
          (mp:process-run-function ; start a new thread for the execution of the next method
            "next thread" ; name of the thread, not necessary but useful for debugging
            nil ; process initialization keywords, not needed here
            (lambda () ; function to call
            (setf (parent (r-block (om::object editor))) (om::object editor))
              ;; (setf (r-block (om::object editor)) (make-instance 'r :parent (om::object editor) (om::object editor)))
              (om::openeditorframe ; open a window displaying the editor of the first A block
                (om::omNG-make-new-instance (r-block (om::object editor)) "Window r")
              )
            )
          )
        )
    )
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 225 10) ; position (horizontal, vertical)
        (om::om-make-point 80 25) ; size (horizontal, vertical)
        "d"
        :di-action #'(lambda (b)
          (print "Selected d")
          (mp:process-run-function ; start a new thread for the execution of the next method
            "next thread" ; name of the thread, not necessary but useful for debugging
            nil ; process initialization keywords, not needed here
            (lambda () ; function to call
                (setf (parent (d-block (om::object editor))) (om::object editor))
              ;; (setf (d-block (om::object editor)) (make-instance 'd :parent (om::object editor) (om::object editor)))
              (om::openeditorframe ; open a window displaying the editor of the first A block
```

```
        (om::omNG-make-new-instance (d-block (om::object editor)) "Window d")
       )
      )
     )
    )
  )
  (om::om-make-dialog-item
    'om::om-button
    (om::om-make-point 335 10) ; position (horizontal, vertical)
    (om::om-make-point 80 25) ; size (horizontal, vertical)
    "c"
    :di-action #'(lambda (b)
     (print "Selected c")
     (mp:process-run-function ; start a new thread for the execution of the next method
      "next thread" ; name of the thread, not necessary but useful for debugging
      nil ; process initialization keywords, not needed here
      (lambda () ; function to call
        (setf (parent (c-block (om::object editor))) (om::object editor))
       ;; (setf (c-block (om::object editor)) (make-instance 'c :parent (om::object editor) (om::object editor)))
       (om::openeditorframe ; open a window displaying the editor of the first A block
        (om::omNG-make-new-instance (c-block (om::object editor)) "Window c")
       )
      )
     )
    )
  )
 )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;           B PANEL          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;



(defun make-B-panel (editor B-panel)
 ;; (print "Block-position")
 ;; (print (block-position (om::object editor)))
 (om::om-add-subviews
  B-panel
  (om::om-make-dialog-item
```

```
        'om::om-button
        (om::om-make-point 5 10) ; position (horizontal, vertical)
        (om::om-make-point 80 25) ; size (horizontal, vertical)
        "s"
        :di-action #'(lambda (b)
          (print "Selected s")
          (mp:process-run-function ; start a new thread for the execution of the next
          ↪  method
            "next thread" ; name of the thread, not necessary but useful for debugging
            nil ; process initialization keywords, not needed here
            (lambda () ; function to call
              ;; (setf (s-block (om::object editor)) (make-instance 's :parent (om::object
              ↪  editor) (om::object editor)))
              (om::openeditorframe ; open a window displaying the editor of the first A
              ↪  block
                (om::omNG-make-new-instance (s-block (om::object editor)) "Window s")
              )
            )
          )
        )
    )
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 115 10) ; position (horizontal, vertical)
        (om::om-make-point 80 25) ; size (horizontal, vertical)
        "r"
        :di-action #'(lambda (b)
          (print "Selected r")
          (mp:process-run-function ; start a new thread for the execution of the next
          ↪  method
            "next thread" ; name of the thread, not necessary but useful for debugging
            nil ; process initialization keywords, not needed here
            (lambda () ; function to call
              ;; (setf (r-block (om::object editor)) (make-instance 'r :parent (om::object
              ↪  editor) (om::object editor)))
              (om::openeditorframe ; open a window displaying the editor of the first A
              ↪  block
                (om::omNG-make-new-instance (r-block (om::object editor)) "Window r")
              )
            )
          )
        )
    )
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 225 10) ; position (horizontal, vertical)
```

```lisp
        (om::om-make-point 80 25) ; size (horizontal, vertical)
        "d"
        :di-action #'(lambda (b)
          (print "Selected d")
          (mp:process-run-function ; start a new thread for the execution of the next method
            "next thread" ; name of the thread, not necessary but useful for debugging
            nil ; process initialization keywords, not needed here
            (lambda () ; function to call
              ;; (setf (d-block (om::object editor)) (make-instance 'd :parent (om::object editor) (om::object editor)))
              (om::openeditorframe ; open a window displaying the editor of the first A block
                (om::omNG-make-new-instance (d-block (om::object editor)) "Window d")
              )
            )
          )
        )
    )
    (om::om-make-dialog-item
        'om::om-button
        (om::om-make-point 335 10) ; position (horizontal, vertical)
        (om::om-make-point 80 25) ; size (horizontal, vertical)
        "c"
        :di-action #'(lambda (b)
          (print "Selected c")
          (mp:process-run-function ; start a new thread for the execution of the next method
            "next thread" ; name of the thread, not necessary but useful for debugging
            nil ; process initialization keywords, not needed here
            (lambda () ; function to call
              ;; (setf (c-block (om::object editor)) (make-instance 'c :parent (om::object editor) (om::object editor)))
              (om::openeditorframe ; open a window displaying the editor of the first A block
                (om::omNG-make-new-instance (c-block (om::object editor)) "Window c")
              )
            )
          )
        )
    )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
```

```lisp
;;                            CHANGES PANEL                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-changes-panel (editor panel)
  (om::om-add-subviews
    panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 10 10)
      (om::om-make-point 300 20)
      "Types of changes"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 30)
      (om::om-make-point 300 20)
      "Relative to rock"
      :checked-p (relative-to-parent (om::object editor))
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (setf (relative-to-parent (om::object editor)) 1)
                       (setf (relative-to-parent (om::object editor)) nil)
                     )
      )
    )

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 10 50)
      (om::om-make-point 300 20)
      "Relative to same type blocks"
      :checked-p (relative-to-same (om::object editor))
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (setf (relative-to-same (om::object editor)) 1)
                       (setf (relative-to-same (om::object editor)) nil)
                     )
      )
    )

  )
)

```

```lisp
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                     CONSTRAINTS PANELS                       ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; If first block of its type
(defun make-constraints-AB-panel (editor panel)
  (om::om-add-subviews
    panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 2)
      (om::om-make-point 120 20)
      "Block constraints"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 50)
      (om::om-make-point 200 20)
      "Number of bars"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 50)
      (om::om-make-point 80 20)
      "Bar length"
      :range (bar-length-range (om::object editor))
      :value (number-to-string (bar-length (om::object editor)))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (setf (bar-length (om::object editor)) (string-to-number check))
        (change-subblocks-values (om::object editor) :bar-length (bar-length (om::object editor)))
        (propagate-bar-length-srdc (om::object editor))
        (set-bar-length-up (om::object editor))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 100)
```

```
      (om::om-make-point 200 20)
      "Min note length"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 120 100)
      (om::om-make-point 20 20)
      ""
      :checked-p (min-note-length-flag (om::object editor))
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (setf (min-note-length-flag (om::object editor)) 1)
                       (setf (min-note-length-flag (om::object editor)) nil)
                     )
                     (change-subblocks-values (om::object editor)
                                  :min-note-length-flag (min-note-length-flag (om::object editor))
                                  :min-note-length (min-note-length (om::object editor)))
      )
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 100)
      (om::om-make-point 80 20); size
      "Minimum note length"
      :range  (loop :for n :from 0 :upto 4 :collect (number-to-string (expt 2 n)))
      :value (number-to-string (min-note-length (om::object editor)))
      :di-action #'(lambda (m)
        (let ((old-diff 0))
          (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
          (if (relative-to-same (om::object editor))
              (setq old-diff (diff-min-length (om::object editor)))
          )
          (setf (min-note-length (om::object editor)) (string-to-number check))
          (change-subblocks-values (om::object editor)
                                  :min-note-length-flag (min-note-length-flag (om::object editor))
                                  :min-note-length (min-note-length (om::object editor)))
          (if (relative-to-same (om::object editor))
              (propagate-AB (om::object editor) :diff-min-length (- old-diff (diff-min-length (om::object editor))))
          )
        )
```

```
      )
    )

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

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 120 150)
      (om::om-make-point 20 20)
      ""
      :checked-p (max-note-length-flag (om::object editor))
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (setf (max-note-length-flag (om::object editor)) 1)
                       (setf (max-note-length-flag (om::object editor)) nil)
                     )
                     (change-subblocks-values (om::object editor)
                                  :max-note-length-flag (max-note-length-flag (om::object
                                  ↪  editor))
                                  :max-note-length (max-note-length (om::object editor)))
      )
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 150)
      (om::om-make-point 80 20); size
      "Maximum note length"
      :range  (loop :for n :from 0 :upto 4 :collect (number-to-string (expt 2 n)))
      :value (number-to-string (max-note-length (om::object editor)))
      :di-action #'(lambda (m)
        (let ((old-diff 0))
          (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
          (if (relative-to-same (om::object editor))
              (setq old-diff (diff-max-length (om::object editor)))
          )
          (setf (max-note-length (om::object editor)) (string-to-number check))
          (change-subblocks-values (om::object editor)
                                  :max-note-length-flag (max-note-length-flag (om::object
                                  ↪  editor))
```

```
                                  :max-note-length (max-note-length (om::object editor)))
          (if (relative-to-same (om::object editor))
              (propagate-AB (om::object editor) :diff-max-length (- old-diff
              ↪  (diff-max-length (om::object editor))))
          )
        )
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 300 10)
      (om::om-make-point 200 20)
      "Pitch constraints"
      :font om::*om-default-font1b*
    )

    ; Key

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 300 50)
      (om::om-make-point 200 20)
      "Chord key"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 400 50)
      (om::om-make-point 80 20)
      "Chord key"
      :range '("C" "C#" "D" "Eb" "E" "F" "F#" "G" "Ab" "A" "Bb" "B")
      :value (chord-key (om::object editor))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-key (om::object editor)) nil)
          (setf (chord-key (om::object editor)) check)
        )
        (let ((old-diff 0))
          (if (relative-to-same (om::object editor))
              (setq old-diff (diff-chord-key (om::object editor)))
          )
          (change-subblocks-values (om::object editor) :chord-key check)
          (if (relative-to-same (om::object editor))
```

```lisp
            (propagate-AB (om::object editor) :diff-chord-key (- old-diff (diff-chord-key
            ↪  (om::object editor))))
          )
        )
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 300 100)
      (om::om-make-point 200 20)
      "Chord quality"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 400 100)
      (om::om-make-point 80 20)
      "Chord quality"
      :value (chord-quality (om::object editor))
      :range '("Major" "Minor" "Augmented" "Diminished")
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-quality (om::object editor)) nil)
          (setf (chord-quality (om::object editor)) check))
        (change-subblocks-values (om::object editor) :chord-quality check)

      )
    )

     (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 300 150)
      (om::om-make-point 200 20)
      "Minimum pitch"
      :font om::*om-default-font1b*
    )


    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 300 170)
      (om::om-make-point 150 20)
      "Minimum pitch"
```

```
      :range '(1 127)
      :increment 1
      :value (min-pitch (om::object editor))
      :di-action #'(lambda (s)
        (setf (min-pitch (om::object editor)) (om::om-slider-value s))
        (let ((old-diff 0))
          (if (relative-to-same (om::object editor))
              (setq old-diff (diff-min-pitch (om::object editor)))
          )
          (change-subblocks-values (om::object editor)
                                   :min-pitch (min-pitch (om::object editor)))
          (if (relative-to-same (om::object editor))
              (propagate-AB (om::object editor) :diff-min-pitch (- old-diff
              ↪  (diff-min-pitch (om::object editor))))
          )
        )
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 300 220)
      (om::om-make-point 200 20)
      "Maximum pitch"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 300 240)
      (om::om-make-point 150 20)
      "Maximum pitch"
      :range '(1 127)
      :increment 1
      :value (max-pitch (om::object editor))
      :di-action #'(lambda (s)
        (setf (max-pitch (om::object editor)) (om::om-slider-value s))
        (let ((old-diff 0))
          (if (relative-to-same (om::object editor))
              (setq old-diff (diff-max-pitch (om::object editor)))
          )
          (change-subblocks-values (om::object editor)
                                   :max-pitch (max-pitch (om::object editor)))
          (if (relative-to-same (om::object editor))
              (propagate-AB (om::object editor) :diff-max-pitch (- old-diff
              ↪  (diff-max-pitch (om::object editor))))
```

```
          )
        )
      )
    )
  )

)

;; If not first block of its type
(defun make-constraints-not-first-panel (editor panel)
  (let ((subviews '()))
    (setf subviews (append subviews (list
      (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 250 10)
      (om::om-make-point 200 20)
      "Pitch constraints"
      :font om::*om-default-font1b*
      )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 250 50)
      (om::om-make-point 200 20)
      "Chord key"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 350 50)
      (om::om-make-point 80 20)
      "Chord key"
      :range '("C" "C#" "D" "Eb" "E" "F" "F#" "G" "Ab" "A" "Bb" "B")
      :value (chord-key (om::object editor))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-key (om::object editor)) nil)
          (setf (chord-key (om::object editor)) check)
        )
        (let ((old-diff 0))
          (if (relative-to-same (om::object editor))
              (setq old-diff (diff-chord-key (om::object editor)))
          )
          (change-subblocks-values (om::object editor) :chord-key check)
```

```
          (if (relative-to-same (om::object editor))
            (propagate-AB (om::object editor) :diff-chord-key (- old-diff (diff-chord-key
            ↪  (om::object editor))))
          )
        )
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 250 100)
      (om::om-make-point 200 20)
      "Chord quality"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 350 100)
      (om::om-make-point 80 20)
      "Chord quality"
      :value (chord-quality (om::object editor))
      :range '("Major" "Minor" "Augmented" "Diminished")
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-quality (om::object editor)) nil)
          (setf (chord-quality (om::object editor)) check))
        (change-subblocks-values (om::object editor) :chord-quality check)

      )
    )

     (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 250 150)
      (om::om-make-point 200 20)
      "Minimum pitch"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 250 170)
      (om::om-make-point 150 20)
      "Minimum pitch"
```

```
    :range '(1 127)
    :increment 1
    :value (min-pitch (om::object editor))
    :di-action #'(lambda (s)
      (setf (min-pitch (om::object editor)) (om::om-slider-value s))
      (let ((old-diff 0))
        (if (relative-to-same (om::object editor))
            (setq old-diff (diff-min-pitch (om::object editor)))
        )
        (change-subblocks-values (om::object editor)
                                 :min-pitch (min-pitch (om::object editor)))
        (if (relative-to-same (om::object editor))
            (propagate-AB (om::object editor) :diff-min-pitch (- old-diff
            ↪  (diff-min-pitch (om::object editor))))
        )
      )
    )
  )

  (om::om-make-dialog-item
    'om::om-static-text
    (om::om-make-point 250 220)
    (om::om-make-point 200 20)
    "Maximum pitch"
    :font om::*om-default-font1b*
  )

  (om::om-make-dialog-item
    'om::slider
    (om::om-make-point 250 240)
    (om::om-make-point 150 20)
    "Maximum pitch"
    :range '(1 127)
    :increment 1
    :value (max-pitch (om::object editor))
    :di-action #'(lambda (s)
      (setf (max-pitch (om::object editor)) (om::om-slider-value s))
      (let ((old-diff 0))
        (if (relative-to-same (om::object editor))
            (setq old-diff (diff-max-pitch (om::object editor)))
        )
        (change-subblocks-values (om::object editor)
                                 :max-pitch (max-pitch (om::object editor)))
        (if (relative-to-same (om::object editor))
            (propagate-AB (om::object editor) :diff-max-pitch (- old-diff
            ↪  (diff-max-pitch (om::object editor))))
```

```
          )
        )
      )
    )

    )
    ))

    (if (typep (om::object editor) 'mldz::a)
      (setf subviews (append subviews (list
          (om::om-make-dialog-item
            'om::om-static-text
            (om::om-make-point 10 10)
            (om::om-make-point 200 20)
            "Similarity with first A block"
            :font om::*om-default-font1b*
          )
          (om::om-make-dialog-item
            'om::slider
            (om::om-make-point 10 40)
            (om::om-make-point 150 20)
            "Similarity with first A block"
            :range '(1 100)
            :increment 1
            :value (similarity-percent-A0 (om::object editor))
            :di-action #'(lambda (s)
              (setf (similarity-percent-A0 (om::object editor)) (om::om-slider-value s))
              (print "similarity-percent-A0: ")
              (print (similarity-percent-A0 (om::object editor)))
            )
          )
      )))
      (setf subviews (append subviews (list
          (om::om-make-dialog-item
            'om::om-static-text
            (om::om-make-point 10 10)
            (om::om-make-point 200 20)
            "Similarity with first B block"
            :font om::*om-default-font1b*
          )
          (om::om-make-dialog-item
            'om::slider
            (om::om-make-point 10 40)
            (om::om-make-point 150 20)
            "Similarity with first B block"
            :range '(1 100)
```

```
            :increment 1
            :value (similarity-percent-B0 (om::object editor))
            :di-action #'(lambda (s)
              (setf (similarity-percent-B0 (om::object editor)) (om::om-slider-value s))
              (print "similarity-percent-B0: ")
              (print (similarity-percent-B0 (om::object editor)))
            )
          )
      )))
    )

    (loop :for x :in subviews :do
      (om::om-add-subviews
          panel
          x
      )
    )
  )
)
```

D.2.3 sources/rock-srdc.lisp

This file contains the s, r, d and B objects. First by defining the objects and their attributes.

Then by defining the interfaces

```
(in-package :mldz)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          s CLASS                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(om::defclass! s ()
    (
      (parent
          :accessor parent :initarg :parent :initform nil
          :documentation "parent block from which the block comes from")
      (accomp
          :accessor accomp :initarg :accomp :initform (make-instance 'accompaniment)
          :documentation "acompaniment block for this part of the song")
      (relative-to-parent
```

```
            :accessor relative-to-parent :initarg :relative-to-parent :initform 1 :type integer
            :documentation "Flag to no if the cnahges in the attributes are relative to the parent block")
        (bar-length
            :accessor bar-length :initform 0 :type integer
            :documentation "Number of bars of this block")
        (min-note-length-flag
            :accessor min-note-length-flag :initform nil :type integer
            :documentation "Flag to post the minimum note length constraint")
        (min-note-length
            :accessor min-note-length :initform 1 :type integer
            :documentation "Minimum note length value")
        (diff-min-length
            :accessor diff-min-length :initform 0 :type integer
            :documentation "Difference for relative changes")
        (max-note-length-flag
            :accessor max-note-length-flag :initform nil :type integer
            :documentation "Flag to post the maximum note length constraint")
        (max-note-length
            :accessor max-note-length :initform 16 :type integer
            :documentation "Maximum note length value")
        (diff-max-length
            :accessor diff-max-length :initform 0 :type integer
            :documentation "Difference for relative changes")
        (chord-key
            :accessor chord-key :initform "C" :type string
            :documentation "key to set the scale in")
        (diff-chord-key
            :accessor diff-chord-key :initform 0 :type integer
            :documentation "Difference for relative changes")
        (chord-quality
            :accessor chord-quality :initform "Major" :type string
            :documentation "quality to set the scale in")
        (diff-chord-quality
            :accessor diff-chord-quality :initform 0 :type integer
            :documentation "Difference for relative changes")
        (min-pitch
            :accessor min-pitch :initform 1 :type integer
            :documentation "Minimum pitch value")
        (diff-min-pitch
            :accessor diff-min-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (max-pitch
            :accessor max-pitch :initform 127 :type integer
            :documentation "Maximum pitch value")
```

```lisp
      (diff-max-pitch
          :accessor diff-max-pitch :initform 0 :type integer
          :documentation "Difference for relative changes")
    )
)

(defclass s-editor (om::editorview) ())

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

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

(defmethod initialize-instance ((self s-editor) &rest args)
  ;;; do what needs to be done by default
  (call-next-method)
  (make-my-interface self)
)


(defmethod make-my-interface ((self s-editor))

  ; create the main view of the object
  (make-main-view self)

  (let*
    (
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different regions of the tool ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       (constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 300)
        :position (om::om-make-point 5 5)
        :bg-color om::*azulito*)
      )
      (accompaniment-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 300 300)
        :position (om::om-make-point 510 5)
        :bg-color om::*azulito*)
      )
```

```
    )

    (setf elements-constraints-panel (make-constraints-srdc-panel self constraints-panel))
    (setf elements-accompaniment-panel (make-accompaniment-panel self accompaniment-panel))

    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      constraints-panel
      accompaniment-panel
    )
  )
  ; return the editor
  self
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                            r CLASS                          ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(om::defclass! r ()
    (
      (parent
          :accessor parent :initarg :parent :initform nil
          :documentation "parent block from which the block comes from")
      (accomp
          :accessor accomp :initarg :accomp :initform (make-instance 'accompaniment)
          :documentation "acompaniment block for this part of the song")
      (relative-to-parent
          :accessor relative-to-parent :initarg :relative-to-parent :initform 1 :type integer
          :documentation "Flag to no if the cnahges in the attributes are relative to the parent block")
      (bar-length
          :accessor bar-length :initform 0 :type integer
          :documentation "Number of bars of this block")
      (min-note-length-flag
          :accessor min-note-length-flag :initform nil :type integer
          :documentation "Flag to post the minimum note length constraint")
      (min-note-length
          :accessor min-note-length :initform 1 :type integer
          :documentation "Minimum note length value")
```

```
        (diff-min-length
            :accessor diff-min-length :initform 0 :type integer
            :documentation "Difference for relative changes")
        (max-note-length-flag
            :accessor max-note-length-flag :initform nil :type integer
            :documentation "Flag to post the maximum note length constraint")
        (max-note-length
            :accessor max-note-length :initform 16 :type integer
            :documentation "Maximum note length value")
        (diff-max-length
            :accessor diff-max-length :initform 0 :type integer
            :documentation "Difference for relative changes")
        (chord-key
            :accessor chord-key :initform "C" :type string
            :documentation "key to set the scale in")
        (diff-chord-key
            :accessor diff-chord-key :initform 0 :type integer
            :documentation "Difference for relative changes")
        (chord-quality
            :accessor chord-quality :initform "Major" :type string
            :documentation "quality to set the scale in")
        (diff-chord-quality
            :accessor diff-chord-quality :initform 0 :type integer
            :documentation "Difference for relative changes")
        (min-pitch
            :accessor min-pitch :initform 1 :type integer
            :documentation "Minimum pitch value")
        (diff-min-pitch
            :accessor diff-min-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (max-pitch
            :accessor max-pitch :initform 127 :type integer
            :documentation "Maximum pitch value")
        (diff-max-pitch
            :accessor diff-max-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (similarity-percent-s
            :accessor similarity-percent-s :initform 50 :type integer
            :documentation "percentage of ressemblance with the s block of with the same
            ↪  parent")
        (semitones
          :accessor semitones :initform 0 :type integer
          :documentation "Semitones of transposition from the s-block of the same parent")
      )
)
```

```
(defclass r-editor (om::editorview) ())

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

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

(defmethod initialize-instance ((self r-editor) &rest args)
  ;;; do what needs to be done by default
  (call-next-method) ; start the search by default?
  (make-my-interface self)
)


(defmethod make-my-interface ((self r-editor))

  ; create the main view of the object
  (make-main-view self)

  (let*
    (
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different regions of the tool ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       (constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 195)
        :position (om::om-make-point 5 5)
        :bg-color om::*azulito*)
      )
      (r-constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 100)
        :position (om::om-make-point 5 205)
        :bg-color om::*azulito*)
      )
      (accompaniment-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 300 300)
        :position (om::om-make-point 510 5)
        :bg-color om::*azulito*)
      )
    )
```

```

    (setf elements-constraints-panel (make-constraints-srdc-panel self constraints-panel))
    (setf elements-accompaniment-panel (make-accompaniment-panel self
    ↪  accompaniment-panel))
    (setf elements-r-constraints-panel (make-r-constraints-panel self
    ↪  r-constraints-panel))

    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      constraints-panel
      accompaniment-panel
      r-constraints-panel
    )
  )
  ; return the editor
  self
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                          d CLASS                            ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(om::defclass! d ()
    (
      (parent
          :accessor parent :initarg :parent :initform nil
          :documentation "parent block from which the block comes from")
      (accomp
          :accessor accomp :initarg :accomp :initform (make-instance 'accompaniment)
          :documentation "acompaniment block for this part of the song")
      (relative-to-parent
          :accessor relative-to-parent :initarg :relative-to-parent :initform 1 :type
          ↪  integer
          :documentation "Flag to no if the cnahges in the attributes are relative to the
          ↪  parent block")
      (bar-length
          :accessor bar-length :initform 0 :type integer
          :documentation "Number of bars of this block")
      (min-note-length-flag
          :accessor min-note-length-flag :initform nil :type integer
          :documentation "Flag to post the minimum note length constraint")
      (min-note-length
```

```
        :accessor min-note-length :initform 1 :type integer
        :documentation "Minimum note length value")
    (diff-min-length
        :accessor diff-min-length :initform 0 :type integer
        :documentation "Difference for relative changes")
    (max-note-length-flag
        :accessor max-note-length-flag :initform nil :type integer
        :documentation "Flag to post the maximum note length constraint")
    (max-note-length
        :accessor max-note-length :initform 16 :type integer
        :documentation "Maximum note length value")
    (diff-max-length
        :accessor diff-max-length :initform 0 :type integer
        :documentation "Difference for relative changes")
    (chord-key
        :accessor chord-key :initform "C" :type string
        :documentation "key to set the scale in")
    (diff-chord-key
        :accessor diff-chord-key :initform 0 :type integer
        :documentation "Difference for relative changes")
    (chord-quality
        :accessor chord-quality :initform "Major" :type string
        :documentation "quality to set the scale in")
    (diff-chord-quality
        :accessor diff-chord-quality :initform 0 :type integer
        :documentation "Difference for relative changes")
    (min-pitch
        :accessor min-pitch :initform 1 :type integer
        :documentation "Minimum pitch value")
    (diff-min-pitch
        :accessor diff-min-pitch :initform 0 :type integer
        :documentation "Difference for relative changes")
    (max-pitch
        :accessor max-pitch :initform 127 :type integer
        :documentation "Maximum pitch value")
    (diff-max-pitch
        :accessor diff-max-pitch :initform 0 :type integer
        :documentation "Difference for relative changes")
    (difference-percent-s
        :accessor difference-percent-s :initform 75 :type integer
        :documentation "percentage of difference with the s block of with the same parent")
    (semitones
      :accessor semitones :initform 0 :type integer
      :documentation "Semitones of transposition from the s-block of the same parent")
  )
```

```
)

(defclass d-editor (om::editorview) ())

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

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

(defmethod initialize-instance ((self d-editor) &rest args)
  ;;; do what needs to be done by default
  (call-next-method)
  (make-my-interface self)
)

(defmethod make-my-interface ((self d-editor))

  ; create the main view of the object
  (make-main-view self)

  (let*
    (
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different regions of the tool ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
       (constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 195)
        :position (om::om-make-point 5 5)
        :bg-color om::*azulito*)
      )
      (accompaniment-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 300 300)
        :position (om::om-make-point 510 5)
        :bg-color om::*azulito*)
      )
      (d-constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 100)
        :position (om::om-make-point 5 205)
        :bg-color om::*azulito*)
      )
```

```
    )

    (setf elements-d-constraints-panel (make-d-constraints-panel self d-constraints-panel))

    ; add the subviews for the different parts into the main view
    (setf elements-constraints-panel (make-constraints-srdc-panel self constraints-panel))
    (setf elements-accompaniment-panel (make-accompaniment-panel self accompaniment-panel))

    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      constraints-panel
      accompaniment-panel
      d-constraints-panel
    )
  )
  ; return the editor
  self
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                           c CLASS                           ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(om::defclass! c ()
    (
      (parent
          :accessor parent :initarg :parent :initform nil
          :documentation "parent block from which the block comes from")
      (accomp
          :accessor accomp :initarg :accomp :initform (make-instance 'accompaniment)
          :documentation "acompaniment block for this part of the song")
      (relative-to-parent
          :accessor relative-to-parent :initarg :relative-to-parent :initform 1 :type integer
          :documentation "Flag to no if the cnahges in the attributes are relative to the parent block")
      (bar-length
          :accessor bar-length :initform 0 :type integer
          :documentation "Number of bars of this block")
      (min-note-length-flag
```

```
            :accessor min-note-length-flag :initform nil :type integer
            :documentation "Flag to post the minimum note length constraint")
        (min-note-length
            :accessor min-note-length :initform 1 :type integer
            :documentation "Minimum note length value")
        (diff-min-length
            :accessor diff-min-length :initform 0 :type integer
            :documentation "Difference for relative changes")
        (max-note-length-flag
            :accessor max-note-length-flag :initform nil :type integer
            :documentation "Flag to post the maximum note length constraint")
        (max-note-length
            :accessor max-note-length :initform 16 :type integer
            :documentation "Maximum note length value")
        (diff-max-length
            :accessor diff-max-length :initform 0 :type integer
            :documentation "Difference for relative changes")
        (chord-key
            :accessor chord-key :initform "C" :type string
            :documentation "key to set the scale in")
        (diff-chord-key
            :accessor diff-chord-key :initform 0 :type integer
            :documentation "Difference for relative changes")
        (chord-quality
            :accessor chord-quality :initform "Major" :type string
            :documentation "quality to set the scale in")
        (diff-chord-quality
            :accessor diff-chord-quality :initform 0 :type integer
            :documentation "Difference for relative changes")
        (min-pitch
            :accessor min-pitch :initform 1 :type integer
            :documentation "Minimum pitch value")
        (diff-min-pitch
            :accessor diff-min-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (max-pitch
            :accessor max-pitch :initform 127 :type integer
            :documentation "Maximum pitch value")
        (diff-max-pitch
            :accessor diff-max-pitch :initform 0 :type integer
            :documentation "Difference for relative changes")
        (cadence-type
            :accessor cadence-type :initform "Perfect" :type string
            :documentation "Type of cadence used in the current block")
        (min-note-length-mult
            :accessor min-note-length-mult :initform 2 :type integer
```

```
          :documentation "Multiplicator to slow down the song")
    )
)

(defclass c-editor (om::editorview) ())

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

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

(defmethod initialize-instance ((self c-editor) &rest args)
  ;;; do what needs to be done by default
  (call-next-method) ; start the search by default?
  (make-my-interface self)
)

(defmethod make-my-interface ((self c-editor))

  ; create the main view of the object
  (make-main-view self)

  (let*
    (
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
      ;;; setting the different regions of the tool ;;;
      ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

      (constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 300)
        :position (om::om-make-point 5 5)
        :bg-color om::*azulito*)
      )
      (c-constraints-panel (om::om-make-view 'om::om-view
        :size (om::om-make-point 500 100)
        :position (om::om-make-point 5 310)
        :bg-color om::*azulito*)
      )
    )

```

```

    (setf elements-c-constraints-panel (make-c-constraints-panel self
    ↪  c-constraints-panel))

    ; add the subviews for the different parts into the main view
    (setf elements-constraints-panel (make-constraints-srdc-panel self constraints-panel))

    ; add the subviews for the different parts into the main view
    (om::om-add-subviews
      self
      constraints-panel
      c-constraints-panel
    )
  )
  ; return the editor
  self
)

```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                     r CONSTRAINTS PANEL                      ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-r-constraints-panel (editor panel)
  (om::om-add-subviews
    panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 10 10)
      (om::om-make-point 200 20)
      "Similarity with s block"
      :font om::*om-default-font1b*
    )
    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 10 40)
      (om::om-make-point 150 20)
      "Similarity with s block"
      :range '(1 100)
      :increment 1
      :value (similarity-percent-s (om::object editor))
```

```
      :di-action #'(lambda (s)
        (setf (similarity-percent-s (om::object editor)) (om::om-slider-value s))
        (print "similarity-percent-s: ")
        (print (similarity-percent-s (om::object editor)))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 200 10)
      (om::om-make-point 100 50)
      "Semitones from s block"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 300 10)
      (om::om-make-point 80 20)
      "semitones from s block"
      :range (loop :for i :from -12 :below 12 :collect (number-to-string i))
      :value (number-to-string (semitones (om::object editor)))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (setf (semitones (om::object editor)) (string-to-number check))
      )
    )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      c CONSTRAINTS PANEL                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-c-constraints-panel (editor panel)
  (om::om-add-subviews
    panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 10 10)
      (om::om-make-point 200 20)
      "Cadence choice"
      :font om::*om-default-font1b*
    )
```

```
    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 10 40)
      (om::om-make-point 150 20)
      "Cadence choice"
      :range '("Perfect" "Plagal" "Semi" "None")
      :value (cadence-type (om::object editor))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (cadence-type (om::object editor)) "None")
          (setf (cadence-type (om::object editor)) check)
        )
      )
    )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                      d CONSTRAINTS PANEL                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
(defun make-d-constraints-panel (editor panel)
  (om::om-add-subviews
    panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 10 10)
      (om::om-make-point 200 20)
      "Difference with s block"
      :font om::*om-default-font1b*
    )
    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 10 40)
      (om::om-make-point 150 20)
      "Difference with s block"
      :range '(1 100)
      :increment 1
      :value (difference-percent-s (om::object editor))
      :di-action #'(lambda (s)
        (setf (difference-percent-s (om::object editor)) (om::om-slider-value s))
        (print "difference-percent-s: ")
        (print (difference-percent-s (om::object editor)))
      )
```

```
    )
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 200 10)
      (om::om-make-point 100 50)
      "Semitones from s block"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 300 10)
      (om::om-make-point 80 20)
      "semitones from s block"
      :range (loop :for i :from -12 :below 12 :collect (number-to-string i))
      :value (number-to-string (semitones (om::object editor)))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (setf (semitones (om::object editor)) (string-to-number check))
      )
    )
  )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                    srdc CONSTRAINTS PANEL                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun make-constraints-srdc-panel (editor panel)
  (om::om-add-subviews
    panel
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 10)
      (om::om-make-point 120 20)
      "Block constraints"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 50)
      (om::om-make-point 200 20)
      "Number of bars"
```

```lisp
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 50)
      (om::om-make-point 80 20)
      "Bar length"
      :range (bar-length-range (om::object editor))
      :value (number-to-string (bar-length (om::object editor)))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (setf (bar-length (om::object editor)) (string-to-number check))
        (set-bar-length-up (om::object editor))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 100)
      (om::om-make-point 200 20)
      "Min note length"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 120 100)
      (om::om-make-point 20 20)
      ""
      :checked-p (min-note-length-flag (om::object editor))
      :di-action #'(lambda (c)
                  (if (om::om-checked-p c)
                    (setf (min-note-length-flag (om::object editor)) 1)
                    (setf (min-note-length-flag (om::object editor)) nil)
                  )
      )
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 100)
      (om::om-make-point 80 20); size
      "Minimum note length"
      :range  (loop :for n :from 0 :upto 4 :collect (number-to-string (expt 2 n)))
      :value (number-to-string (min-note-length (om::object editor)))
```

```
      :di-action #'(lambda (m)
        (let ((old-diff 0))
          (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
          (setf (min-note-length (om::object editor)) (string-to-number check))
        )
      )
    )

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

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 120 150)
      (om::om-make-point 20 20)
      ""
      :checked-p (max-note-length-flag (om::object editor))
      :di-action #'(lambda (c)
                    (if (om::om-checked-p c)
                      (setf (max-note-length-flag (om::object editor)) 1)
                      (setf (max-note-length-flag (om::object editor)) nil)
                    )
      )
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 170 150)
      (om::om-make-point 80 20); size
      "Maximum note length"
      :range  (loop :for n :from 0 :upto 4 :collect (number-to-string (expt 2 n)))
      :value (number-to-string (max-note-length (om::object editor)))
      :di-action #'(lambda (m)
        (let ((old-diff 0))
          (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
          (setf (max-note-length (om::object editor)) (string-to-number check))        )
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
```

```
      (om::om-make-point 300 10)
      (om::om-make-point 200 20)
      "Pitch constraints"
      :font om::*om-default-font1b*
    )

     (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 300 50)
      (om::om-make-point 200 20)
      "Minimum pitch"
      :font om::*om-default-font1b*
    )


    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 300 70)
      (om::om-make-point 150 20)
      "Minimum pitch"
      :range '(1 127)
      :increment 1
      :value (min-pitch (om::object editor))
      :di-action #'(lambda (s)
        (setf (min-pitch (om::object editor)) (om::om-slider-value s))
      )
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 300 120)
      (om::om-make-point 200 20)
      "Maximum pitch"
      :font om::*om-default-font1b*
    )


    (om::om-make-dialog-item
      'om::slider
      (om::om-make-point 300 140)
      (om::om-make-point 150 20)
      "Maximum pitch"
      :range '(1 127)
      :increment 1
      :value (max-pitch (om::object editor))
      :di-action #'(lambda (s)
```

```
        (setf (max-pitch (om::object editor)) (om::om-slider-value s))
      )
    )
  )

)
```

D.2.4 sources/rock-accompaniment.lisp

This file contains the object describing the accompaniment.

```
(in-package :mldz)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;                        ACCOMPANIMENT CLASS                        ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(om::defclass! accompaniment ()
    (
      (parent
          :accessor parent :initarg :parent :initform nil
          :documentation "parent block containing the instance of this block")
      (relative-to-parent
          :accessor relative-to-parent :initarg :relative-to-parent :initform 1 :type
          ↪  integer
          :documentation "Flag to now if the block attributes are reltive to its
          ↪  parent's")
      (bar-length
          :accessor bar-length :initform 0 :type integer
          :documentation "Number of bars of the block")
      (min-simultaneous-notes
          :accessor min-simultaneous-notes :initform 3 :type integer
          :documentation "Minimum notes played simultaneously")
      (diff-max-sim
          :accessor diff-max-sim :initform 0 :type integer
          :documentation "Difference for relative changes")
      (max-simultaneous-notes
          :accessor max-simultaneous-notes :initform 3 :type integer
          :documentation "Maximum notes played simultaneously")
      (diff-min-sim
          :accessor diff-min-sim :initform 0 :type integer
```

```
          :documentation "Difference for relative changes")
      (min-note-length-flag
          :accessor min-note-length-flag :initform 1 :type integer
          :documentation "Flag stating if the note-min-length constrain must be posted")
      (min-note-length
          :accessor min-note-length :initform 16 :type integer
          :documentation "Minimum note length value")
      (diff-min-length
          :accessor diff-min-length :initform 0 :type integer
          :documentation "Difference for relative changes")
      (max-note-length-flag
          :accessor max-note-length-flag :initform 1 :type integer
          :documentation "Flag stating if the note-max-length constrain must be posted")
      (max-note-length
          :accessor max-note-length :initform 16 :type integer
          :documentation "Maximum note length value")
      (diff-max-length
          :accessor diff-max-length :initform 0 :type integer
          :documentation "Difference for relative changes")
      (chord-key
          :accessor chord-key :initform "C" :type string
          :documentation "Chord key to set the scale in")
      (diff-chord-key
          :accessor diff-chord-key :initform 0 :type integer
          :documentation "Difference for relative changes")
      (chord-quality
          :accessor chord-quality :initform "Major" :type string
          :documentation "Quality to set the scale in")
      (diff-chord-quality
          :accessor diff-chord-quality :initform 0 :type integer
          :documentation "Difference for relative changes")
      (min-pitch
          :accessor min-pitch :initform 1 :type integer
          :documentation "Minimum pitch value")
      (diff-min-pitch
          :accessor diff-min-pitch :initform 0 :type integer
          :documentation "Difference for relative changes")
      (max-pitch
          :accessor max-pitch :initform 127 :type integer
          :documentation "Maximum pitch value")
      (diff-max-pitch
          :accessor diff-max-pitch :initform 0 :type integer
          :documentation "Difference for relative changes")
    )
)
```

```
(defun make-accompaniment-panel (editor panel)
  (om::om-add-subviews
    panel
  (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 10)
      (om::om-make-point 200 20)
      "Accompaniment constraints"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 50)
      (om::om-make-point 200 20)
      "Min note length"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 145 50)
      (om::om-make-point 20 20)
      ""
      :checked-p (min-note-length-flag (accomp (om::object editor)))
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (setf (min-note-length-flag (accomp (om::object editor))) 1)
                       (setf (min-note-length-flag (accomp (om::object editor))) nil)
                     )
      )
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 165 50)
      (om::om-make-point 80 20); size
      "Min note length"
      :range  (loop :for n :from 0 :upto 4 :collect (number-to-string (expt 2 n)))
      :value (number-to-string (min-note-length (accomp (om::object editor))))
      :di-action #'(lambda (m)
          (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
          (setf (min-note-length (accomp (om::object editor))) (string-to-number check))
      )
    )

```

```lisp
    (om::om-make-dialog-item
      'om::om-static-text
      (om::om-make-point 15 100)
      (om::om-make-point 200 20)
      "Max note length"
      :font om::*om-default-font1b*
    )

    (om::om-make-dialog-item
      'om::om-check-box
      (om::om-make-point 145 100)
      (om::om-make-point 20 20)
      ""
      :checked-p (max-note-length-flag (accomp (om::object editor)))
      :di-action #'(lambda (c)
                     (if (om::om-checked-p c)
                       (setf (max-note-length-flag (accomp (om::object editor))) 1)
                       (setf (max-note-length-flag (accomp (om::object editor))) nil)
                     )
      )
    )

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 165 100)
      (om::om-make-point 80 20); size
      "Max note length"
      :range  (loop :for n :from 0 :upto 4 :collect (number-to-string (expt 2 n)))
      :value (number-to-string (max-note-length (accomp (om::object editor))))
      :di-action #'(lambda (m)
          (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
          (setf (max-note-length (accomp (om::object editor))) (string-to-number check))
      )
    )

    ; Key

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

    (om::om-make-dialog-item
```

```
      'om::pop-up-menu
      (om::om-make-point 165 150)
      (om::om-make-point 80 20)
      "Chord key"
      :range '("C" "C#" "D" "Eb" "E" "F" "F#" "G" "Ab" "A" "Bb" "B")
      :value (chord-key (accomp (om::object editor)))
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-key (accomp (om::object editor))) nil)
          (setf (chord-key (accomp (om::object editor))) check)
        )
      )
    )

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

    (om::om-make-dialog-item
      'om::pop-up-menu
      (om::om-make-point 165 200)
      (om::om-make-point 80 20)
      "Chord quality"
      :value (chord-quality (accomp (om::object editor)))
      :range '("Major" "Minor" "Augmented" "Diminished")
      :di-action #'(lambda (m)
        (setq check (nth (om::om-get-selected-item-index m) (om::om-get-item-list m)))
        (if (string= check "None")
          (setf (chord-quality (accomp (om::object editor))) nil)
          (setf (chord-quality (accomp (om::object editor))) check))
      )
    )
  )
)
```

D.3 CSP Files

This section will contain the code for the Constraint Satisfaction problem described in chapter 4. It is distributed over two files:

- **rock-csp.lisp** contains the function to create the search space and its variables, the function to post the constraints given in the constraints panels from the interface, and the functions to get the next solution or stop the search.
- **rock-csts.lisp** contains the constraints implementation as described in appendix C, but in Lisp this time.

D.3.1 sources/rock-csp.lisp

This file is the first file creating the search and handling the solutions.

```lisp
(in-package :mldz)

;;;;;;;;;;;;;;;;;
; NEW-MELODIZER ;
;;;;;;;;;;;;;;;;;

; <rock-csp> the rock object defining the constraints
; <percent-diff> percentage of difference wanted for the solutions
; This function creates the CSP by creating the space and the variables, posting the constraints and the branching, specifying
; the search options and creating the search engine.
(defmethod rock-solver (rock-csp percent-diff branching)
    (let ((sp (gil::new-space)); create the space;
        push pull playing push-acc pull-acc playing-acc
        tstop sopts temp
        pos

        (max-pitch 127)
        (bars (bar-length rock-csp))
        (quant 16)
        )

        ;Setting constraint for this block and child blocks
        (setq temp (constrain-rock sp rock-csp))
        (setq push (nth 0 temp))
        (setq pull (nth 1 temp))
        (setq playing (nth 2 temp))
        (setq push-acc (nth 3 temp))
        (setq pull-acc (nth 4 temp))
        (setq playing-acc (nth 5 temp))

        ;; Define branching for BAB
        (gil::g-branch sp push gil::INT_VAR_SIZE_MIN gil::INT_VAL_RND)
        (gil::g-branch sp pull gil::INT_VAR_SIZE_MIN gil::INT_VAL_RND)
        (gil::g-branch sp playing gil::INT_VAR_SIZE_MIN gil::INT_VAL_RND)
        (gil::g-branch sp push-acc gil::SET_VAR_SIZE_MIN gil::SET_VAL_RND_INC)
```

```
        (gil::g-branch sp pull-acc gil::SET_VAR_SIZE_MIN gil::SET_VAL_RND_INC)
        (gil::g-branch sp playing-acc gil::SET_VAR_SIZE_MIN gil::SET_VAL_RND_INC)

        (gil::g-specify-sol-variables sp playing)
        (gil::g-specify-percent-diff sp percent-diff)

        ;time stop
        (setq tstop (gil::t-stop)); create the time stop object
        (gil::time-stop-init tstop 5000); initialize it (time is expressed in ms)

        ;search options
        (setq sopts (gil::search-opts)); create the search options object
        (gil::init-search-opts sopts); initialize it
        (gil::set-n-threads sopts 1); set the number of threads to be used during the search (default is 1, 0 means as many as available)
        (gil::set-time-stop sopts tstop); set the timestop object to stop the search if it takes too long

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

        (print "new-melodizer basic CSP constructed")

        ; return
        (list se push pull playing push-acc pull-acc playing-acc tstop sopts bars quant sp)
    )
)

;recursive function to set the constraint on all the blocks in the tree structure
; TODO : adapt function for A A B A and launch functions for s r d c
(defun constrain-rock (sp rock-csp)
    (print "At the start of constrain-rock")

    ; return pull push playing
    (let (pull push playing pull-acc push-acc playing-acc block-list positions
        sub-push sub-pull pitches-notes lengths-notes

         (bars (bar-length rock-csp))
         (quant 16)
         (max-pitch 127)
         (max-simultaneous-notes 10)
         (min-simultaneous-notes 0)
         (no-note -1)
         (startidx 0)
         nb-notes push-A0 push-B0
```

```lisp
        )

        (setq nb-notes (+ (* bars quant) 1))

        ;; initialize the variables
        (setq push (gil::add-int-var-array sp nb-notes no-note max-pitch))
        (setq pull (gil::add-int-var-array sp nb-notes no-note max-pitch))
        (setq playing (gil::add-int-var-array sp nb-notes no-note max-pitch))

        (setq push-acc (gil::add-set-var-array sp nb-notes 0 max-pitch 0
        ↪  max-simultaneous-notes))
        (setq pull-acc (gil::add-set-var-array sp nb-notes 0 max-pitch 0
        ↪  max-simultaneous-notes))
        (setq playing-acc (gil::add-set-var-array sp nb-notes 0 max-pitch
        ↪  min-simultaneous-notes max-simultaneous-notes))

        ;; connects push pull and playing with constraints
        (link-push-pull-playing-int sp push pull playing max-pitch)
        ;; Limit intervals between consecutive notes
        (limit-intervals-cst sp playing)
        (link-push-pull-playing-set sp push-acc pull-acc playing-acc max-pitch
        ↪  max-simultaneous-notes)



        ;; set constraints on push pull and playing from all blocks in the structure
        (setq block-list (block-list rock-csp))

        ;; iterate over all blocks A and B in block-list
        (loop :for i :from 0 :below (length block-list) :by 1 :do
            ;; for every A/B block, post constraints from s,r,d,c
            ;; cut the push pull playing array into (length block-list) parts and feed the
            ↪  adequate part
            ;; to (constrain-ppp-from-srdc)
            (let (temp-push temp-pull temp-playing temp-push-acc temp-pull-acc
            ↪  temp-playing-acc
                  srdc-parent notes-per-block)
                (setq srdc-parent (nth i block-list))
                (setq notes-per-block (* (bar-length srdc-parent) quant))
                (setq temp-push (sublst push startidx notes-per-block))
                (setq temp-pull (sublst pull startidx notes-per-block))
                (setq temp-playing (sublst playing startidx notes-per-block))
                (setq temp-push-acc (sublst push-acc startidx notes-per-block))
                (setq temp-pull-acc (sublst pull-acc startidx notes-per-block))
                (setq temp-playing-acc (sublst playing-acc startidx notes-per-block))
                (if (= i (idx-first-a rock-csp))
```

```lisp
                    (setq push-A0 temp-push)
                )
                (if (= i (idx-first-b rock-csp))
                    (setq push-B0 temp-push)
                )
                (if (> startidx 0)
                    (progn
                        ;; Last played note of the previous block must be pulled
                        (gil::g-rel sp (first temp-pull) gil::IRT_EQ (nth (- startidx 1) playing))
                    )
                )

                (constrain-srdc-from-parent srdc-parent temp-push temp-pull temp-playing
                                            temp-push-acc temp-pull-acc temp-playing-acc push-A0 push-B0 quant max-pitch sp)
                (setq startidx (+ startidx notes-per-block))
            )
        )

        ;; return
        (list push pull playing push-acc pull-acc playing-acc)
    )
)

;posts the constraints specified in the block
(defun post-rock-constraints (sp rock push pull playing is-cadence post-chord)
    (print "posting rock constraints")
    (if (typep rock 'mldz::accompaniment);; Only accompaniment is polymorphique
        (progn
            (if (and (min-simultaneous-notes rock) (typep (nth 0 push) 'gil::set-var))
                (gil::g-card sp playing (min-simultaneous-notes rock) (max-simultaneous-notes rock))
            )
            (if (and (max-simultaneous-notes rock) (typep (nth 0 push) 'gil::set-var))
                (gil::g-card sp playing (min-simultaneous-notes rock) (max-simultaneous-notes rock))
            )
        )
    )

    (cond
        ((not (typep rock 'mldz::accompaniment))
            (progn
                ; Pitch constraints
                (if (and post-chord (chord-key rock))
```

```
                    (if (typep (nth 0 push) 'gil::set-var)
                        (chord-key-cst sp push rock)
                        (chord-key-cst-int sp push playing rock)
                    )
                )

                (if (min-note-length-flag rock)
                    (if is-cadence
                        (note-min-length-rock sp push pull playing (smallest 16 (* (min-note-length-mult rock) (min-note-length rock))))
                        (note-min-length-rock sp push pull playing (min-note-length rock))
                    )
                    (if is-cadence
                        (note-min-length-rock sp push pull playing (min-note-length-mult rock))
                        (note-min-length-rock sp push pull playing 1)
                    )
                )

                (if (max-note-length-flag rock)
                    (if is-cadence
                        (note-max-length-rock sp push pull (biggest (max-note-length rock) (* (min-note-length-mult rock) (min-note-length rock))))
                        (note-max-length-rock sp push pull (max-note-length rock))
                    )
                    (if is-cadence
                        (note-max-length-rock sp push pull 16)
                        (note-max-length-rock sp push pull 16)
                    )
                )

            )
        )


        ((and    is-cadence
                (typep rock 'mldz::accompaniment))
            (progn
                ; Time constraints
                (if (min-note-length-flag rock)
                    (note-min-length-rock sp push pull playing (* (/ (min-note-length rock) 2) (bar-length rock)))
                )

                (if (max-note-length-flag rock)
                    (note-max-length-rock sp push pull (* (/ (max-note-length rock) 2) (bar-length rock)))
```

```
                )
            )
        )

        ((and    (not is-cadence)
                (typep rock 'mldz::accompaniment))

            (progn
                ; Pitch constraints
                (if (and post-chord (chord-key rock))
                    (if (typep (nth 0 push) 'gil::set-var)
                        (chord-key-cst sp playing rock)
                        (chord-key-cst-int sp push playing rock)
                    )
                )
                ; Time constraints
                (if (min-note-length-flag rock)
                    (note-min-length-rock sp push pull playing (min-note-length rock))
                )

                (if (max-note-length-flag rock)
                    (note-max-length-rock sp push pull (max-note-length rock))
                )
            )
        )

    )

    (pitch-range sp push (min-pitch rock) (max-pitch rock))

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

; <l> is a list containing the search engine for the problem and the variables
; <rock-object> is a rock object
; this function finds the next solution of the CSP using the search engine given as an argument
(defmethod new-rock-next (l rock-object)
    (let ((se (nth 0 l))
         (push (nth 1 l))
         (pull (nth 2 l))
         (playing (nth 3 l))
         (push-acc (nth 4 l))
         (pull-acc (nth 5 l))
```

```
            (playing-acc (nth 6 l))
            (tstop (nth 7 l))
            (sopts (nth 8 l))
            (bars (nth 9 l))
            (quant (nth 10 l))
            (sp (nth 11 l))
            (check t); for the while loop
            sol score-voice score-acc)

        (print "in search rock")
        (gil::time-stop-reset tstop);reset the tstop timer before launching the search

        (om::while check :do

            (setq sol (gil::search-next se)); search the next solution
            (if (null sol)
                (stopped-or-ended (gil::stopped se) (stop-search rock-object) tstop); check if there are solutions left and if the user wishes to continue searching
                (setf check nil); we have found a solution so break the loop
            )
        )

        ;créer score qui retourne la liste de pitch et la rhythm tree
        (setq score-voice (build-voice-int sol push pull playing bars quant (tempo rock-object)))
        (setq score-acc (build-voice sol push-acc pull-acc bars quant (tempo rock-object)))

        (list
            (make-instance 'om::poly
                :voices (list
                            (make-instance 'om::voice
                                :chords (first score-voice)
                                :tree (second score-voice)
                                :tempo (tempo rock-object)
                            )
                            (make-instance 'om::voice
                                :chords (first score-acc)
                                :tree (second score-acc)
                                :tempo (tempo rock-object)
                            )
                        )
            )

            se push pull playing push-acc pull-acc playing-acc tstop sopts bars quant sp)
```

```lisp

    )
)

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

D.3.2 sources/rock-csts.lisp

This file contains the implementation of the constraints in Common Lisp.

```lisp
(in-package :mldz)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Link arrays of music representation ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Post the constraints to link the three arrays of the representation when using SetVar
(defun link-push-pull-playing-set (sp push pull playing max-pitch max-simultaneous-notes)
    ;initial constraint on pull, push, playing and durations
    (gil::g-empty sp (first pull)) ; pull[0] == empty
    (gil::g-rel sp (first push) gil::SRT_EQ (first playing)) ; push[0] == playing [0]

    ;connect push, pull and playing
    (loop :for j :from 1 :below (length push) :do ;for each interval
        (let (temp z c)
            (setq temp (gil::add-set-var sp 0 max-pitch 0 max-simultaneous-notes));
            ↪  temporary variables
            (gil::g-op sp (nth (- j 1) playing) gil::SOT_MINUS (nth j pull) temp); temp[0]
            ↪  = playing[j-1] - pull[j]
            (gil::g-op sp temp gil::SOT_UNION (nth j push) (nth j playing)); playing[j] ==
            ↪  playing[j-1] - pull[j] + push[j] Playing note
            (gil::g-rel sp (nth j pull) gil::SRT_SUB (nth (- j 1) playing)) ; pull[j] <=
            ↪  playing[j-1] cannot pull a note not playing
```

```lisp
                (gil::g-set-op sp (nth (- j 1) playing) gil::SOT_MINUS (nth j pull) gil::SRT_DISJ (nth j push)); push[j] || playing[j-1] - pull[j] Cannot push a note still playing
            )
        )
)

;; Post the constraints to link the three arrays of the representation when using IntVar
(defun link-push-pull-playing-int (sp push pull playing max-pitch)
    ;initial constraint on pull, push, playing and durations
    (gil::g-rel sp (first pull) gil::IRT_EQ -1) ; pull[0] == empty
    (gil::g-rel sp (first push) gil::IRT_EQ (first playing)) ; push[0] == playing [0]

    (loop :for j :from 16 :below (length push) :by 16 :do
        (gil::g-rel sp (nth j pull) gil::IRT_EQ (nth (- j 1) playing))
    )

    ;connect push, pull and playing
    (loop :for j :from 1 :below (length push) :do ;for each interval
        (let (
            playing-j-playing-j-one
            push-j-pull-j
            push-j-playing-j
            pull-j-playing-j-one
            pull-j-one
            push-j-one
            push-j-nq-one
            playing-j-one
            )
            (setq
                playing-j-playing-j-one (gil::add-bool-var-expr sp (nth j playing) gil::IRT_EQ (nth (- j 1) playing))
                push-j-pull-j (gil::add-bool-var-expr sp (nth j push) gil::IRT_EQ (nth j pull))
                push-j-playing-j (gil::add-bool-var-expr sp (nth j push) gil::IRT_EQ (nth j playing))
                pull-j-playing-j-one (gil::add-bool-var-expr sp (nth j pull) gil::IRT_EQ (nth (- j 1) playing))
                pull-j-one (gil::add-bool-var-expr sp (nth j pull) gil::IRT_EQ -1)
                push-j-one (gil::add-bool-var-expr sp (nth j push) gil::IRT_EQ -1)
                push-j-nq-one (gil::add-bool-var-expr sp (nth j push) gil::IRT_NQ -1)
                playing-j-one (gil::add-bool-var-expr sp (nth j playing) gil::IRT_EQ -1)
            )

            ;; playing[j] can only be equal to the preceding played note or a new pushed note
```

```
            ;; playing[j] = playing[j-1] || playing[j] = push[j]
            (gil::g-op sp playing-j-playing-j-one gil::BOT_OR push-j-playing-j 1)
            ;; push[j] can only equal the current note playing or -1
            ;; push[j] = playing[j] || push[j] = -1
            (gil::g-op sp push-j-playing-j gil::BOT_OR push-j-one 1)
            ;; A note can be pulled only if it was previously playing
            ;; pull[j] = playing[j-1] || pull[j] = -1
            (gil::g-op sp pull-j-playing-j-one gil::BOT_OR pull-j-one 1)
            ;; A note can be pushed only if the previous playing note was pulled
            ;; push[j] /= -1 => pull[j] = playing[j-1]
            (gil::g-op sp push-j-nq-one gil::BOT_IMP pull-j-playing-j-one 1)
            ;; No note playing implies no note pushed and previous note pulled
            ;; playing[j] = -1 => push[j] = -1 && pull[j] = playing[j-1]
            (gil::g-op sp playing-j-one gil::BOT_IMP push-j-one 1)
            (gil::g-op sp playing-j-one gil::BOT_IMP pull-j-playing-j-one 1)
            ;; Same note playing implies the note to either have been pushed and pulled
            ;; at the same time, or neither pushed or pulled
            ;; push[j] = pull[j] <=> playing[j] = playing[j-1]
            (gil::g-op sp playing-j-playing-j-one gil::BOT_IMP push-j-pull-j 1)
            (gil::g-op sp push-j-pull-j gil::BOT_IMP playing-j-playing-j-one 1)
        )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; Constrain Blocks and their sub-blocks ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Call the right function to constrain the block by their type
(defun constrain-srdc-from-parent (srdc-parent push pull playing push-acc pull-acc
↪  playing-acc push-A0 push-B0 quant max-pitch sp)
    (if (typep srdc-parent 'mldz::a)
        ;; The block is of type A, constrain it as such
        (constrain-srdc-from-A srdc-parent push pull playing push-acc pull-acc playing-acc
        ↪  push-A0 quant max-pitch sp)
        ;; The block is of type B, constrain it as such
        (constrain-srdc-from-B srdc-parent push pull playing push-acc pull-acc playing-acc
        ↪  push-B0 quant max-pitch sp)
    )
)


;; Split the three arrays for the sub-blocks then call the block-specific constraints
(defun constrain-srdc-from-AB (A-block push pull playing push-acc pull-acc playing-acc
↪  post-constraints quant max-pitch sp)
    (print "constrain-srdc-from-AB")
```

```

    (if (> (bar-length (s-block A-block)) 0)
        ;; bars*quant elements in each subblock and starts at startidx
        ;; for the sub arrays of push pull playing
        (let ((bars (bar-length (s-block A-block)))
            (s-block (s-block A-block))
            (r-block (r-block A-block))
            (d-block (d-block A-block))
            (c-block (c-block A-block))
            notes-in-subblock
            startidx-s startidx-r startidx-d startidx-c
            temp-push-s temp-pull-s temp-playing-s
            temp-push-s-acc temp-pull-s-acc temp-playing-s-acc
            temp-push-r temp-pull-r temp-playing-r
            temp-push-r-acc temp-pull-r-acc temp-playing-r-acc
            temp-push-d temp-pull-d temp-playing-d
            temp-push-d-acc temp-pull-d-acc temp-playing-d-acc
            temp-push-c temp-pull-c temp-playing-c
            temp-push-c-acc temp-pull-c-acc temp-playing-c-acc
            )


            ;; notes in each sub block (s/r/d/c)
            (setq notes-in-subblock (* bars quant))
            ;; sectioning the array into the respective parts for s r d c

            ;; access push pull playing arrays for the section related to s
            ;; (sublst x y z) creates a list based on list x from index y and of z
            ↪  sequential elements
            (setq startidx-s 0)
            (setq temp-push-s (sublst push startidx-s notes-in-subblock))
            (setq temp-pull-s (sublst pull startidx-s notes-in-subblock))
            (setq temp-playing-s (sublst playing startidx-s notes-in-subblock))
            (setq temp-push-s-acc (sublst push-acc startidx-s notes-in-subblock))
            (setq temp-pull-s-acc (sublst pull-acc startidx-s notes-in-subblock))
            (setq temp-playing-s-acc (sublst playing-acc startidx-s notes-in-subblock))

            ;; access push pull playing arrays for the section related to r
            (setq startidx-r notes-in-subblock)
            (setq temp-push-r (sublst push startidx-r notes-in-subblock))
            (setq temp-pull-r (sublst pull startidx-r notes-in-subblock))
            (setq temp-playing-r (sublst playing startidx-r notes-in-subblock))
            (setq temp-push-r-acc (sublst push-acc startidx-r notes-in-subblock))
            (setq temp-pull-r-acc (sublst pull-acc startidx-r notes-in-subblock))
            (setq temp-playing-r-acc (sublst playing-acc startidx-r notes-in-subblock))

```

```
            ;; access push pull playing arrays for the section related to d
            (setq startidx-d (+ startidx-r notes-in-subblock))
            (setq temp-push-d (sublst push startidx-d notes-in-subblock))
            (setq temp-pull-d (sublst pull startidx-d notes-in-subblock))
            (setq temp-playing-d (sublst playing startidx-d notes-in-subblock))
            (setq temp-push-d-acc (sublst push-acc startidx-d notes-in-subblock))
            (setq temp-pull-d-acc (sublst pull-acc startidx-d notes-in-subblock))
            (setq temp-playing-d-acc (sublst playing-acc startidx-d notes-in-subblock))
            (gil::g-rel sp (nth 0 temp-pull-d) gil::IRT_EQ (nth (- startidx-d 1) playing)) ; pull[0]=playing[previous]

            ;; access push pull playing arrays for the section related to c
            (setq startidx-c (+ startidx-d notes-in-subblock))
            (setq temp-push-c (sublst push startidx-c notes-in-subblock))
            (setq temp-pull-c (sublst pull startidx-c notes-in-subblock))
            (setq temp-playing-c (sublst playing startidx-c notes-in-subblock))
            (setq temp-push-c-acc (sublst push-acc startidx-c notes-in-subblock))
            (setq temp-pull-c-acc (sublst pull-acc startidx-c notes-in-subblock))
            (setq temp-playing-c-acc (sublst playing-acc startidx-c notes-in-subblock))
            (gil::g-rel sp (nth startidx-c pull) gil::IRT_EQ (nth (- startidx-c 1) playing)) ; pull[0]=playing[previous]

            ;; set constraints on these arrays from the values saved in the slots of s-block
            ;; s
            (print "constraining s")
            (constrain-s sp s-block A-block temp-push-s temp-pull-s temp-playing-s
                                            temp-push-s-acc temp-pull-s-acc temp-playing-s-acc
                                            max-pitch post-constraints)

            ;; r
            (print "constraining r")
            (constrain-r sp r-block A-block temp-push-r temp-pull-r temp-playing-r
                                            temp-push-r-acc temp-pull-r-acc temp-playing-r-acc
                                            temp-push-s temp-pull-s temp-playing-s
                                            max-pitch post-constraints)

            ;; d
            (print "constraining d")
            (constrain-d sp d-block A-block temp-push-d temp-pull-d temp-playing-d
                                            temp-push-d-acc temp-pull-d-acc temp-playing-d-acc
                                            temp-push-s temp-pull-s temp-playing-s
                                            max-pitch post-constraints)
```

```lisp

            ;; c
            (print "constraining c")
            (constrain-c sp c-block A-block temp-push-c temp-pull-c temp-playing-c
                                                temp-push-c-acc temp-pull-c-acc
                                                ↪  temp-playing-c-acc
                                                max-pitch post-constraints)

        )
    )

)

;; Constrain the A blocks with the resemblance if they are not the first A
(defun constrain-srdc-from-A (A-block push pull playing push-acc pull-acc playing-acc
↪  push-A0 quant max-pitch sp)
    (print "constrain-srdc-from-A")
    (let ((post-constraints t) (sim (similarity-percent-A0 A-block)))

    ;; If the block is not the first one of its type, the resemblance must be set with the
    ↪  first
    (if (not (= (block-position-A A-block) 0))
        (let (temp-push)
            (setq temp-push (transpose-chords-key sp (chord-key (nth (idx-first-a (parent
            ↪  A-block)) (block-list (parent A-block))))
                                                    (chord-quality (nth (idx-first-a (parent
                                                    ↪  A-block)) (block-list (parent
                                                    ↪  A-block))))
                                                    (chord-key A-block) (chord-quality A-block)
                                                    ↪  push-A0))
            (cst-common-vars sp temp-push push sim)
            ;; if it has 100% resemblance with the first A, posting constraints on melody
            ↪  might create conflicts
            (if (= sim 100)
                (setq post-constraints nil)
                (setq post-constraints t)
            )
        )
    )

    ;; A and B behave the same way, the only distinction is done
    ;; with the resemblance beween blocks of the same type
    ;; so the same function can be called for the sub-blocks
    (constrain-srdc-from-AB A-block push pull playing push-acc pull-acc playing-acc
    ↪  post-constraints quant max-pitch sp)
    )
```

```lisp
)

;; Constrain the B blocks with the resemblance if they are not the first B
(defun constrain-srdc-from-B (B-block push pull playing push-acc pull-acc playing-acc push-B0 quant max-pitch sp)
    (print "constrain-srdc-from-B")
    (let ((post-constraints t) (sim (similarity-percent-B0 B-block)))

    (if (not (= (block-position-B B-block) 0))
        (let (temp-push)
            (setq temp-push (transpose-chords-key sp (chord-key (nth (idx-first-b (parent B-block)) (block-list (parent B-block))))
                                                    (chord-quality (nth (idx-first-b (parent B-block)) (block-list (parent B-block))))
                                                    (chord-key B-block) (chord-quality B-block) push-B0))
            (cst-common-vars sp temp-push push sim)
            ;; if it has 100% resemblance with the first A, posting constraints on melody might create conflicts
            (if (= sim 100)
                (setq post-constraints nil)
                (setq post-constraints t)
            )
        )
    )

    ;; A and B behave the same way, the only distinction is done
    ;; with the resemblance beween blocks of the same type
    ;; so the same function can be called for the sub-blocks
    (constrain-srdc-from-AB B-block push pull playing push-acc pull-acc playing-acc post-constraints quant max-pitch sp)
    )
)

;; for now these constrain-srdc functions take the parent block as argument in case it comes in handy
;; when we implement more constraints which could be specified through slots of the parent block
(defun constrain-s (sp s-block s-parent push pull playing push-acc pull-acc playing-acc max-pitch post-constraints)

    ;; if  (/= melody-source nil) and (block-position-A == 0)
    (let ((melody-A (melody-source-A (parent s-parent)))
        (melody-B (melody-source-B (parent s-parent)))
        (first-A (= (block-position-A s-parent) 0))
```

```
        (first-B (= (block-position-B s-parent) 0))
        set-A set-B
        )
        (setq set-A (and first-A melody-A))
        (setq set-B (and first-B melody-B))

        (if (or set-A set-B)
            ;; if in a block that needs to have it's melody set to a source
            (if set-A
                ;; set-A
                (let (push-source pull-source playing-source ppp-source)
                    (setq ppp-source (create-push-pull-int (melody-source-A (parent s-parent)) 16))

                    (setq push-source (first ppp-source))
                    (setq pull-source (second ppp-source))
                    (setq playing-source (third ppp-source))

                    (loop :for i :from 0 :below (length push-source) :by 1 :do
                        (gil::g-rel sp (nth i push) gil::IRT_EQ (nth i push-source))
                    )
                    (loop :for i :from 1 :below (- (length pull-source) 1) :by 1 :do
                        (gil::g-rel sp (nth i pull) gil::IRT_EQ (nth i pull-source))
                    )
                    (loop :for i :from 0 :below (length playing-source) :by 1 :do
                        (gil::g-rel sp (nth i playing) gil::IRT_EQ (nth i playing-source))
                    )

                    (print "First A block's s has been set to the source melody")
                    (if (< (length push-source) (length push))
                        (post-rock-constraints sp s-block   (sublst push (length push-source) (- (length push) (length push-source)))
                                                            (sublst pull (length push-source) (- (length push) (length push-source)))
                                                            (sublst playing (length push-source) (- (length push) (length push-source)))
                                                            nil t)
                    )
                )
                ;; set-B
                (let (push-source pull-source playing-source ppp-source)
                    (setq ppp-source (create-push-pull-int (melody-source-B (parent s-parent)) 16))
```

```

                    (setq push-source (first ppp-source))
                    (setq pull-source (second ppp-source))
                    (setq playing-source (third ppp-source))

                    (loop :for i :from 0 :below (length push-source) :by 1 :do
                        (gil::g-rel sp (nth i push) gil::IRT_EQ (nth i push-source))
                    )
                    (loop :for i :from 1 :below (- (length pull-source) 1) :by 1 :do
                        (gil::g-rel sp (nth i pull) gil::IRT_EQ (nth i pull-source))
                    )
                    (loop :for i :from 0 :below (length playing-source) :by 1 :do
                        (gil::g-rel sp (nth i playing) gil::IRT_EQ (nth i playing-source))
                    )
                    (if (< (length push-source) (length push))
                        (post-rock-constraints sp s-block   (sublst push (length push-source) (- (length push) (length push-source)))
                                                            (sublst pull (length push-source) (- (length push) (length push-source)))
                                                            (sublst playing (length push-source) (- (length push) (length push-source)))
                                                            nil t)
                    )
                    (print "First B block's s has been set to the source melody")
                )
            )
            ;; neither set-A nor set-B =>
            ;; don't need to set a source melody, constrain as it should normally do
            (post-rock-constraints sp s-block push pull playing nil post-constraints)
        )
        ;; ;; accompaniment should always be constrained
        (post-rock-constraints sp (accomp s-block) push-acc pull-acc playing-acc nil t)
    )

)


;; Constrain the r block based on its resemblance with the s-block
(defun constrain-r (sp r-block r-parent push pull playing push-acc pull-acc playing-acc
                                        push-s pull-s playing-s max-pitch post-constraints)

```

```
    (gil::g-rel sp (first pull) gil::IRT_EQ (nth (- (length playing-s) 1) playing-s)) ; pull[0]=playing-s[quant-1]

    ;; post optional constraints defined in the rock csp
    ;; dont constrain if source melody is given or the similarity with the s block is 100%
    (let (melody)
        (if (typep r-parent 'mldz::a)
            (setq melody (melody-source-A (parent r-parent)))
            (setq melody (melody-source-B (parent r-parent)))
        )
        (post-rock-constraints sp r-block push pull playing nil (and post-constraints (or (not melody) (< (similarity-percent-s r-block) 100))))
    )


    (post-rock-constraints sp (accomp r-block) push-acc pull-acc playing-acc nil t)

    ;; constrain r such that it has a similarity of (similarity-percent-s r-block) with notes played in s-block
    ;; transposed the number of semitones asked of the r-block
    (let ((sim (similarity-percent-s r-block))
            temp-push  temp-playing
        )
        (setq temp-push (transpose-chords-semitones sp (chord-key (s-block r-parent)) (chord-quality (s-block r-parent))
                                            (semitones r-block) push-s))
        (cst-common-vars sp temp-push push sim)
    )
)

; Constrain the d-block based on its resemblance with the s-bloc
(defun constrain-d (sp d-block d-parent push pull playing push-acc pull-acc playing-acc
                                        push-s pull-s playing-s max-pitch post-constraints)
    (post-rock-constraints sp d-block push pull playing nil post-constraints)
    (post-rock-constraints sp (accomp d-block) push-acc pull-acc playing-acc nil t)

     ;; constrain d such that it has a difference of (difference-percent-s d-block) with notes played in s-block
    ;; transposed the number of semitones asked of the d-block
    (let ((diff (difference-percent-s d-block))
            temp-push  temp-playing
        )
        (setq temp-push (transpose-chords-semitones sp (chord-key (s-block d-parent)) (chord-quality (s-block d-parent))
                                            (semitones d-block) push-s))
```

```
        (cst-common-vars sp temp-push push (- 100 diff))
    )
)

;; constrain c such that is respects the cadence specific rules
(defun constrain-c (sp c-block c-parent push pull playing push-acc pull-acc playing-acc max-pitch post-constraints)

    (let ((block-list-len (length (block-list (parent c-parent)))) ;; how many blocks are in the global structure
        (position (block-position c-parent)) ;; position of the current block in the global structure (start index is 0)
        (c-type (cadence-type c-block))
        (key (chord-key c-block))
        (quality (chord-quality c-block))
        (chord-midi-value (name-to-note-value (chord-key c-block)))
        (triad-to-play (list)) ;; intervals depending on quality
        (chords-to-play (list)) ;; root key(s) on which the triad(s) is(are) played
        (notes-to-play (list)) ;; notes to be pushed, list of lists
        (mnl (min-note-length (accomp c-block)))
        )
        (cond ((string= quality "Major") (setq triad-to-play (list 0 4 7)))
            ((string= quality "Minor") (setq triad-to-play (list 0 3 7)))
            ((string= quality "Augmented") (setq triad-to-play (list 0 4 8)))
            ((string= quality "Diminished") (setq triad-to-play (list 0 3 6)))
        )
        (cond
            ((string= c-type "None")
                (print "cadence-type")
                (print "No cadence")
                ;; TODO: Check if None functions properly
            )
            ((string= c-type "Perfect")
                (print "cadence-type")
                (print "Perfect")

                ;; Perfect V -> I
                (setq chords-to-play (list 7 0))
                (setq notes-to-play (append notes-to-play (list (+ (+ chord-midi-value (nth 0 chords-to-play)) (nth 0 triad-to-play)) (+ (+ chord-midi-value (nth 0 chords-to-play)) (nth 1 triad-to-play)) (+ (+ chord-midi-value (nth 0 chords-to-play)) (nth 2 triad-to-play)))))

                (setq notes-to-play (append (list notes-to-play) (list (list (+ (+ chord-midi-value (nth 1 chords-to-play)) (nth 0 triad-to-play)) (+ (+ chord-midi-value (nth 1 chords-to-play)) (nth 1 triad-to-play)) (+ (+ chord-midi-value (nth 1 chords-to-play)) (nth 2 triad-to-play))))))
```

```

            (gil::g-rel sp (nth 0 push-acc) gil::SRT_EQ (nth 0 notes-to-play))
            (gil::g-rel sp (nth (* (/ mnl 2) (bar-length (accomp c-block))) push-acc)
            ↪   gil::SRT_EQ (nth 1 notes-to-play))
        )
        ((string= c-type "Plagal")
            (print "cadence-type")
            (print "Plagal")

            ;; Plagal IV -> I
            (setq chords-to-play (list 5 0))
            (setq notes-to-play (append notes-to-play (list (+ (+ chord-midi-value
            ↪   (nth 0 chords-to-play)) (nth 0 triad-to-play)) (+ (+ chord-midi-value
            ↪   (nth 0 chords-to-play)) (nth 1 triad-to-play)) (+ (+ chord-midi-value
            ↪   (nth 0 chords-to-play)) (nth 2 triad-to-play)))))

            (setq notes-to-play (append (list notes-to-play) (list (list (+ (+
            ↪   chord-midi-value (nth 1 chords-to-play)) (nth 0 triad-to-play)) (+ (+
            ↪   chord-midi-value (nth 1 chords-to-play)) (nth 1 triad-to-play)) (+ (+
            ↪   chord-midi-value (nth 1 chords-to-play)) (nth 2 triad-to-play))))))

            (gil::g-rel sp (nth 0 push-acc) gil::SRT_EQ (nth 0 notes-to-play))
            (gil::g-rel sp (nth (* (/ mnl 2) (bar-length (accomp c-block))) push-acc)
            ↪   gil::SRT_EQ (nth 1 notes-to-play))
        )
        ((string= c-type "Semi")
            (print "cadence-type")
            (print "Semi")

            ;; Demi I -> V
            (setq chords-to-play (list 0 7))
            (setq notes-to-play (append notes-to-play (list (+ (+ chord-midi-value
            ↪   (nth 0 chords-to-play)) (nth 0 triad-to-play)) (+ (+ chord-midi-value
            ↪   (nth 0 chords-to-play)) (nth 1 triad-to-play)) (+ (+ chord-midi-value
            ↪   (nth 0 chords-to-play)) (nth 2 triad-to-play)))))

            (setq notes-to-play (append (list notes-to-play) (list (list (+ (+
            ↪   chord-midi-value (nth 1 chords-to-play)) (nth 0 triad-to-play)) (+ (+
            ↪   chord-midi-value (nth 1 chords-to-play)) (nth 1 triad-to-play)) (+ (+
            ↪   chord-midi-value (nth 1 chords-to-play)) (nth 2 triad-to-play))))))

            (gil::g-rel sp (nth 0 push-acc) gil::SRT_EQ (nth 0 notes-to-play))
            (gil::g-rel sp (nth (* (/ mnl 2) (bar-length (accomp c-block))) push-acc)
            ↪   gil::SRT_EQ (nth 1 notes-to-play))
        )
        ((string= c-type "Deceptive")
```

```
                (print "cadence-type")
                (print "Deceptive")
                ;; Deceptive V -> VI || V -> III
            )
        )
    )

    (let ((bar-len (bar-length c-block))
        (quant 16)
        (chord-midi-value (name-to-note-value (chord-key c-block)))
        notes
        final-idx
        )
        (setq notes (octaves-of-note chord-midi-value))
        (setq final-idx (- (* bar-len quant) 1))
        (gil::g-dom sp (nth final-idx playing) notes)
    )
    (post-rock-constraints sp c-block push pull playing t post-constraints)

    (post-rock-constraints sp (accomp c-block) push-acc pull-acc playing-acc t t)
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; LIMITING NOTE TO THE SCALE ;;
;; OR THE CHORDS              ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;; Constraints on polyphonic voices
(defun chord-key-cst (sp playing rock)
    (let ((key (chord-key rock))
        (quality (chord-quality rock))
        (chord-midi-value (name-to-note-value (chord-key rock)))
        (triad-to-play (list)) ;; intervals depending on quality
        (notes-to-play (list))
        )
        (cond ((string= quality "Major") (setq triad-to-play (list 0 4 3)))
            ((string= quality "Minor") (setq triad-to-play (list 0 3 4)))
            ((string= quality "Diminished") (setq triad-to-play (list 0 3 3)))
            ((string= quality "Augmented") (setq triad-to-play (list 0 4 4)))
        )
        (setq notes-to-play (build-chordset triad-to-play (- chord-midi-value 60)))
        (loop :for i :from 0 :below (length playing) :do
            (let ((bool-array (gil::add-bool-var-array sp (length notes-to-play) 0
            ↪  1)));;Array to state that one triad is played
                (loop :for j :from 0 :below (length notes-to-play) :do
                    (gil::g-rel-reify sp (nth i playing) gil::SRT_EQ (nth j notes-to-play)
                    ↪  (nth j bool-array) gil::RM_IMP)
```

```
                )
                ;; Exactly one triad can be played at each time
                (gil::g-rel sp gil::BOT_OR bool-array 1)
            )
        )
    )
)

;; Constraints on monophonic voices
(defun chord-key-cst-int (sp push playing rock)
    (let (
        (chord (get-scale-chord (chord-quality rock)))
        (offset (- (name-to-note-value (chord-key rock)) 60))
        chordset
        )
        (setq chordset (build-scaleset chord offset))
        (loop :for i :from 0 :below (length playing) :by 1 :do
            (let (bool-array bool-temp)
                (setq bool-array (gil::add-bool-var-array sp (+ (length chordset) 1) 0 1))
                (loop :for n :from 0 :below (length chordset) :by 1 :do
                    (let (bool)
                        (setq bool (gil::add-bool-var-expr sp (nth i playing) gil::IRT_EQ
                        ↪  (nth n chordset)))
                        (gil::g-rel sp bool gil::IRT_EQ (nth n bool-array))
                    )
                )
                (setq bool-temp (gil::add-bool-var-expr sp (nth i playing) gil::IRT_EQ
                ↪  -1))
                (gil::g-rel sp bool-temp gil::IRT_EQ (nth (length chordset) bool-array))
                (gil::g-rel sp gil::BOT_OR bool-array 1)
            )
        )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;
; LIMITING PITCH RANGE ;
;;;;;;;;;;;;;;;;;;;;;;;;;

(defun pitch-range (sp push min-pitch max-pitch)
    (loop :for j :below (length push) :by 1 :do
        (if (typep (nth j push) 'gil::int-var)
            ;; Constraints on monophonic voices
            (progn
                (let (bool-temp bool-one bool-min bool-max)
                    (setq bool-one (gil::add-bool-var-expr sp (nth j push) gil::IRT_EQ
                    ↪  -1))
```

```lisp
                    (setq bool-min (gil::add-bool-var-expr sp (nth j push) gil::IRT_GQ min-pitch))
                    (setq bool-max (gil::add-bool-var-expr sp (nth j push) gil::IRT_LQ max-pitch))
                    (setq bool-temp (gil::add-bool-var sp 0 1))
                    (gil::g-op sp bool-min gil::BOT_AND bool-max bool-temp)
                    (gil::g-op sp bool-temp gil::BOT_OR bool-one 1)
                )
            )
            ;; Constraints on polyphonic voices
            (gil::g-dom-ints sp (nth j push) gil::SRT_SUB min-pitch max-pitch)
        )
    )
)


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; LIMITING MINIMUM NOTE LENGTH ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun note-min-length-rock (sp push pull playing min-length)
    (loop :for j :from 0 :below (length push) :by 1 :do
        (loop :for k :from 1 :below min-length :by 1 :while (< (+ j k) (length pull)) :do
            (if (typep (nth j push) 'gil::int-var)
                ;; Constraints on monophonic voices
                (let (bool-temp bool2 bool3 bool4 bool5 bool6)
                    (setq bool-temp (gil::add-bool-var-expr sp (nth j push) gil::IRT_NQ -1))
                    (setq bool3 (gil::add-bool-var-expr sp (nth (+ j k) pull) gil::IRT_EQ -1))
                    (gil::g-op sp bool-temp gil::BOT_IMP bool3 1)

                    ;; Limiting silence minimum length
                    (if (> j 0)
                        (progn
                            (setq bool2 (gil::add-bool-var-expr sp (nth j playing) gil::IRT_EQ -1))
                            (setq bool5 (gil::add-bool-var-expr sp (nth (- j 1) playing) gil::IRT_NQ -1))
                            (setq bool4 (gil::add-bool-var-expr sp (nth (+ j k) playing) gil::IRT_EQ -1))
                            (setq bool6 (gil::add-bool-var sp 0 1))
                            (gil::g-op sp bool5 gil::BOT_AND bool2 bool6)
                            (gil::g-op sp bool6 gil::BOT_IMP bool4 1)
                        )
                        (progn
```

```
                            (setq bool2 (gil::add-bool-var-expr sp (nth j playing)
                            ↪  gil::IRT_EQ -1))
                            (setq bool4 (gil::add-bool-var-expr sp (nth (+ j k) playing)
                            ↪  gil::IRT_EQ -1))
                            (gil::g-op sp bool2 gil::BOT_IMP bool4 1)
                        )
                    )

                )
                ;; Constraints on polyphonic voices
                (gil::g-rel sp (nth (+ j k) pull) gil::SRT_DISJ (nth j push))
            )
        )
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; LIMITING MAXIMUM NOTE LENGTH ;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun note-max-length-rock (sp push pull max-length)
    (setq l max-length)
    (if (typep (nth 0 push) 'gil::int-var)
        ;; Constraints on monophonic voices
        (loop :for j :from 0 :below (- (length push) l) :by 1 :do
            (let (  (count (gil::add-int-var sp 0 l))
                    (int-array (gil::add-int-var-array sp l 0 l)))
                (loop :for k :from 0 :below l :by 1 :do
                    (setf (nth k int-array) (gil::add-int-var-expr sp (nth j push)
                    ↪  gil::IOP_SUB (nth (+ 1 (+ j k)) pull)))
                )
                (gil::g-count sp int-array 0 gil::IRT_EQ count)
                (gil::g-rel sp count gil::IRT_GQ 1)
            )
        )
        ;; Constraints on polyphonic voices
        (loop :for j :from 0 :below (- (length push) l) :by 1 :do
            (let ((l-pull (gil::add-set-var-array sp l 0 127 0 127))
                (l-pull-union (gil::add-set-var sp 0 127 0 127)))
                (loop :for k :from 0 :below l :by 1 :do
                    (gil::g-rel sp (nth k l-pull) gil::SRT_EQ (nth (+ 1 (+ j k)) pull))
                )
                (gil::g-setunion sp l-pull-union l-pull)
                (gil::g-rel sp (nth j push) gil::SRT_SUB l-pull-union)
            )
        )
```

```
    )

)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; LIMITING THE NUMBER OF COMMON NOTES ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun cst-common-vars (sp vars1 vars2 sim)
    (let (count-vars int-array n-vars perc)
        (setq perc (/ sim 100))
        (setq n-vars (ceiling (* (length vars1) perc)))

        (setq count (gil::add-int-var sp 0 (length vars1)))
        (setq int-array (gil::add-int-var-array sp (length vars1) -127 127))

        (loop :for i :from 0 :below (min (length vars1) (length vars2)) do
            (setf (nth i int-array) (gil::add-int-var-expr sp (nth i vars1) gil::IOP_SUB
            ↪  (nth i vars2)))
        )

        (gil::g-count sp int-array 0 gil::IRT_EQ count)
        (gil::g-rel sp count gil::IRT_GQ n-vars)
    )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; LIMITING THE INTERVALS BETWEEN NOTES ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun limit-intervals-cst (sp playing)
    (let ((max-interval 7))
        (loop :for i :from 1 :below (length playing) :do
            (limit-one-interval-cst sp (nth i playing) (nth (- i 1) playing) max-interval)
        )
    )
)

(defun limit-one-interval-cst (sp playing-i playing-i-one max-interval)
    (let (bool-interval-max interval interval-abs bool-pi bool-pi-one bool)
            (setq bool-pi (gil::add-bool-var-expr sp playing-i gil::IRT_EQ -1))
            (setq bool-pi-one (gil::add-bool-var-expr sp playing-i-one gil::IRT_EQ -1))

            ;; Define the interval between the two notes
            ;; interval = |playing[i] - playing[i-1]|
            (setq interval (gil::add-int-var-expr sp playing-i gil::IOP_SUB
            ↪  playing-i-one))
```

```
            (setq interval-abs (gil::add-int-var sp 0 127))
            (gil::g-abs sp interval interval-abs)

            ;; The maximum interval
            ;; interval <= 7 (perfect fifth)
            (setq bool-interval-max (gil::add-bool-var-expr sp interval-abs gil::IRT_LQ max-interval))

            ;; playing[i] = -1 OR |interval| <= max-interval
            (setq bool (gil::add-bool-var sp 0 1))
            (gil::g-op sp bool-pi gil::BOT_OR bool-pi-one bool)
            (gil::g-op sp bool gil::BOT_OR bool-interval-max 1)
        )
)

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;; TRANSPOSING AN ARRAY OF VARIABLE ;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

(defun transpose-chords-key (sp chord1 quality1 chord2 quality2 push)
    (let (
        (notes (build-scaleset (get-scale-chord quality1)
                    (- (name-to-note-value chord1) 60)))
        (new-notes (build-scaleset (get-scale-chord quality2)
                    (- (name-to-note-value chord2) 60)))
        temp-push
        )
        (setq notes (append '(-1) notes))
        (setq new-notes (append '(-1) new-notes))
        (setq temp-push (gil::add-int-var-array sp (length push) -1 127))
        (loop :for i :from 0 :below (length push) :do
            (let ((bool-array (gil::add-bool-var-array sp (length notes) 0 1)) bool-temp bool-tot difference)
                (loop :for n :from 0 :below (min (length notes) (length new-notes)) :do
                    (let (bool1 bool2)
                        ;; If the note belongs to the chord, force the new note to belong to the new chord
                        (setq bool1 (gil::add-bool-var-expr sp (nth i push) gil::IRT_EQ (nth n notes)))
                        (setq bool2 (gil::add-bool-var-expr sp (nth i temp-push) gil::IRT_EQ (nth n new-notes)))
                        (gil::g-op sp bool1 gil::BOT_IMP bool2 1)
                    )
                )
            )
        )
```

```
        temp-push
    )
)


(defun transpose-chords-semitones (sp chord1 quality1 semitones push)
    (let (
        (notes (build-scaleset (get-scale-chord quality1)  ;if - mode selectionné
                    (- (name-to-note-value chord1) 60)))
        temp-push  new-notes
        )
        (setq new-notes (loop :for i :from 0 :below (length notes) :collect (+ (nth i notes) semitones)))
        (setq notes (append '(-1) notes))
        (setq new-notes (append '(-1) new-notes))
        (setq temp-push (gil::add-int-var-array sp (length push) -1 127))
        (loop :for i :from 0 :below (length push) :do
            (let ((bool-array (gil::add-bool-var-array sp (length notes) 0 1)) bool-temp bool-tot difference)
                (loop :for n :from 0 :below (min (length notes) (length new-notes)) :do
                    (let (bool1 bool2)
                        ;; If the note belongs to the chord, force the new note to belong to the new chord
                        (setq bool1 (gil::add-bool-var-expr sp (nth i push) gil::IRT_EQ (nth n notes)))
                        (setq bool2 (gil::add-bool-var-expr sp (nth i temp-push) gil::IRT_EQ (nth n new-notes)))
                        (gil::g-op sp bool1 gil::BOT_IMP bool2 1)
                    )
                )
            )
        )
        temp-push
    )
)
```

D.4 Utilities Functions

Two files define utilities function that were used throughout Melodizer Rock other files:

- **rock-utils.csp** contains the functions specifically added for Melodizer Rock
- **melodizer-utils.lisp** contains functions from the previous works of Melodizer 1.0 [2] and Melodizer 2.0 [3]. As some were used for Melodizer Rock, it is recalled here.

D.4.1 sources/rock-utils.lisp

This file contains useful functions such as one for calculating the length of a tree, or to propagate the attribute values through the blocks ...

```lisp
(in-package :mldz)

;; Function to change the values of a sub-block according to the new value
;; of the parent block and the differences calculated before
(defun change-subblocks-values (rock-block &key bar-length
                                                chord-key
                                                min-pitch
                                                max-pitch
                                                min-note-length-flag
                                                min-note-length
                                                max-note-length-flag
                                                max-note-length
                                                min-simultaneous-notes
                                                max-simultaneous-notes
                                                chord-quality
                                                semitones)
    (let (block-list)

    ;; Setup the sub-block list for the loop
    (cond
        ((typep rock-block 'mldz::rock) (setq block-list (block-list rock-block)))
        ((or (typep rock-block 'mldz::a) (typep rock-block 'mldz::b))
            (setq block-list (list   (s-block rock-block)
                                    (r-block rock-block)
                                    (d-block rock-block)
                                    (c-block rock-block)))
        )
        ((or (typep rock-block 'mldz::s) (typep rock-block 'mldz::r)
            (typep rock-block 'mldz::d) (typep rock-block 'mldz::c))
            (setq block-list (list (accomp rock-block)))
        )
    )

    ;; Update the diff parameter for this block
    (if (not (typep rock-block 'mldz::rock))
        (progn

            ;; Pitch constraints
            (if (and chord-key (chord-key (parent rock-block)))
                (setf   (diff-chord-key rock-block)
                        (-  (name-to-note-value (chord-key (parent rock-block)))
                            (name-to-note-value chord-key)))
```

```
            )
            (if (and min-pitch (min-pitch (parent rock-block)))
                (setf   (diff-min-pitch rock-block)
                        (- (min-pitch (parent rock-block))
                            min-pitch))
            )
            (if (and max-pitch (max-pitch (parent rock-block)))
                (setf   (diff-max-pitch rock-block)
                        (- (max-pitch (parent rock-block))
                            max-pitch))

            )

            ;;Other constraints
            (if (not (typep rock-block 'mldz::accompaniment))
                (progn
                    (if (and (or min-note-length-flag min-note-length) (min-note-length
                    ↪  (parent rock-block)))
                        (setf   (diff-min-length rock-block)
                                (-  (log (min-note-length (parent rock-block)) 2)
                                    (log min-note-length 2)))

                    )
                    (if (and (or max-note-length-flag max-note-length) (max-note-length
                    ↪  (parent rock-block)))
                        (setf   (diff-max-length rock-block)
                                (-  (log (max-note-length (parent rock-block)) 2)
                                    (log max-note-length 2)))
                    )
                )
            )

        )
    )

    ;; Loop on sub-blocks to update their values
    (loop :for x in block-list do
        (setf (parent x) rock-block)
        (if bar-length
            (progn
                (setq n-bars (/ bar-length (list-length block-list)))
                (setf (bar-length x) n-bars)
            )
        )

        ;;Pitch constraints
```

```
        (if chord-key
            (cond
                ((relative-to-parent x)
                    (setf (chord-key x) (note-value-to-name (- (name-to-note-value chord-key) (diff-chord-key x))))
                )
            )
        )
        (if min-pitch
            (cond
                ((relative-to-parent x)
                    (setf (min-pitch x) (- min-pitch (diff-min-pitch x)))
                )
            )

        )
        (if max-pitch
            (cond
                ((relative-to-parent x)
                    (setf (max-pitch x) (- max-pitch (diff-max-pitch x)))
                )
            )
        )
        (if chord-quality
            (setf (chord-quality x) chord-quality)
        )

        ;; Other constraints
        (if (not (typep x 'mldz::accompaniment))
            (progn
                (if min-note-length
                    (cond
                        ((relative-to-parent x)
                        (progn
                            (setf   (min-note-length-flag x) min-note-length-flag
                                    (min-note-length x) (floor (expt 2 (- (log min-note-length 2) (diff-min-length x)))))
                        )
                        )
                    )
                )
                (if max-note-length
                    (cond
                        ((relative-to-parent x)
                            (setf   (max-note-length-flag x) max-note-length-flag
                                    (max-note-length x) (floor (expt 2 (- (log max-note-length 2) (diff-max-length x))))))
```

```
                    )
                )
                (if semitones
                    (setf (semitones x) semitones)
                )
            )
        )


        (change-subblocks-values x  :bar-length (bar-length x)
                                    :chord-key (chord-key x)
                                    :min-pitch (min-pitch x)
                                    :max-pitch (max-pitch x)
                                    :min-note-length-flag (min-note-length-flag x)
                                    :min-note-length (min-note-length x)
                                    :max-note-length-flag (max-note-length-flag x)
                                    :max-note-length  (max-note-length x)
                                    :min-simultaneous-notes min-simultaneous-notes
                                    :max-simultaneous-notes max-simultaneous-notes
                                    :chord-quality (chord-quality x)
                                    :semitones semitones
        )

    )

    )
)

;; Function that returns a list corresponding to the values the
;; bar-length parameter of a block can take
(defun bar-length-range (rock-block)
    (if (or (typep rock-block 'mldz::s)
            (typep rock-block 'mldz::r)
            (typep rock-block 'mldz::d)
            (typep rock-block 'mldz::c))
        (loop   :for n
                :from 0
                :below 5
                :by 1
                :collect (number-to-string n))
        ;; When it  is rock block, it must have a number of bar
        ;; divisable between all the blocks A and B and their s r d c sub-blocks
        ;; thus 4 per element of its block-list
        (let ((sum (bar-length rock-block))(result (list)))
            (if (typep rock-block 'mldz::rock)
                (if (= sum 0)
```

```
                    (if (block-list rock-block)
                        (progn
                            (setq n-block (list-length (block-list rock-block)))
                            (setq result (append '("0") (loop :for n
                                                    :from (* 4 n-block)
                                                    :below (+ (* 16 n-block) 1)
                                                    :by (* 4 n-block)
                                                    :collect (number-to-string n))))
                        )
                        (setf result '("0"))
                    )
                    (setq result (list (number-to-string sum)))
                )
            )
            ;; When it is a block A or B, it must be a multiple of 4
            (if (or (typep rock-block 'mldz::a) (typep rock-block 'mldz::b))
                (if (= sum 0)
                    (setq result (append (loop :for n
                                            :from 0
                                            :below 17
                                            :by 4
                                            :collect (number-to-string n))))

                    (setq result (list (number-to-string sum)))
                )
            )
            result
        )
    )
)

;; Compute the bar-length of a rock block based
;; on the bar-length of its sub-blocks
(defun bar-length-sum-rock (rock)
    (let ((sum 0))
        (loop :for n :from 0 :below (list-length (block-list rock)) :by 1
        do
            (setq sum (+ sum (bar-length (nth n (block-list rock)))))
        )
        sum
    )
)

;; Compute the bar-length of a A or B block based
;; on the bar-length of its sub-blocks
(defun bar-length-sum-AB (A)
```

```
    (+  (bar-length (s-block A))
        (bar-length (r-block A))
        (bar-length (d-block A))
        (bar-length (c-block A)))
)

;;; When the bar-length of a sub-block is changed,
;; the bar-length of the parents is adapted
(defun set-bar-length-up (rock-block)
    (if (or (typep (parent rock-block) 'mldz::a) (typep (parent rock-block) 'mldz::b))
        (setf (bar-length (parent rock-block)) (bar-length-sum-AB (parent rock-block)))
        (setf (bar-length (parent rock-block)) (bar-length-sum-rock (parent rock-block)))
    )
    ;; (make-my-interface (parent rock-block))
    (if (not (typep (parent rock-block) 'mldz::rock))
        (set-bar-length-up (parent rock-block))
    )
)

;; Round up to the next exponent of 2
(defun ceil-to-exp (val)
    (cond
        ((<= val 1) 1)
        ((<= val 2) 2)
        ((<= val 4) 4)
        ((<= val 8) 8)
        ((<= val 16) 16)
    )
)

;; Compute the total length of a tree
(defun get-length-tree (tree)
    (let ((length 0))
        (loop :for i :from 0 :below (length tree) :do
            (if (typep (nth i tree) 'list)
                (setq length (+ length (first (nth i tree))))
                (setq length (+ length (abs (nth i tree))))
            )
        )
        length
    )
)

;; When bar-length of a s r d or c is changed, the other block
;; with the same parents get the same bar length
(defun propagate-bar-length-srdc (rock-block)
```

```
    (let ((parent (parent rock-block)) (nbars (bar-length rock-block)))
        (if (or (typep parent 'mldz::a) (typep parent 'mldz::b))
            (progn
                (setf (bar-length (s-block parent)) nbars)
                (setf (bar-length (r-block parent)) nbars)
                (setf (bar-length (d-block parent)) nbars)
                (setf (bar-length (c-block parent)) nbars)
            )
        )
    )
)

;; http://www.lee-mac.com/sublist.html
;; Sublst  -  Lee Mac
;; The list analog of the substr function
;; lst - [lst] List from which sublist is to be returned
;; idx - [int] Zero-based index at which to start the sublist
;; len - [int] Length of the sublist or nil to return all items following idx
(defun sublst (lst idx len)
    (cond
        (   (null lst) nil)
        (   (< 0  idx) (sublst (cdr lst) (1- idx) len))
        (   (null len) lst)
        (   (< 0  len) (cons (car lst) (sublst (cdr lst) idx (1- len))))
    )
)

;; Count the number of blocks of type A in block-list
(defun count-A-block-list (block-list)
    (let ((count 0))
        (dolist (n block-list)
            (if (typep n 'mldz::a)
                (setq count (+ count 1))
            )
        )
        count
    )
)

;; Count the number of blocks of type B in block-list
(defun count-B-block-list (block-list)
    (let ((count 0))
        (dolist (n block-list)
            (if (typep n 'mldz::b)
                (setq count (+ count 1))
            )
```

```
        )
        count
    )
)


;; each diff argument is the difference between the old diff and new diff of the changed block A or B
;; For example, if a block A goes from diff-max-pitch 5 to diff-max-pitch 3, the argument diff-max-pitch is 2
(defun propagate-AB (AB-block &key  diff-min-sim
                                    diff-max-sim
                                    diff-min-length
                                    diff-max-length
                                    diff-chord-key
                                    diff-chord-quality
                                    diff-min-pitch
                                    diff-max-pitch)
    (let (
        (parent (parent AB-block))
        (type-block (type-of AB-block))
        block-list
        )
        (setf block-list (block-list parent))
        ;; For each block of the same type in block-list
        ;; If they are relative, change their value according to the difference
        (loop :for x in block-list do
            (if (and (not (eq x AB-block)) (relative-to-same x) (typep x type-block))
                (progn
                    (if diff-min-sim
                        (progn
                            (setf   (diff-min-sim x) (- (diff-min-sim x) diff-min-sim))
                            (setf   (min-simultaneous-notes x) (- (min-simultaneous-notes parent) (diff-min-sim x)))
                            (change-subblocks-values x
                                  :min-simultaneous-notes (min-simultaneous-notes x))
                        )
                    )
                    (if diff-max-sim
                        (progn
                            (setf   (diff-max-sim x) (- (diff-max-sim x) diff-max-sim))
                            (setf   (max-simultaneous-notes x) (- (max-simultaneous-notes parent) (diff-max-sim x)))
                            (change-subblocks-values x
                                  :max-simultaneous-notes (max-simultaneous-notes x))
                        )
```

```
                    )
                    (if diff-min-length
                        (progn
                            (setf   (diff-min-length x) (- (diff-min-length x)
                            ↪  diff-min-length))
                            (setf   (min-note-length x) (floor (expt 2 (- (log
                            ↪  (min-note-length parent) 2) (diff-min-length x)))))
                            (change-subblocks-values x
                                    :min-note-length-flag (min-note-length-flag x)
                                    :min-note-length (min-note-length x))
                        )
                    )
                    (if diff-max-length
                        (progn
                            (setf   (diff-max-length x) (- (diff-max-length x)
                            ↪  diff-max-length))
                            (setf   (max-note-length x) (floor (expt 2 (- (log
                            ↪  (max-note-length parent) 2) (diff-max-length x)))))
                            (change-subblocks-values x
                                    :max-note-length-flag (max-note-length-flag x)
                                    :max-note-length (max-note-length x))
                        )
                    )
                    (if diff-chord-key
                        (progn
                            (setf   (diff-chord-key x) (- (diff-chord-key x)
                            ↪  diff-chord-key))
                            (setf   (chord-key x) (note-value-to-name (-
                            ↪  (name-to-note-value (chord-key parent)) (diff-chord-key
                            ↪  x))))
                            (change-subblocks-values x
                                  :chord-key (chord-key x))
                        )
                    )
                    (if diff-chord-quality
                        (progn
                            (setf   (diff-chord-quality x) (- (diff-chord-quality x)
                            ↪  diff-chord-quality))
                            (setf   (chord-quality x) (- (chord-quality parent)
                            ↪  (diff-chord-quality x)))
                            (change-subblocks-values x
                                  :chord-quality (chord-quality x))
                        )
                    )
                    (if diff-min-pitch
                        (progn
```

```
                            (setf   (diff-min-pitch x) (- (diff-min-pitch x) diff-min-pitch))
                            (setf   (min-pitch x) (- (min-pitch parent) (diff-min-pitch x)))
                            (change-subblocks-values x
                                    :min-pitch (min-pitch x))
                        )
                    )
                    (if diff-max-pitch
                        (progn
                            (setf   (diff-max-pitch x) (- (diff-max-pitch x) diff-max-pitch))
                            (setf   (max-pitch x) (- (max-pitch parent) (diff-max-pitch x)))
                            (change-subblocks-values x
                                    :max-pitch (max-pitch x))
                        )
                    )
                )
            )
        )
    )
)

;; https://stackoverflow.com/questions/59920951/defining-a-minimum-function-to-return-the-minimum-of-
(defun smallest (x y)
    (if (< x y) x y)
)

(defun biggest (x y)
    (if (< x y) y x)
)

(defun octaves-of-note (note)
    (let ((modnote (mod note 12)))
        (loop for i from 0 to (/ 128 12)
            collect (+ (* i 12) modnote)
            ;; collect (+ (* i -12) modnote)
        )
    )
)

; Create push and pull list from a voice object
(defun create-push-pull-int (input-chords quant)
    (let (temp
```

```lisp
         (next 0)
         (push (list))
         (pull (list '-1))
         (playing (list))
         (tree (om::tree input-chords))
         (pitch (to-pitch-list (om::chords input-chords))))
         (setq tree (second tree))
         (loop :for i :from 0 :below (length tree) :by 1 :do
            (let ((subtree (second (nth i tree))))
                (setq temp (read-tree-int (make-list quant :initial-element -1) (make-list quant :initial-element -1) (make-list quant :initial-element -1) subtree pitch 0 (/ quant (ceil-to-exp (get-length-tree subtree))) next))
                (setq push (append push (first temp)))
                (setq pull (append pull (second temp)))
                (setq playing (append playing (third temp)))
                (setf next (fourth temp))
            )
         )
         (list push pull playing))
)

;; ((4 4) (1 1 1 1))
; <tree> is the rhythm tree to read
; <pitch> is the ordered list of pitch (each element of push is represented by a list with the pitch of notes played on this quant)
; <pos> is the next position in push to add values
; <length> is the current duration of a note to add
; <next> is the index in pitch of the next notes we will add
;recursive function to read a rhythm tree and create push and pull
(defun read-tree-int (push pull playing tree pitch pos length next)
    (progn
        (loop :for i :from 0 :below (length tree) :by 1 :do
            (if (typep (nth i tree) 'list)
                (let (temp)
                    (setq temp (read-tree-int push pull playing (second (nth i tree)) pitch pos (/ (* length (first (nth i tree))) (length (second (nth i tree)))) next))
                    (setq push (first temp))
                    (setq pull (second temp))
                    (setq playing (third temp))
                    (setf next (fourth temp))
                    (setf pos (fifth temp))
                )
                (progn
                    (let (next-pitch)
```

```lisp
                        (if (> (nth i tree) 0)
                            (setq next-pitch (first (nth next pitch)))
                            (setq next-pitch -1)
                        )
                        (setf (nth pos push) next-pitch)
                        (loop :for j :from pos :below (+ pos (abs (* length (nth i tree)))) :by 1 :do
                            (setf (nth j playing) next-pitch)
                        )
                        (setf pos (+ pos (abs (* length (nth i tree)))))
                        (setf (nth (- pos 1) pull) next-pitch)
                        (if (> (nth i tree) 0)
                            (setf next (+ next 1))
                        )
                    )
                )
            )
        )
        (list push pull playing next pos)
    )
)

; Getting a list of chords and a rhythm tree from the playing list of intvar
(defun build-voice-int (sol push pull playing bars quant tempo)
    (let ((p-push (list))
          (p-pull (list))
          (p-playing (list))
          (chords (list))
          (tree (list))
          (ties (list))
          (prev 0)
          )
    (setq p-push (nconc p-push (mapcar (lambda (n) (* 100 (gil::g-values sol n))) push)))
    (setq p-pull (nconc p-pull (mapcar (lambda (n) (* 100 (gil::g-values sol n))) pull)))
    (setq p-playing (nconc p-playing (mapcar (lambda (n) (* 100 (gil::g-values sol n))) playing)))

    (setq count 0)
    ;; (setq rest 0)
    (loop :for b :from 0 :below bars :by 1 :do
        (if (< (nth (* b quant) p-playing) 0)
            (setq rest 1)
            (setq rest 0)
        )
        (setq rhythm (list))
        (loop :for q :from 0 :below quant :by 1 :do
```

```
            (setq i (+ (* b quant) q))
            (cond
                ((>= (nth i p-push) 0)
                     ; if rhythm impulse
                     (progn
                        (setq duration 0)
                        (setq j (+ i 1))
                        (loop
                            (if (>= j (length p-pull))
                                (setq duration (* (floor 60000 (* tempo quant)) (- j i)))
                                (return)
                            )
                            (if (>= (nth j p-pull) 0)
                                (if (= (nth j p-pull) (nth i p-push))
                                    (progn
                                        (setq duration (* (floor 60000 (* tempo quant)) (-
                                        ↪  j i)))
                                        (return)
                                    )
                                )
                            )
                            (incf j)
                        )
                        (setq chord (make-instance 'chord :LMidic (list (nth i p-push))
                        ↪  :Ldur (list duration)))
                        (setq chords (nconc chords (list chord)))
                        (cond
                            ((= rest 1)
                                (progn
                                    (setq rhythm (nconc rhythm (list (* -1 count))))
                                    (setq rest 0)))
                            ((/= q 0)
                                (setq rhythm (nconc rhythm (list count))))
                        )
                        (setq count 1))
                )
                ((and (< (nth i p-playing) 0) (= rest 0))
                    (setq rest 1)
                    (if (> count 0)
                        (setq rhythm (nconc rhythm (list count)))
                    )
                    (setq count 1)
                )
                ; else
                (t (setq count (+ count 1)))
            )
```

```
        )
        (if (= rest 1)
            (setq rhythm (nconc rhythm (list (* -1 count))))
            (setq rhythm (nconc rhythm (list count)))
        )
        (setq count 0)
        (setq rhythm (list '(4 4) rhythm))

        (setq tree (nconc tree (list rhythm)))
    )
    (setq tree (list '? tree))
    (list chords tree)
    )
)

; returns the list of intervals defining a given mode
(defun get-scale-chord (mode)
    (cond
        ((string-equal mode "Major")
            (list 2 2 1 2 2 2 1)
        )
        ((string-equal mode "Minor")
            (list 2 1 2 2 1 2 2)
        )
        ((string-equal mode "Diminished")
            (list 2 1 2 1 2 1 2)
        )
        ((string-equal mode "Augmented")
            (list 3 1 3 1 3 1)
        )
    )
)

(defun build-chordset (chord offset)
    (let ((noteset (build-notesets chord offset)) (chordset (list)))
        (loop :for i :from 0 :below (length (first noteset)) :do
            (setq chordset (nconc chordset (list (list (nth i (nth 0 noteset)) (nth i (nth
            ↪  1 noteset)) (nth i (nth 2 noteset))))))
        )
        chordset
    )
)
```

D.4.2 sources/melodizer-utils.lisp

This file contains useful functions that weren't created for Meldoizer Rock.

```
(in-package :mldz)

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

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

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

;converts the name of a note to its value
(defmethod name-to-note-value (name)
    (cond
        ((string-equal name "C") 60)
        ((string-equal name "C#") 61)
        ((string-equal name "D") 62)
        ((string-equal name "Eb") 63)
```

```
        ((string-equal name "E") 64)
        ((string-equal name "F") 65)
        ((string-equal name "F#") 66)
        ((string-equal name "G") 67)
        ((string-equal name "Ab") 68)
        ((string-equal name "A") 69)
        ((string-equal name "Bb") 70)
        ((string-equal name "B") 71)
    )
)

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

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


; finds the biggest element in a list of lists
(defun max-list-list (L)
    (cond
        ((null (car L)) nil); the list is empty -> return nil
        ((null (cdr L)) (max-list (car L))); the list has 1 element -> return it
        (T
```

```lisp
            (let ((head (max-list (car L))); default behavior
                 (tailMax (max-list-list (cdr L))))
                (if (> head tailMax) head tailMax)
            )
        )
    )
)

; create a list from min to max by step
(defun range (max &key (min 0) (step 1))
   (loop :for n :from min :below max :by step
      :collect n))

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

```
                                    (om::omNG-make-new-instance
                                        (output-motif (om::object self))
                                        (format nil "motif ~D" (1+ indx)); name of the window
                                    )
                                )
                            )
                        )
                        ((string-equal output "output-phrase")
                            (setf (output-phrase (om::object self)) (nth (om::om-get-selected-item-index m) data))
                            (let ((indx (om::om-get-selected-item-index m)))
                                (om::openeditorframe
                                    (om::omNG-make-new-instance
                                        (output-phrase (om::object self))
                                        (format nil "phrase ~D" (1+ indx)); name of the window
                                    )
                                )
                            )
                        )
                        ((string-equal output "output-period")
                            (setf (output-period (om::object self)) (nth (om::om-get-selected-item-index m) data))
                            (let ((indx (om::om-get-selected-item-index m)))
                                (om::openeditorframe
                                    (om::omNG-make-new-instance
                                        (output-period (om::object self))
                                        (format nil "period ~D" (1+ indx))
                                    )
                                )
                            )
                        )
                    )
      )
    )
  )
)

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

```lisp
)

;function to get the duration (in ms) of the notes
(defmethod voice-durs ((self voice))
  "on passe de voice a chord-seq juste pour avoir les onsets"
    (let ((obj (om::objfromobjs self (make-instance 'om::chord-seq))))
        (om::ldur obj)
    )
)

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

```

```
(defun get-chord (quality)
    (cond
        ((string-equal quality "Major")
            (list 4 3 5)
        )
        ((string-equal quality "Minor")
            (list 3 4 5)
        )
        ((string-equal quality "Augmented")
            (list 4 4 4)
        )
        ((string-equal quality "Diminished")
            (list 3 3 6)
        )
        ((string-equal quality "Major 7")
            (list 4 3 4 1)
        )
        ((string-equal quality "Minor 7")
            (list 3 4 3 2)
        )
        ((string-equal quality "Dominant 7" )
            (list 4 3 3 2)
        )
        ((string-equal quality "Minor 7 flat 5")
            (list 3 3 4 2)
        )
        ((string-equal quality "Diminished 7")
            (list 3 3 3 3)
        )
        ((string-equal quality "Minor-major 7")
            (list 3 4 4 1)
        )

        ; TODO gérer les accords 9 ou +
        ((string-equal quality "Major 9")
            (list 3 4 5)
        )
        ((string-equal quality "Minor 9")
            (list 4 3 5)
        )
        ((string-equal quality "9 Augmented 5")
            (list 3 4 5)
        )
        ((string-equal quality "9 flatted 5")
            (list 3 4 5)
        )
```

```lisp
        ((string-equal quality "7 flat 9")
            (list 4 3 5)
        )
        ((string-equal quality "Augmented 9")
            (list 3 4 5)
        )
        ((string-equal quality "Minor 11")
            (list 3 4 5)
        )
        ((string-equal quality "Major 11")
            (list 4 3 5)
        )
        ((string-equal quality "Dominant 11")
            (list 3 4 5)
        )
        ((string-equal quality "Dominant # 11")
            (list 4 3 5)
        )
        ((string-equal quality "Major # 11")
            (list 3 4 5)
        )
    )
)

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

; function to get all notes playable on top of a given chord CHECK WHAT NOTES CAN BE
↪ PLAYED FOR OTHER CASES THAN M/m
(defun get-admissible-notes (chords mode inversion)
    (let ((return-list '()))
        (cond
            ((string-equal mode "major"); on top of a major chord, you can play either of
            ↪  the notes from the chord though the preferred order is 1-5-3
```

```
                (setf return-list (reduce #'cons
                    (get-all-notes (first chords))
                    :initial-value return-list
                    :from-end t
                ))
                (setf return-list (reduce #'cons
                    (get-all-notes (second chords))
                    :initial-value return-list
                    :from-end t
                ))
                (setf return-list (reduce #'cons
                    (get-all-notes (third chords))
                    :initial-value return-list
                    :from-end t
                ))
            )
            ((string-equal mode "minor"); on top of a minor chord, you can play either of
            ↪   the notes from the chord though the preferred order is 1-5-3
                (setf return-list (reduce #'cons
                    (get-all-notes (first chords))
                    :initial-value return-list
                    :from-end t
                ))
                (setf return-list (reduce #'cons
                    (get-all-notes (second chords))
                    :initial-value return-list
                    :from-end t
                ))
                (setf return-list (reduce #'cons
                    (get-all-notes (third chords))
                    :initial-value return-list
                    :from-end t
                ))
            )
            ((string-equal mode "diminished"); only the third can be played on top of
            ↪   diminished chords
                (cond
                    ((= inversion 0)
                        (setf return-list (reduce #'cons
                            (get-all-notes (second chords))
                            :initial-value return-list
                            :from-end t
                        ))
                    )
                    ((= inversion 1)
                        (setf return-list (reduce #'cons
```

```
                            (get-all-notes (first chords))
                            :initial-value return-list
                            :from-end t
                        ))
                    )
                    ((= inversion 2)
                        (setf return-list (reduce #'cons
                            (get-all-notes (third chords))
                            :initial-value return-list
                            :from-end t
                        ))
                    )
                )
            )
        )
    )
)

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

;makes a list (name voice-instance) from a list of voices:
;(from Karim Haddad)
(defun make-data-sol (liste)
```

```lisp
  (loop for l in liste
        for i from 1 to (length liste)
        collect (list (format nil "solution ~D: ~A"  i l) l)))


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

; Create push and pull list from a voice object
(defun create-push-pull (input-chords quant)
    (let (temp
         (next 0)
         (push (list))
         (pull (list '-1))
        ;; (pull (list))
         (playing (list))
         (tree (om::tree input-chords))
         (pitch (to-pitch-list (om::chords input-chords))))
         (setq tree (second tree))
         (print "before chords")
         (print input-chords)
         (print "tree:")
         (print tree)
         (loop :for i :from 0 :below (length tree) :by 1 :do
            (print "call to read-tree")
            ;; bugs on the first call to read-tree with this error :
            ;; ERROR: Cannot take CDR of 1.
            (setq temp (read-tree (make-list quant :initial-element -1) (make-list quant
            ↪  :initial-element -1) (make-list quant :initial-element -1) (second (nth i
            ↪  tree)) pitch 0 quant next))
            (setq push (append push (first temp)))
            (setq pull (append pull (second temp)))
            (setq playing (append playing (third temp)))
            (setf next (fourth temp))
         )
```

```
         (list push pull playing))
)

;; (car cdr)

;; ((4 4) (1 1 1 1))
; <tree> is the rhythm tree to read
; <pitch> is the ordered list of pitch (each element of push is represented by a list with the pitch of notes played on this quant)
; <pos> is the next position in push to add values
; <length> is the current duration of a note to add
; <next> is the index in pitch of the next notes we will add
;recursive function to read a rhythm tree and create push and pull
(defun read-tree (push pull playing tree pitch pos length next)
    (print "in read-tree")
    (progn
        (print "Pitch:")
        (print pitch)
        (setf length (/ length (ceil-to-exp (length tree))))
        (print "pre-loop")
        (loop :for i :from 0 :below (length tree) :by 1 :do
            (if (typep (nth i tree) 'list)
                (let (temp)
                    (print "if")
                    (setq temp (read-tree push pull playing (second (nth i tree)) pitch pos length next))
                    (setq push (first temp))
                    (setq pull (second temp))
                    (setq playing (third temp))
                    (setf next (fourth temp))
                    (setf pos (fifth temp))
                )
                (progn
                    (print "else")
                    (setf (nth pos push) (nth next pitch))
                    (loop :for j :from pos :below (+ pos (* length (nth i tree))) :by 1 :do
                        (setf (nth j playing) (nth next pitch))
                    )
                    (setf pos (+ pos (* length (nth i tree))))
                    (setf (nth (- pos 1) pull) (nth next pitch))
                    (setf next (+ next 1))
                )
            )
        )
        (list push pull playing next pos)
```

```
    )
)

; <input-chords> is the voice objects for the chords
; <quantOrig> quantification used by melodizer
; Return a list in which each element i represent a note starting at a time i*quant
; -1 means no note starting at that time, a chord object means multiple note starting
(defun create-push (input-chords quantOrig)
    (let ((note-starting-times (voice-onsets input-chords))
          (quant (/ (second (first (om::tempo input-chords))) (/ quantOrig 16)))
          (tree (om::tree input-chords))
          (push-list (list))
          (chords (to-pitch-list (om::chords input-chords))) ; get chords list
         )
         (setf note-starting-times (mapcar (lambda (n) (/ n quant)) note-starting-times)) ; dividing note-starting-times by quant
         (loop :for j :from 0 :below (+ (max-list note-starting-times) 1) :by 1 :do
            (if (= j (car note-starting-times)); if j == note-starting-times[0]
                (progn
                    (setq push-list (nconc push-list (list (car chords))))
                    (setf chords (cdr chords))
                    (setf note-starting-times (cdr note-starting-times))) ;add chords[0] to push and prune qt[0] and pchords[0]
                (setq push-list (nconc push-list (list -1)))) ; else add -1 to push
        )
    )
)


; <input-chords> is the voice objects for the chords
; <quant> NOT USED YET (FORCED TO 500) smallest possible note length
; Return a list in which each element i represent a note stopping at a time i*quant
; -1 means no note stop at that time, a chord object means multiple note starting
(defun create-pull (input-chords)
    (let ((note-starting-times (voice-onsets input-chords)) ; note-starting-times = start time of each chord
          (note-dur-times (voice-durs input-chords)) ; note-dur-times = duration of each note
          (note-stopping-times (list))
          (quant 500)
          (pull-list (list))
          (pitch (to-pitch-list (om::chords input-chords))) ; get chords list
         )
         (setf note-starting-times (mapcar (lambda (n) (/ n quant)) note-starting-times)) ; dividing note-starting-times by quant
         (setf note-dur-times (mapcar (lambda (n) (mapcar (lambda (m) (/ m quant)) n)) note-dur-times)) ; dividing note-dur-times by quant
```

```
         (loop :for j :from 0 :below (length note-starting-times) :by 1 :do
             (setq note-stopping-times (nconc note-stopping-times (list (mapcar (lambda (n) (+ n (nth j note-starting-times))) (nth j note-dur-times))))) ; Adding note-starting-times to note-dur-times to get note-stopping-times
        )
        (loop :for j :from 0 :below (+ (max-list-list note-stopping-times) 1) :by 1 :do
              (setq pull-list (nconc pull-list (list -1))))
        (loop for l in note-stopping-times
              for k in pitch do
            (loop for i in l
                  for j in k do
                  (if (typep (nth i pull-list) 'list)
                      (setf (nth i pull-list) (nconc (nth i pull-list) (list j)))
                      (setf (nth i pull-list) (list j)))
             )
        )
    )
)

; reformat a scale to be a canvas of pitch and not intervals
(defun adapt-scale (scale)
    (let ((major-modified (list (first scale))))
         (loop :for i :from 1 :below (length scale) :by 1 :do
            (setq major-modified (nconc major-modified (list (+ (nth i scale) (nth (- i 1) major-modified)))))
         )
    (return-from adapt-scale major-modified)
    )
)

; build the list of acceptable pitch based on the scale and a key offset
(defun build-scaleset (scale offset)
    (let ((major-modified (adapt-scale scale))
          (scaleset (list)))
        (loop :for octave :from -1 :below 11 :by 1 append
              (setq scaleset (nconc scaleset (mapcar (lambda (n) (+ (+ n (* octave 12)) offset)) major-modified)))
        )
        (setq scaleset (remove-if 'minusp scaleset))
    )
)

; build the list of acceptable pitch based on the scale and a key offset
(defun build-notesets (chord offset)
    (let ((chord-modified (adapt-scale chord))
          (notesets (list)))
```

```lisp
        (loop :for i :from 0 :below (length chord-modified) :by 1 :do
            (setq noteset (list))
            (loop :for octave :from -1 :below 11 :by 1 append
                  (setq noteset (nconc noteset (list (+ (+ (nth i chord-modified) (* octave 12)) offset))))
            )
            (setq noteset (remove-if 'minusp noteset))
            (setq notesets (nconc notesets (list noteset)))
        )
        notesets
    )
)



; <chords> a list of chord object
; Return the list of pitch contained in chords in midi format
(defun to-pitch-list (chords)
     (loop :for n :from 0 :below (length chords) :by 1 collect (to-midi (om::lmidic (nth n chords))))
)


; Getting a list of chords and a rhythm tree from the playing list of intvar
(defun build-voice (sol push pull bars quant tempo)
    (let ((p-push (list))
          (p-pull (list))
          (chords (list))
          (tree (list))
          (ties (list))
          (prev 0)
          )

    (setq p-pull (nconc p-pull (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) pull)))
    (setq p-push (nconc p-push (mapcar (lambda (n) (to-midicent (gil::g-values sol n))) push)))
    (setq count 1)
    (loop :for b :from 0 :below bars :by 1 :do
        (if (not (nth (* b quant) p-push))
            (setq rest 1)
            (setq rest 0)
        )
        (setq rhythm (list))
        (loop :for q :from 0 :below quant :by 1 :do
            (setq i (+ (* b quant) q))
```

```
            (cond
                ((nth i p-push)
                     ; if rhythm impulse
                     (progn
                        (setq durations (list))
                        (loop :for m :in (nth i p-push) :do
                            (setq j (+ i 1))
                            (loop
                                (if (nth j p-pull)
                                    (if (find m (nth j p-pull))
                                        (progn
                                            (setq dur (* (floor 60000 (* tempo quant)) (-
                                            ↪  j i)))
                                            (setq durations (nconc durations (list dur)))

                                            (return)
                                        )
                                    )
                                )
                                (incf j)
                            )
                        )
                        (setq chord (make-instance 'chord :LMidic (nth i p-push) :Ldur
                        ↪  durations))
                        (setq chords (nconc chords (list chord)))
                        (cond
                            ((= rest 1)
                                (progn
                                    (setq rhythm (nconc rhythm (list (* -1 count))))
                                    (setq rest 0)))
                            ((/= q 0)
                                (setq rhythm (nconc rhythm (list count))))
                        )
                        (setq count 1))
                )
                ; else
                (t (setq count (+ count 1)))
            )
        )
        (if (= rest 1)
            (setq rhythm (nconc rhythm (list (* -1 count))))
            (setq rhythm (nconc rhythm (list count)))
        )
        (setq count 0)
        (setq rhythm (list '(4 4) rhythm))

```

```
        (setq tree (nconc tree (list rhythm)))
    )
    (setq tree (list '? tree))

    (list chords tree)
    )
)

(defun build-chord-seq (sol push pull bars quant tempo)
    (let ((p-push (list))
          (p-pull (list))
          (chords (list))
          (durations (list))
          (onsets (list)))

        (setq p-pull (nconc p-pull (mapcar (lambda (n) (to-midicent (gil::g-values sol n)))
        ↪  pull)))
        (setq p-push (nconc p-push (mapcar (lambda (n) (to-midicent (gil::g-values sol n)))
        ↪  push)))

        (loop :for i :from 0 :below (+ (* bars quant) 1) :do
            (if (nth i p-push)
                (progn
                    (setq onset (* (/ 60000 (* tempo (/ quant 4))) i))
                    (setq duration (list))
                    (loop :for m :in (nth i p-push) :do
                        (setq j (+ i 1))
                        (loop
                            (if (nth j p-pull)
                                (if (find m (nth j p-pull))
                                    (progn
                                        (setq dur (* (/ 60000 (* tempo (/ quant 4))) (- j
                                        ↪  i)))
                                        (setq duration (nconc duration (list dur)))

                                        (return)
                                    )
                                )
                            )
                            (incf j)
                        )
                    )
                    (setq chords (nconc chords (list (nth i p-push))))
                    (setq durations (nconc durations (list duration)))
                    (setq onsets (nconc onsets (list onset)))
                )
```

```
            )
        )

        (list chords onsets durations)
    )
)

;return T if the two list have the same elements (order doesn't matter)
(defun compare (l1 l2)
  (and (subsetp l1 l2) (subsetp l2 l1)))

; return the quant value based on the index selected
(defun get-quant (str)
  (cond ((string= str "1 bar") 1)
    ((string= str "1/2 bar") 2)
    ((string= str "1 beat") 4)
    ((string= str "1/2 beat") 8)
    ((string= str "1/4 beat") 16)
    ((string= str "1/8 beat") 32)
    ((string= str "1/3 bar") 3)
    ((string= str "1/6 bar") 6)
    ((string= str "1/3 beat") 12)
    ((string= str "1/6 beat") 24)
    ((string= str "1/12 beat") 48)
    ((not str) 192))
)

; return the quant value based on the index selected
(defun get-length (str)
  (cond ((string= str "1 bar") 192)
    ((string= str "1/2 bar") 96)
    ((string= str "1 beat") 48)
    ((string= str "1/2 beat") 24)
    ((string= str "1/4 beat") 12)
    ((string= str "1/8 beat") 6)
    ((string= str "1/3 bar") 64)
    ((string= str "1/6 bar") 32)
    ((string= str "1/3 beat") 16)
    ((string= str "1/6 beat") 8)
    ((string= str "1/12 beat") 4)
    ((not str) 1))
)

; shuffles a list
; from https://gist.github.com/shortsightedsid/62d0ee21bfca53d9b69e
(defun list-shuffler (input-list &optional accumulator)
```

```lisp
  "Shuffle a list using tail call recursion."
  (if (eq input-list nil)
      accumulator
      (progn
        (rotatef (car input-list)
                 (nth (random (length input-list)) input-list))
        (list-shuffler (cdr input-list)
                                (append accumulator (list (car input-list)))))))

(defun set-percent-diff (sp percent-diff sol push pull playing)
    (let ((p-push (list))
          (p-pull (list))
          (p-playing (list)))
          (print "set-percent-diff")
        (setq p-push (nconc p-push (mapcar (lambda (n) (gil::g-values sol n)) push)))
        (setq p-pull (nconc p-pull (mapcar (lambda (n) (gil::g-values sol n)) pull)))
        (setq p-playing (nconc p-playing (mapcar (lambda (n) (gil::g-values sol n))
        ↪  playing)))

        (loop :for i :from 0 :below (length playing) :by 1
        do
            (if (< (random 101) percent-diff)
                (gil::g-rel sp (nth i playing) gil::SRT_NQ (nth i p-playing))
            )
        )
    )
)
```

D.5 GiL Example

```lisp
(in-package :mldz)

; DUMMY-PROBLEM
; This function creates a CSP by creating the space and the variables, posting the
↪  branching, specifying
; the search options and creating the search engine.
(defun dummy-problem ()
    (let ((sp (gil::new-space)); create the space;
```

```lisp
        vars se tstop sopts max id-list)

        ;initialize the variables
        (setq vars (gil::add-int-var-array sp 3 1 4))

        ; constraints
        (gil::g-count-array sp vars (list 1 1 1 1) gil::IRT_EQ 2)
        ; branching
        (gil::g-branch sp vars gil::INT_VAR_SIZE_MIN gil::INT_VAL_MIN)

        ;time stop
        (setq tstop (gil::t-stop)); create the time stop object
        (gil::time-stop-init tstop 500000); initialize it (time is expressed in ms)

        (setq sopts (gil::search-opts)); create the search options object
        (gil::init-search-opts sopts); initialize it
        (gil::set-time-stop sopts tstop); set the timestop object to stop the search if it takes too long

        ; search engine
        (setq se (gil::search-engine sp (gil::opts sopts) gil::BAB)); branch and bound search-engine, remove t for dfs
        (print se)

        (print "CSP constructed")
        ; return
        (list se vars tstop sopts)
    )
)

; SEARCH-NEXT-DUMMY-PROBLEM
; <l> is a list containing in that order the search engine for the problem, the variables
; this function finds the next solution of the CSP using the search engine given as an argument
(defun search-next-dummy-problem (l)
    (let ((se (first l))
         (pitch* (second l))
         (tstop (third l))
```

```
         (sopts (fourth l))
         sol pitches)

        (gil::time-stop-reset tstop);reset the tstop timer before launching the
        ↪  search
        (setq sol (gil::search-next se)); search the next solution, sol is the
        ↪  space of the solution
        (if (null sol)
            (error "No more solutions")
        )
        ; print the solution from GiL
        (setq pitches (gil::g-values sol pitch*)); store the values of the
        ↪  solution
        (print "pitches")
        (print pitches)
    )
)
```

Appendix E

Collection of Scores

The following chapter contains all the scores cited in this thesis. It contains two categories of scores:

- The scores **produced** by Melodizer Rock as a result of the examples of Chapter 6
- The scores used as an example in the explanation of the thesis or a source melody for the examples of Chapter 6

E.1 Obtained Scores

This section will gives the score produced by Melodizer Rock when tested on the examples of chapter 6

E.1.1 Example 6.1

Those are the two first scores obtained with a simple A block and only a few constraints. Figures E.1 E.2

E.1.2 Example 6.2

Those are the two first scores obtained for an example with both an A block and a B block. Figures E.3 E.4

E.1.3 Example 6.3

Those are the two first obtained scores when testing Melodizer Rock on a structure with two A blocks and a source melody. Figures E.5 E.6

Figure E.1: First solution to an example with a single A block

Figure E.2: Second solution to an example with a single A block

E.1.4 Example 6.4

Those are the first two results of Melodizer Rock when tested on a full AABA structure. Figures E.7 E.8

E.1.5 Example 6.5

Those are the two first scores obtained with the last example, that is, a full AABA structure and a `melody-source-A` and `melody-source-B`. Figures E.9 E.10 E.11 E.12

E.2 External Scores

E.2.1 I'll Be There by The Jackson 5

Example of the song *I'll Be There* by the Jackson 5 in Figure E.13.

E.2.2 Every Breath You Take by The Police

This score was used as a source melody for the example 6.5.

EVERY BREATH YOU TAKE

Written and Composed by
G.M. SUMNER

Moderate Rock


C
D
Dsus
ev - 'ry bond you break, ev - 'ry step you take, I'll be watch-ing you.
Em
D7sus
G
Ev - 'ry sin - gle day,
Em
ev - 'ry word you say, ev - 'ry game you play,
C
D
Dsus
G
ev - 'ry night you stay, I'll be watch-ing you.


C C/Bb Am7
Oh, can't you see you be - long to
G A7
me? How my poor heart aches
D D7sus
with ev - 'ry step you take. Ev - 'ry move you
G Em
make, ev - 'ry vow you break,


C
D
Dsus
ev - 'ry smile _ you fake, ev - 'ry claim _ you stake, I'll be watch-ing you.
Em
To Coda
Eb
3fr
Since you've gone, _ I been lost _
F
_ with - out _ a trace, I dream at night I can on - ly see _ your face.
Eb
3fr
F
I look a-round, but it's you I can't _ re-place. I feel so cold and I


Eb
3fr
long for your em-brace. I keep cry - ing, ba - by, ba - by, please
G
Em
C
D
1
Em
D


G
2
G
D.S. al Coda
Oh, can't you
CODA
Em
C
Ev - 'ry move you make, ev - 'ry step you take,
D
Dsus
Em
I'll be watch-ing you.
D7sus
I'll be watch - ing


G
Em
you. Ev - 'ry breath _ take, ev - 'ry move _ you make, ev - 'ry bond _ you break,
you. (Ev - 'ry move _ you make, ev - 'ry vow _ you break, ev - 'ry smile _ you fake,
C
G
ev - 'ry step ___ you take,
ev - 'ry claim ___ you stake,
ev - 'ry sin - gle day,
I'll be watch - ing you.
Em
ev - 'ry word ___ you say, ev - 'ry game ___ you play,
Repeat and Fade
Optional Ending
C
C
G
ev - 'ry night _ you stay.)
I'll be watch - ing
I'll be watch - ing you.

Figure E.3: First solution of an example with an *A* block and a *B* block

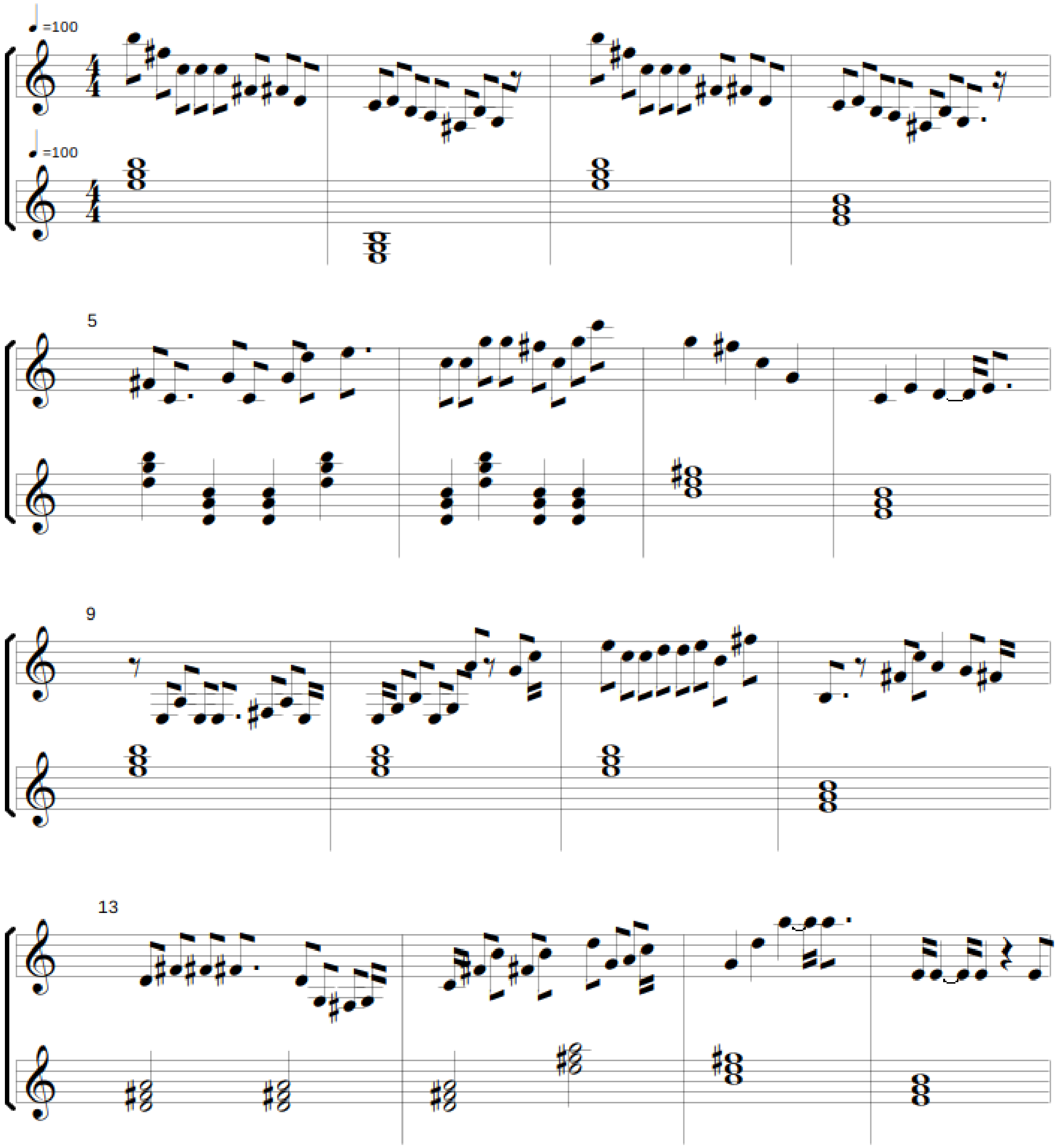


Figure E.4: Second solution of an example with an A block and a B block

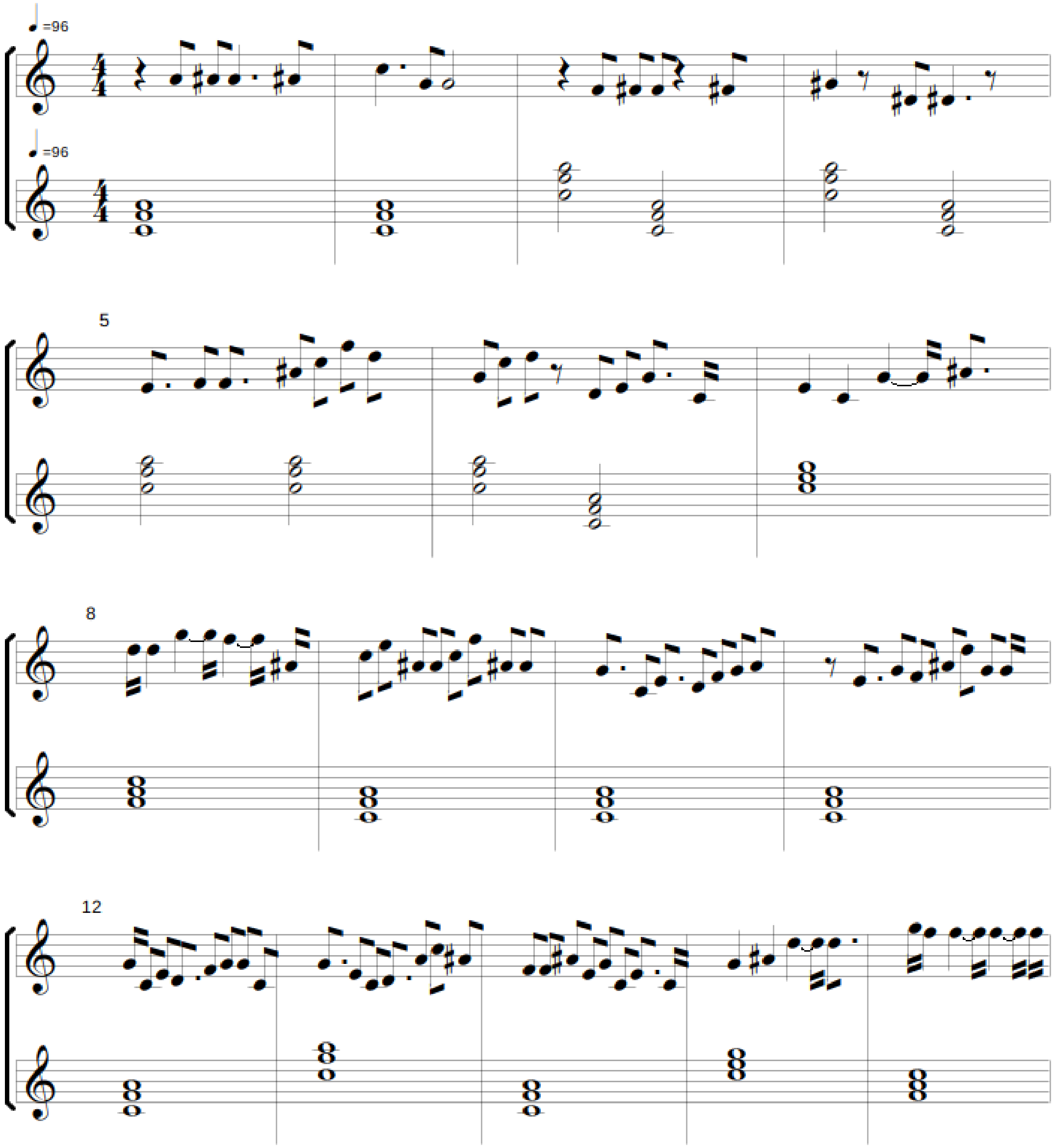


Figure E.5: First solution of an example with two A blocks and a source melody

Figure E.6: Second solution of an example with two *A* blocks and a source melody

Figure E.7: First solution of an example with an $AABA$ structure

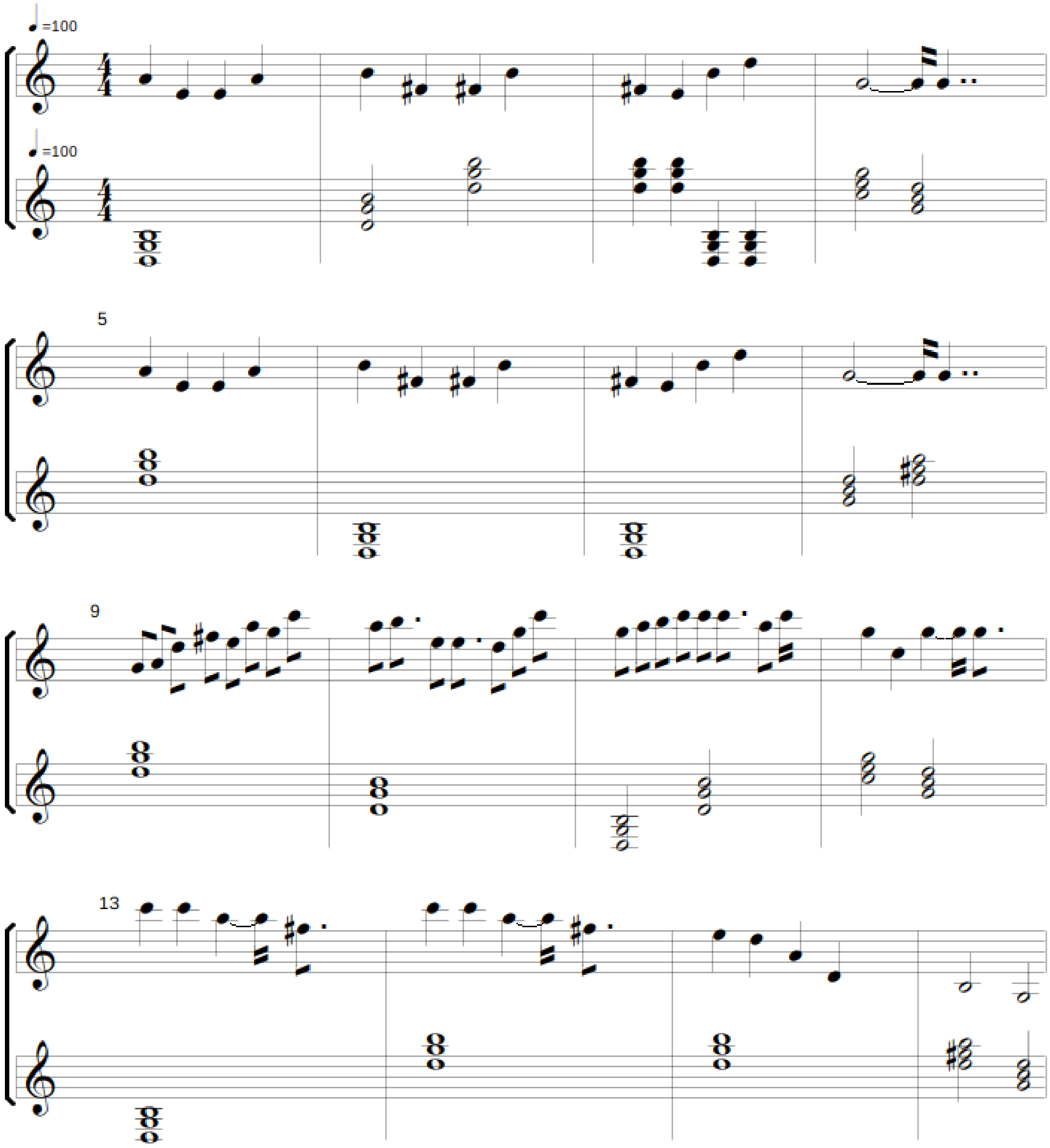


Figure E.8: Second solution of an example with an *AABA* structure

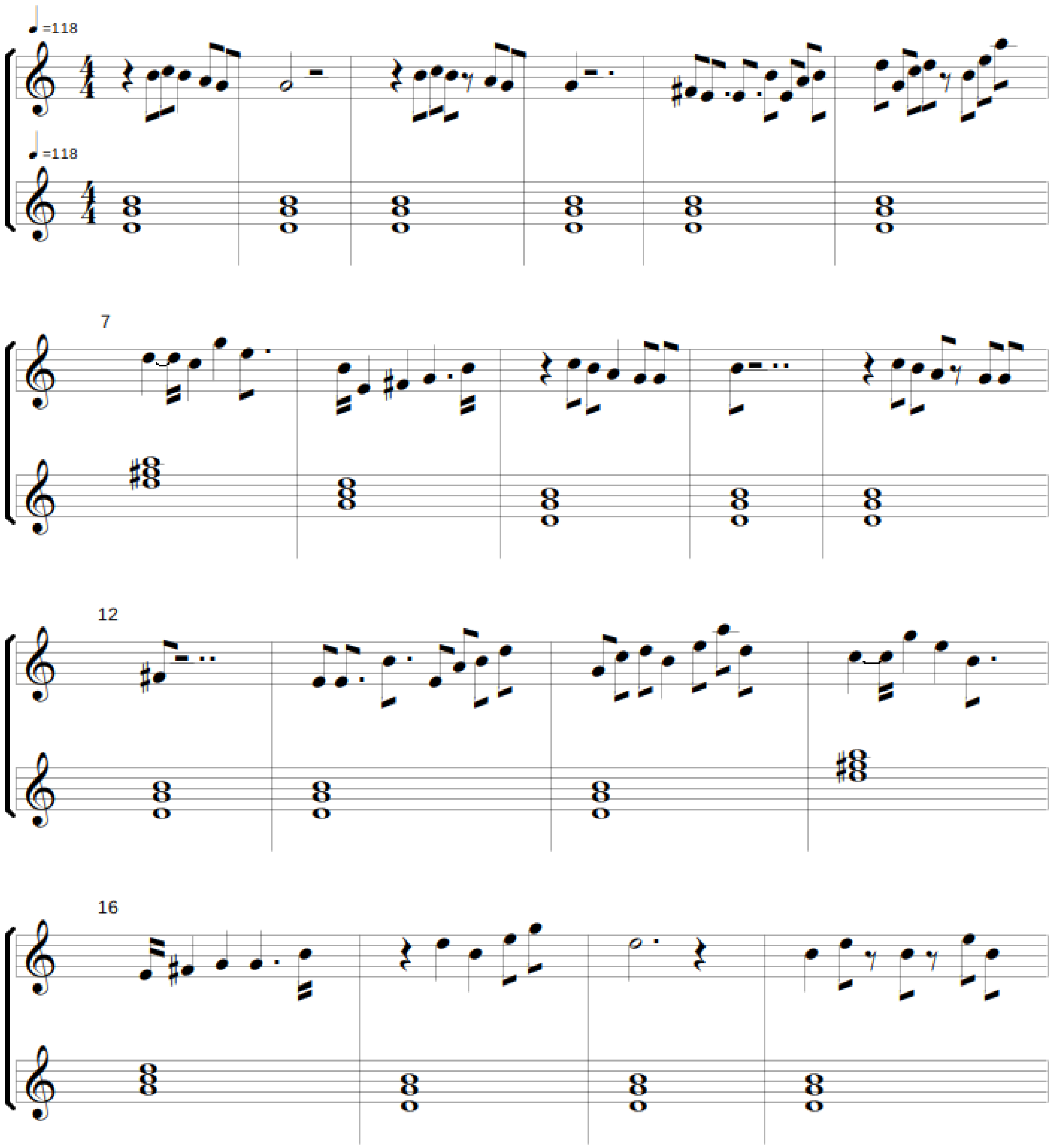


Figure E.9: First page of the first solution given by Melodizer Rock, with the inputs of *Every Breath You take* [12] for an *AABA* structure

Figure E.10: Second page of the first solution given by Melodizer Rock, with the inputs of *Every Breath You take* [12] for an *AABA* structure

Figure E.11: First page of the second solution given by Melodizer Rock, with the inputs of *Every Breath You take* [12] for an *AABA* structure [12]

Figure E.12: Second page of the second solution given by Melodizer Rock, with the inputs of *Every Breath You take* [12] for an *AABA* structure[12]

Figure E.13: "I'll Be There" by the Jackson 5 score as given by Drew Nobile in [4]

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
École polytechnique de Louvain
Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/epl**